1. Installation

MARS (MIPS Assembler and Runtime Simulator) is a software tool used to assemble and run MIPS assembly language programs. To install MARS:

● Download the MARS .jar file from the [official website.]()

● Ensure Java Runtime Environment (JRE) is installed on your machine.

● Run the MARS .jar file by double-clicking it or using the command line:

```
java -jar Mars4_5.jar
```

MARS provides an easy-to-use interface to write, assemble, and execute MIPS assembly code, making it an excellent tool for learning and development.

2. Concatenate

(a) Code Explanation:

```
concat:
    lb $t3, 0($t0)           # Load byte from str1/str2
    beq $t3, $zero, concat_end  # End if null terminator is reached
    sb $t3, 0($t2)           # Store byte in result
    addi $t0, $t0, 1         # Move to next byte of str1/str2
    addi $t2, $t2, 1         # Move to next byte in result
    j concat                 # Repeat
concat_end:
    sb $zero, 0($t2)         # Null-terminate the result string
    jr $ra                   # Return from function
```

- **`lb $t3, 0($t0)`**: Loads a byte from the memory location pointed to by `$t0` (which holds the address of `str1`) into register `$t3`.
- **`beq $t3, $zero, concat_end`**: Checks if the byte loaded in `$t3` is a null terminator (end of string). If it is, the function jumps to the label `concat_end`, signaling the end of the concatenation for this string.
- **`sb $t3, 0($t2)`**: Stores the byte from `$t3` (from `str1`) into the memory location pointed to by `$t2` (the current position in the `result` buffer).
- **`addi $t0, $t0, 1`**: Increments `$t0` to point to the next character in `str1`.
- **`addi $t2, $t2, 1`**: Increments `$t2` to point to the next position in the `result` string where the next character will be written.
- **`j concat`**: Jumps back to the start of the `concat` loop, where the next byte will be loaded and processed.
- **`concat_end:`**: When the null terminator of `str1` is reached, the loop exits.
- **`sb $zero, 0($t2)`**: This stores a null terminator at the end of the concatenated string in `result`, ensuring that the string is properly terminated.

- **`jr $ra`**: This returns control to the calling function (main program) by jumping to the address in `$ra`.

```
    # Concatenate str2
    la $t0, str2
    jal concat
```

- **`la $t0, str2`**: Loads the address of `str2` into `$t0`. Now the same process of concatenation will happen for `str2`.
- **`jal concat`**: Jumps to the `concat` function again to append `str2` to the result.
- **`jal`** stands for **Jump And Link.**

```
remove_newline:
    la $t1, str1                # Load address of string
remove_loop:
    lb $t2, 0($t1)              # Load byte from the string
    beq $t2, $zero, remove_end  # If null terminator, end
    beq $t2, 10, newline_found  # If newline (ASCII 10), replace it
    addi $t1, $t1, 1            # Move to next character
    j remove_loop
newline_found:
    sb $zero, 0($t1)            # Replace newline with null terminator
remove_end:
    jr $ra                      # Return from function
```

- **`la $t1, str1`**: Loads the starting address of the string `str1` into register `$t1`. `$t1` will be used to iterate through the characters in the string one by one.
- **`lb $t2, 0($t1)`**: Loads the byte (character) at the memory address pointed to by `$t1` into register `$t2`. The `lb` instruction stands for "load byte", meaning it reads one character from the string.
- **`beq $t2, $zero, remove_end`**: Compares the byte in `$t2` with zero (`$zero` is always 0 in MIPS). If `$t2` contains the null terminator (`\0`, which has a value of 0), the function jumps to the `remove_end` label, indicating the end of the string, and exits the loop.
- **`beq $t2, 10, newline_found`**: Compares the byte in `$t2` with the ASCII value for a newline (which is 10). If a newline is found, the function jumps to the `newline_found` label to replace it with a null terminator.
- **`addi $t1, $t1, 1`**: Increments `$t1` by 1, moving to the next character in the string.
- **`j remove_loop`**: Unconditionally jumps back to the `remove_loop` label, repeating the process of checking each character in the string.

- **`sb $zero, 0($t1)`**: Stores the value `0` (null terminator) at the memory address pointed to by `$t1`, which is where the newline was found. This effectively replaces the newline character (`\n`) with a null terminator (`\0`), truncating the string at this point.
- **`remove_end:`**: This is the label where the function jumps if the loop reaches the null terminator without finding a newline, or after the newline has been replaced.
- **`jr $ra`**: This instruction means "jump register" and returns control to the calling function by jumping to the address stored in the `$ra` register (return address).

3. Input / Output: Palindrome

(a) Code Explanation:

```
# Function to check if the string is a palindrome
is_palindrome:
    la $t1, input_str      # Start of string
    move $t2, $t1          # Initialize pointer to find the end of the 
string

    # Find the end of the string
    loop_find_end:
        lb $t3, 0($t2)
        beq $t3, $zero, done_find_end  # Stop at null terminator
        addi $t2, $t2, 1               # Move to the next character
        j loop_find_end
    done_find_end:
        subi $t2, $t2, 1   # Point to the last character before the null 
terminator

    # Compare characters from both ends to check palindrome
    loop_compare:
        lb $t3, 0($t1)     # Load character from the start
        lb $t4, 0($t2)     # Load character from the end
        bne $t3, $t4, not_palindrome  # If mismatch, it's not a palindrome
        addi $t1, $t1, 1   # Move start pointer forward
        subi $t2, $t2, 1   # Move end pointer backward
        blt $t1, $t2, loop_compare  # Continue comparing until middle

    # If the loop completes, it's a palindrome
    li $v0, 1             # Return 1 for palindrome
    jr $ra

not_palindrome:
```

```
    li $v0, 0             # Return 0 for not a palindrome
    jr $ra
```

- **`loop_find_end:`**: This is the start of the loop that finds the end of the string by scanning for the null terminator (`\0`).
- **`lb $t3, 0($t2)`**: Loads the byte from the memory address pointed to by `$t2` into `$t3`. This byte is the current character in the string.
- **`beq $t3, $zero, done_find_end`**: Checks if the byte in `$t3` is the null terminator (`$zero` is always 0). If it is, the loop jumps to the label `done_find_end`, signaling the end of the string.
- **`addi $t2, $t2, 1`**: Increments `$t2` by 1 to move to the next character in the string.
- **`j loop_find_end`**: Jumps back to `loop_find_end` to continue checking the next character.
- **`done_find_end:`**: This label is reached when the null terminator is found.
- **`subi $t2, $t2, 1`**: Decrements `$t2` by 1 to point to the last actual character in the string (before the null terminator).
- **`loop_compare:`**: This is the start of the loop that compares characters from the start and end of the string.
- **`lb $t3, 0($t1)`**: Loads the byte (character) at the memory address pointed to by `$t1` (start of the string) into register `$t3`.
- **`lb $t4, 0($t2)`**: Loads the byte (character) at the memory address pointed to by `$t2` (end of the string) into register `$t4`.
- **`bne $t3, $t4, not_palindrome`**: Compares the characters in `$t3` and `$t4`. If they are not equal (`bne`: branch if not equal), the function jumps to the `not_palindrome` label, signaling that the string is not a palindrome.
- **`addi $t1, $t1, 1`**: Increments `$t1` by 1 to move to the next character from the start of the string.
- **`subi $t2, $t2, 1`**: Decrements `$t2` by 1 to move to the previous character from the end of the string.
- **`blt $t1, $t2, loop_compare`**: Compares the pointers `$t1` and `$t2`. If `$t1` is still less than `$t2` (i.e., the pointers haven't crossed), the loop continues by jumping back to `loop_compare`.
- **`li $v0, 1`**: If the loop completes without finding a mismatch, the string is a palindrome. The function loads `1` into `$v0` to indicate this.
- **`jr $ra`**: Returns to the calling function by jumping to the address stored in `$ra`.
- **`not_palindrome:`**: This label is reached if a mismatch is found during comparison.
- **`li $v0, 0`**: Loads `0` into `$v0` to indicate that the string is not a palindrome.
- **`jr $ra`**: Returns to the calling function.
- Removing the `newline` function is the same as above code.

4. Input / Output: Reverse

```
reverse_string:
    la $t0, input_str_rev   # Start of input string
    move $t1, $t0

    # Find end of string
    loop_find_end_rev:
        lb $t2, 0($t1)
        beq $t2, $zero, done_find_end_rev  # Exit when null terminator is
found
        addi $t1, $t1, 1
        j loop_find_end_rev
    done_find_end_rev:
        subi $t1, $t1, 1
```

- **`la $t0, input_str_rev`**: Loads the address of the input string `input_str_rev` into register `$t0`. `$t0` is used as a pointer to the start of the string.
- **`move $t1, $t0`**: Copies the value of `$t0` into `$t1`. `$t1` will be used to iterate through the string and find its end.
- **`loop_find_end_rev:`**: Marks the start of the loop that finds the end of the string.
- **`lb $t2, 0($t1)`**: Loads the byte (character) at the memory address pointed to by `$t1` into `$t2`.
- **`beq $t2, $zero, done_find_end_rev`**: Compares the value in `$t2` to `0` (null terminator). If it matches, the loop jumps to `done_find_end_rev`, indicating the end of the string.
- **`addi $t1, $t1, 1`**: Increments `$t1` to move to the next character in the string.
- **`j loop_find_end_rev`**: Jumps back to `loop_find_end_rev` to continue checking the next character.
- **`done_find_end_rev:`**: This label is reached once the null terminator is found.
- **`subi $t1, $t1, 1`**: Decrements `$t1` to point to the last valid character before the null terminator.

```
    la $t3, result_rev     # Store reversed string in result_rev
    loop_reverse:
    lb $t2, 0($t1)      # Load character from input
    sb $t2, 0($t3)      # Store character in result
    addi $t3, $t3, 1    # Move to next position in result
    subi $t1, $t1, 1    # Move to previous character in input
    bgt $t1, $t0, loop_reverse  # Continue until start pointer <= end
```

- **`la $t3, result_rev`**: Loads the address of the string `result_rev`, where the reversed string will be stored, into `$t3`.
- **`loop_reverse:`**: Marks the start of the loop that reverses the string.
- **`lb $t2, 0($t1)`**: Loads the character from the memory address pointed to by `$t1` (which is the current character at the end of the input string).
- **`sb $t2, 0($t3)`**: Stores the character in `$t2` (from the input string) into the memory address pointed to by `$t3` (the result string).
- **`addi $t3, $t3, 1`**: Increments `$t3` to move to the next position in the result string where the next character will be stored.
- **`subi $t1, $t1, 1`**: Decrements `$t1` to move to the previous character in the input string.
- **`bgt $t1, $t0, loop_reverse`**: Compares `$t1` (current input pointer) to `$t0` (start of input string). If `$t1` is still greater than `$t0`, the loop continues, reversing more characters. When `$t1` becomes less than or equal to `$t0`, the loop ends.

5. Input / Output: Size of String

(a) Code Explanation

```
# Initialize the counter to 0
   la $t0, input_str_size        # Load input string
   li $t1, 0                     # String length counter

   # Loop to calculate the string length
   loop_size:
       lb $t2, 0($t0)
       beq $t2, $zero, done_size  # Stop at null terminator
       beq $t2, 0x0A, done_size   # Stop at newline (ASCII 0x0A)
       addi $t1, $t1, 1           # Increment length
       addi $t0, $t0, 1           # Move to next character
       j loop_size

   done_size:
   # Print the size prompt
```

```
li $v0, 4
la $a0, prompt_size
syscall

# Print the size of the string
li $v0, 1
move $a0, $t1
syscall
```

- **`la $t0, input_str_size`**: Loads the address of the input string `input_str_size` into register `$t0`. This register will be used to point to each character of the string during the length calculation.
- **`li $t1, 0`**: Initializes `$t1` to `0`. This register will serve as the counter for the string length.
- **`loop_size:`**: Label that marks the start of the loop which calculates the string length.
- **`lb $t2, 0($t0)`**: Loads the byte (character) from the address pointed to by `$t0` (current position in the string) into `$t2`.
- **`beq $t2, $zero, done_size`**: Compares the character in `$t2` to `0` (the null terminator). If it's equal, the loop ends by jumping to `done_size`.
- **`beq $t2, 0x0A, done_size`**: Checks if the character in `$t2` is the newline character (ASCII value `0x0A`). If it's a newline, the loop also ends by jumping to `done_size`.
- **`addi $t1, $t1, 1`**: Increments the string length counter `$t1` by 1.
- **`addi $t0, $t0, 1`**: Moves the string pointer `$t0` to the next character by incrementing it by 1.
- **`j loop_size`**: Jumps back to the start of the loop (`loop_size`) to continue checking the next character.
- **`done_size:`**: This label marks the end of the loop, where the length calculation is complete.
- **`li $v0, 4`**: Loads `4` into `$v0`, which is the system call code for printing a string.
- **`la $a0, prompt_size`**: Loads the address of the string `prompt_size` (a message to prompt the user or indicate the size) into `$a0`.
- **`syscall`**: Executes the system call to print the string pointed to by `$a0`.
- **`li $v0, 1`**: Loads `1` into `$v0`, which is the system call code for printing an integer.
- **`move $a0, $t1`**: Moves the value in `$t1` (the string length) into `$a0`, which is the argument register for the print system call.
- **`syscall`**: Executes the system call to print the value in `$a0` (the string length).

6. Challenges

In Palindrome

```
Enter a string: radar
NP
-- program is finished running --
Enter a string: hello
NP
-- program is finished running --
```

- The issue is likely due to the newline character that gets included when the user presses `"Enter"` after typing the input. The MIPS syscall 8 for reading strings doesn't automatically remove the `newline (\n)`, which is added at the end of the string. This extra character causes the palindrome check to fail because `"radar\n"` is not the same as `"radar"`.
- To fix this, we need to remove the newline character after the user input is captured. Here's the updated version of the code, which removes the newline before performing the palindrome check.

In Reverse

```
Enter a string: dhyey
yeey
-- program is finished running --
```

- It seems that the issue arises because the loop condition isn't properly handling the case where the string is very short (or even empty).
- `Explanation` -It seems the issue arises because the reversed string is being written into the same space where the input string was initially stored, causing partial overwriting. To fix this, we need to store the reversed string in a separate buffer.

```
Enter a string: dhyey
yeyh
-- program is finished running --
```

- `Explanation` -The issue seems to be with the loop's exit condition. The current logic is stopping one iteration too early, which causes the first character of the original string to be skipped.
- To fix this, we should allow the loop to continue until we process the first character of the input string as well.

In Size

```
Enter a string: radar
6
-- program is finished running --
```

- The issue arises because when the user enters a string, the newline character `(\n)` is included in the input, and that extra character is being counted. so the count was 1 more than the actual length.
- Just removed the `newline(\n)` character while counting the length of the string

7. Output

Concatenate

```
Enter the first string: dhyey
Enter the second string: findoriya
dhyeyfindoriya

-- program is finished running --
```

.

Palindrome

```
Enter a string: level
P
-- program is finished running --
Enter a string: levelo
NP
-- program is finished running --
```

Reverse

```
Enter a string: yterd hdgdsb
bsdgdh drety

-- program is finished running --
```

Size

```
Enter a string: cricket
Size of the string: 7
-- program is finished running --
```

8. Real World Scenarios.

- MIPS programming provides insights into how processors handle basic operations such as memory management, arithmetic calculations, and control flow.
- Understanding assembly language like MIPS allows for optimization of critical systems where performance is key, such as in `embedded systems`, `device drivers`, or `specialized hardware.`
- **Firmware development**: Where low-level control over hardware is necessary.
- **Performance optimization**: Efficient use of CPU resources through understanding of instruction execution.
- **Security**: In fields like reverse engineering and debugging where understanding the underlying code is crucial.
