**Lab Title:**

Bubble Sort Implementation and Optimization in MIPS Assembly (MARS)

**Student Name:**

*Sameer Ahmed (27818)*

*Muneeb Mughal (27855)*

**Course:**

Computer Organization and Assembly Language (COAL)

**Instructor:**

*Nazia Junaid*

**Date:**

*6/13/2025*

**Objective:**

The primary objective of this lab is to:

* Implement the **Bubble Sort algorithm** in MIPS assembly language using the **MARS simulator**.
* Optimize the implementation to **stop early** if the array becomes sorted before completing all passes.

**Theory:**

**Bubble Sort Overview:**

Bubble Sort is a simple comparison-based sorting algorithm. It repeatedly steps through the list, compares adjacent elements, and swaps them if they are in the wrong order. This process continues until no swaps are needed — indicating that the list is sorted.

**Optimization:**

To improve efficiency, a **"swapped" flag** is introduced. If no swaps are made during an iteration, the array is already sorted, and the loop can exit early — reducing time complexity in best-case scenarios.

**Tools and Environment:**

* **Assembler/Simulator:** MARS (MIPS Assembler and Runtime Simulator)
* **Language:** MIPS Assembly Language
* **System:** Windows OS (any version)

**Code Explanation:**

* The array is declared in the .data section and its size is stored as a separate word.
* The main function performs:
  + Outer loop: Iterates through the array for each pass.
  + Inner loop: Compares adjacent elements and swaps if necessary.
  + An optimization checks whether any swaps occurred; if not, sorting is complete.
* After sorting, the program prints the sorted array line by line.

**Optimized Code Highlights:**

* **Early Exit Check:**
* li $t9, 0 # swapped = 0
* ...
* li $t9, 1 # swapped = 1 (if a swap occurs)
* ...
* beqz $t9, done # Exit if no swaps

**Test Case:**

**Input:**

array: .word 9, 5, 1, 3, 8, 2, 7, 4, 6

**Output (Console):**

1 2 3 4 5 6 7 8 9

**Results and Discussion:**

* The program correctly sorts the array in **ascending order**.
* It prevents out-of-bounds access by carefully checking loop limits.
* The **early-exit optimization** significantly improves performance on partially or already sorted arrays.
* Debugging memory-related exceptions helped reinforce understanding of **pointer arithmetic** in MIPS.

**Conclusion:**

This lab successfully demonstrated how to implement and optimize a classic sorting algorithm in MIPS. It enhanced our understanding of **low-level programming**, **loop control**, **conditional branching**, and **memory management** in assembly.

**Appendix: Full MIPS Code**

.data

array: .word 9, 5, 1, 3, 8, 2, 7, 4, 6 # Array to sort

array\_size: .word 9

newline: .asciiz " "

.text

.globl main

main:

la $t0, array # $t0 = base address of array

lw $t1, array\_size # $t1 = array size

addi $t2, $zero, 0 # i = 0 (outer loop counter)

outer\_loop:

beq $t2, $t1, done # if i == array\_size, done

li $t9, 0 # swapped = 0

addi $t3, $zero, 0 # j = 0

sub $t4, $t1, $t2 # limit = array\_size - i

addi $t4, $t4, -1 # inner loop limit = array\_size - i - 1

inner\_loop:

bge $t3, $t4, check\_swapped # j >= limit -> exit inner loop

# array[j] = $t5, array[j+1] = $t6

sll $t7, $t3, 2 # offset = j \* 4

add $t8, $t0, $t7 # $t8 = &array[j]

lw $t5, 0($t8) # $t5 = array[j]

lw $t6, 4($t8) # $t6 = array[j+1]

ble $t5, $t6, skip\_swap

# swap array[j] and array[j+1]

sw $t6, 0($t8)

sw $t5, 4($t8)

li $t9, 1 # swapped = true

skip\_swap:

addi $t3, $t3, 1

j inner\_loop

check\_swapped:

beqz $t9, done # if swapped == 0, array is sorted

addi $t2, $t2, 1 # i++

j outer\_loop

done:

# Print sorted array

li $t2, 0 # i = 0

print\_loop:

beq $t2, $t1, exit

sll $t7, $t2, 2

add $t8, $t0, $t7

lw $a0, 0($t8)

li $v0, 1

syscall

li $v0, 4

la $a0, newline

syscall

addi $t2, $t2, 1

j print\_loop

exit:

li $v0, 10

syscall