EE161 Lab 1 Report
Hands-on with the MIPS ISA using MARS

Kaushik Vada (vvada002)

October 18, 2025

1 Part 1 – Registers and Arithmetic

Objectives

Swap two register values without a temporary register, then compute arithmetic operations on the swapped operands while observing the resulting register file state.

Implementation Details

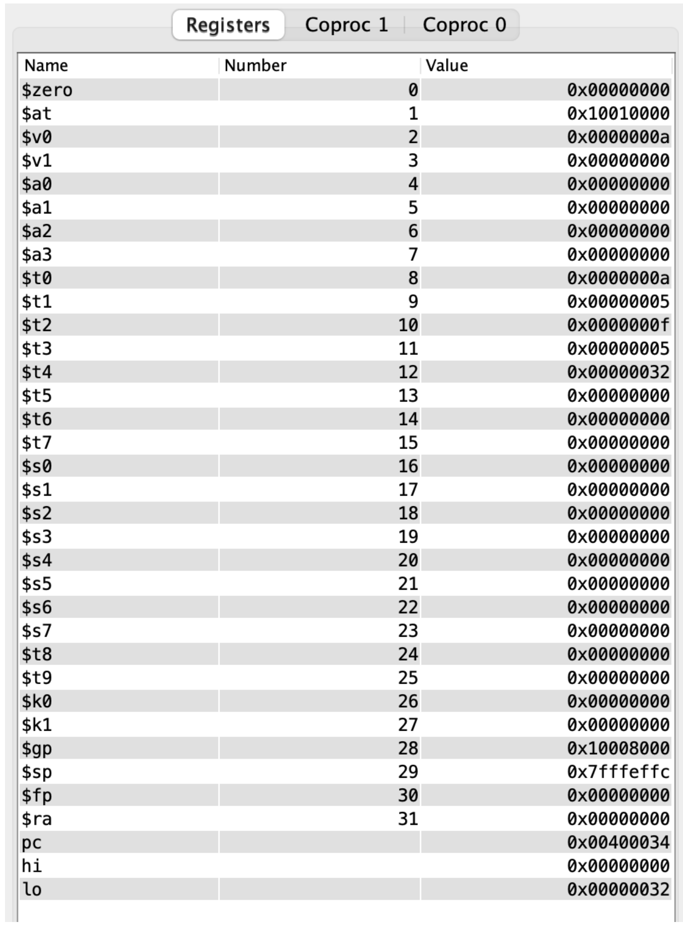
I used an XOR-swapping sequence to exchange the values in `$t0` and `$t1`, guaranteeing no additional registers were required. After the swap, the program performs addition, subtraction, and multiplication, storing the results in `$t2`, `$t3`, and `$t4`. The multiplication result is retrieved from `LO` via `mflo`.

```
# Lab 1: Registers and Arithmetic Operations

.data
    num1: .word 5
    num2: .word 10

.text
    lw $t0, num1
    lw $t1, num2

    xor  $t0, $t0, $t1
    xor  $t1, $t0, $t1
    xor  $t0, $t0, $t1

    addu $t2, $t0, $t1
    subu $t3, $t0, $t1

    mult $t0, $t1
    mflo $t4

    li $v0, 10
    syscall
```

Listing 1: Part 1 solution (`part1_Solution.asm`)

Results

Swapping the loaded constants of 5 and 10 results in `$t0 = 10` and `$t1 = 5`. The computed outputs are `$t2 = 15`, `$t3 = 5`, and `$t4 = 50`, matching the expected arithmetic outcomes. The captured register window in Figure 1 confirms these values inside MARS.



Name	Number	Value
$zero	0	0x00000000
$at	1	0x10010000
$v0	2	0x0000000a
$v1	3	0x00000000
$a0	4	0x00000000
$a1	5	0x00000000
$a2	6	0x00000000
$a3	7	0x00000000
$t0	8	0x0000000a
$t1	9	0x00000005
$t2	10	0x0000000f
$t3	11	0x00000005
$t4	12	0x00000032
$t5	13	0x00000000
$t6	14	0x00000000
$t7	15	0x00000000
$s0	16	0x00000000
$s1	17	0x00000000
$s2	18	0x00000000
$s3	19	0x00000000
$s4	20	0x00000000
$s5	21	0x00000000
$s6	22	0x00000000
$s7	23	0x00000000
$t8	24	0x00000000
$t9	25	0x00000000
$k0	26	0x00000000
$k1	27	0x00000000
$gp	28	0x10008000
$sp	29	0x7fffeffc
$fp	30	0x00000000
$ra	31	0x00000000
pc		0x00400034
hi		0x00000000
lo		0x00000032

Figure 1: Register state after executing Part 1 program.

2 Part 2 – Debugging a Loop and Working with Arrays

Objectives

Repair the provided loop so it correctly iterates across an array, accumulate the original sum, double each element in place, and track the maximum doubled value.

Bug Analysis and Fix

The starter loop failed to advance the base address by a full word and attempted to load using a malformed pseudo-instruction, effectively rereading the same element and leaving the pointer misaligned. Replacing the invalid operation with a proper `lw` from `0($t0)` and incrementing the pointer using `addi $t0, $t0, 4` restored correct traversal. This change ensures each iteration reads the intended element, updates it, and proceeds to the next address boundary.

Implementation Details

The corrected program sums the original array into `$t2`, doubles each element using a left shift, and tracks the maximum doubled result with a compare-and-select pattern. After the loop, it writes the computed sum and maximum to memory so they can be inspected from the data segment.

```
# Lab 2 Starter Code: Debugging a Loop

.data
    arr:      .word 1, 4, 7, 2, 5
    n:        .word 5
    result:   .word 0
    maxval:   .word 0

.text
.globl main
main:
    la   $t0, arr
    lw   $t1, n
    li   $t2, 0
    li   $t5, 0x80000000

loop:
    beq  $t1, $zero, done
    lw   $t3, 0($t0)

    addu $t2, $t2, $t3

    sll  $t4, $t3, 1
    sw   $t4, 0($t0)

    slt  $t6, $t5, $t4
    beq  $t6, $zero, skip_max
    move $t5, $t4
skip_max:
    addi $t0, $t0, 4
    addi $t1, $t1, -1
    j    loop
```

```
done:
    sw   $t2, result
    sw   $t5, maxval

    li   $v0, 10
    syscall
```

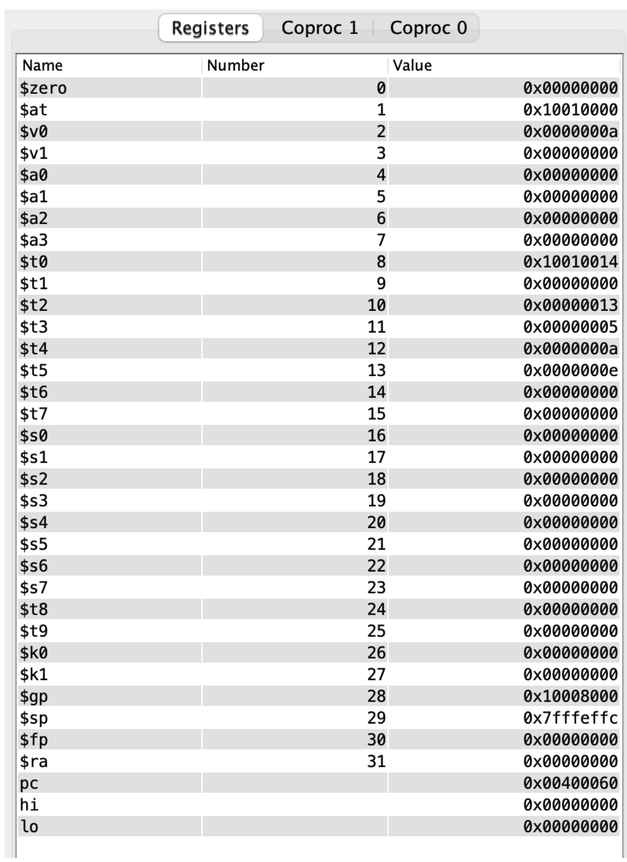
Listing 2: Part 2 solution (`part2_Solution.asm`)

Results

The original sum of the array is 19, while the doubled array becomes {2, 8, 14, 4, 10}. The maximum doubled value is 14. These values agree with the expectations and the data segment snapshot in Figure 2. A quick summary is provided in Table 1.

Table 1: Computed metrics for Part 2.

Quantity	Value
Sum of original elements	19
Doubled array	$\{2, 8, 14, 4, 10\}$
Maximum doubled value	14



Name	Number	Value
$zero	0	0x00000000
$at	1	0x10010000
$v0	2	0x0000000a
$v1	3	0x00000000
$a0	4	0x00000000
$a1	5	0x00000000
$a2	6	0x00000000
$a3	7	0x00000000
$t0	8	0x10010014
$t1	9	0x00000000
$t2	10	0x00000013
$t3	11	0x00000005
$t4	12	0x0000000a
$t5	13	0x0000000e
$t6	14	0x00000000
$t7	15	0x00000000
$s0	16	0x00000000
$s1	17	0x00000000
$s2	18	0x00000000
$s3	19	0x00000000
$s4	20	0x00000000
$s5	21	0x00000000
$s6	22	0x00000000
$s7	23	0x00000000
$t8	24	0x00000000
$t9	25	0x00000000
$k0	26	0x00000000
$k1	27	0x00000000
$gp	28	0x10008000
$sp	29	0x7fffeffc
$fp	30	0x00000000
$ra	31	0x00000000
pc		0x00400060
hi		0x00000000
lo		0x00000000

Figure 2: MARS output showing doubled array, sum, and maximum value.

3 Part 3 – Stack and Function Call Simulation

Objectives

Finalize the provided function template so it obeys the MIPS calling convention, uses the stack to protect caller state, and demonstrates multiple calls that store returned results.

Implementation Details

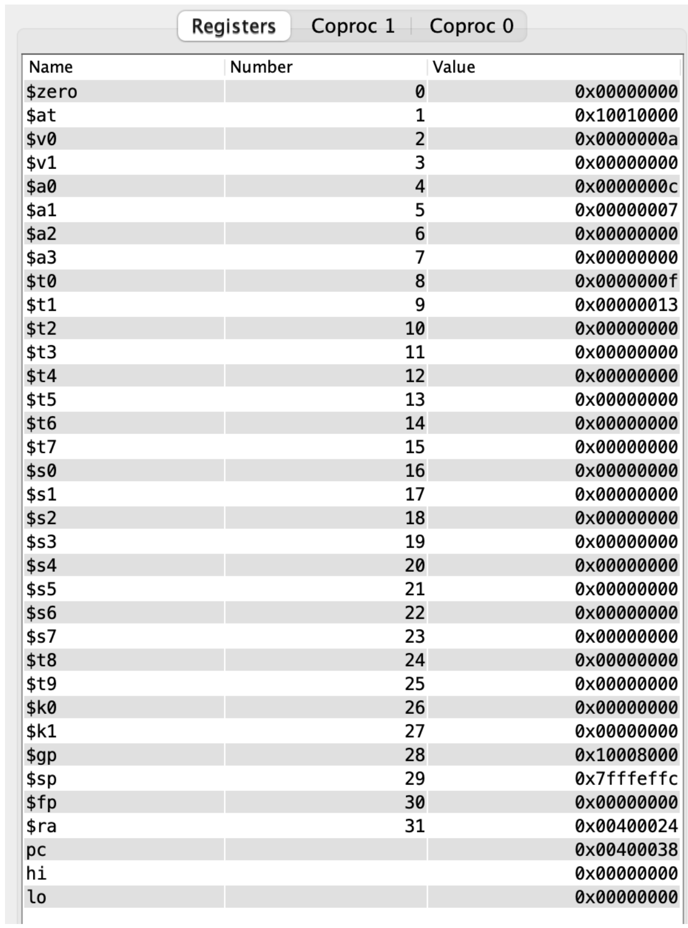
The `sum` procedure now allocates a four-byte stack frame, saves the return address (`$ra`), and restores it prior to returning. The main routine issues two calls with different argument pairs, storing each result in memory locations `result` and `result2`. This sequence validates that `$v0` is successfully returned and that the stack is balanced across calls.

```
# Lab 3 Starter Code: Stack and Function Call Simulation

.data
    result:  .word 0
    result2: .word 0

.text
main:
    li $a0, 5
    li $a1, 10
    jal sum
    move $t0, $v0
    sw $v0, result

    li $a0, 12
    li $a1, 7
    jal sum
    move $t1, $v0
    sw $v0, result2

    li $v0, 10
    syscall

sum:
    addiu $sp, $sp, -4
    sw $ra, 0($sp)

    addu $v0, $a0, $a1

    lw $ra, 0($sp)
    addiu $sp, $sp, 4
    jr $ra
```

Listing 3: Part 3 solution (`part3_Solution.asm`)

Results

The function returns 15 for the first call and 19 for the second, corresponding to the provided argument pairs. Figure 3 captures the register and memory view that verifies both stored results. The explicit push/pop of `$ra` ensures the return address survives nested calls, illustrating the core idea behind stack frames: reserving space for saved registers and local data while maintaining balanced stack pointer adjustments.



Name	Number	Value
$zero	0	0x00000000
$at	1	0x10010000
$v0	2	0x0000000a
$v1	3	0x00000000
$a0	4	0x0000000c
$a1	5	0x00000007
$a2	6	0x00000000
$a3	7	0x00000000
$t0	8	0x0000000f
$t1	9	0x00000013
$t2	10	0x00000000
$t3	11	0x00000000
$t4	12	0x00000000
$t5	13	0x00000000
$t6	14	0x00000000
$t7	15	0x00000000
$s0	16	0x00000000
$s1	17	0x00000000
$s2	18	0x00000000
$s3	19	0x00000000
$s4	20	0x00000000
$s5	21	0x00000000
$s6	22	0x00000000
$s7	23	0x00000000
$t8	24	0x00000000
$t9	25	0x00000000
$k0	26	0x00000000
$k1	27	0x00000000
$gp	28	0x10008000
$sp	29	0x7fffeffc
$fp	30	0x00000000
$ra	31	0x00400024
pc		0x00400038
hi		0x00000000
lo		0x00000000

Figure 3: Register and memory snapshot after executing both function calls.

4 Discussion and Lessons Learned

This lab reinforced several foundational ISA competencies: using bitwise operations to eliminate temporary storage, reasoning about pointer arithmetic and memory alignment, and adhering to calling conventions via explicit stack management. Validating each stage in MARS highlighted how quickly small instruction-level mistakes (such as incorrect pointer increments) propagate through a program, underscoring the value of careful register and memory inspection. The resulting assembly artifacts and screenshots provide a complete record of the working solutions.