UNIVERSITETI® METROPOLITAN TIRANA

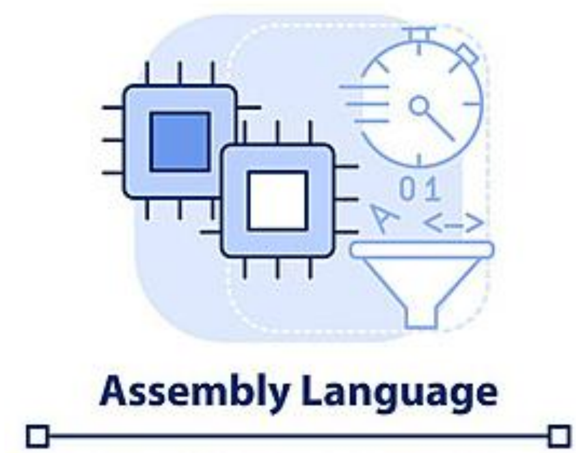
FACULTY OF COMPUTER SCIENCES AND IT

Software Engineering Program

Computer Architecture

DLX Assembly Project Documentation

Scalar Multiplication of a Vector & Caesar Cipher Encryption on a Character Vector



Worked by: Krisalda Mihali

Accepted by: Prof. Oriod Malo

1. Introduction

This project explores the application of DLX assembly language in implementing two essential low-level computational routines: scalar multiplication of a fixed-size vector and Caesar Cipher encryption on individual characters. These exercises serve as practical demonstrations of core computer architecture concepts, such as register-based data manipulation, memory addressing, instruction-level operations, and control flow in assembly. By translating abstract algorithmic logic into assembly code, the project aims to strengthen students' understanding of how simple data processing tasks are executed at the hardware-near level, while also promoting disciplined and structured programming practices.

2. Exercise 1: Scalar Multiplication of a Vector

Objective:

Multiply each element of a 5-element vector by a scalar value using the DLX assembly language, and store the results both in memory and visible CPU registers.

Data Section:

- *vector:* Holds the original array of integers {1, 2, 3, 4, 5}.
- *result:* Reserved memory space to store the output of the multiplication.
- Scalar multiplier: The constant value 3 is loaded directly into a register.
- No explicit *length* is needed since the vector size is known (5 elements).

Key Registers Used:

Register	Purpose
r1	Scalar multiplier (3)
r2	Base address of the input vector
r3	Base address of the result vector
r4–r8	Registers to hold vector elements
r4–r8	Also used to store multiplication results

Explanation:

The program performs scalar multiplication of a 5-element vector using a fixed scalar value (3). It first initializes the scalar and base memory addresses. Then, it loads all vector elements into separate CPU registers (r4 to r8), performs multiplication using the mult instruction available in DLX, and stores the resulting values back into memory at the addresses pointed to by r3.

Each product is kept in the same register that held the original value, effectively overwriting it. The results are then stored sequentially into the reserved result array in memory.

Example Result:

For the input vector: ***[1, 2, 3, 4, 5]*** and scalar multiplier: ***3***
The result stored in registers r4–r8 and memory location result will be: ***[3, 6, 9, 12, 15]***

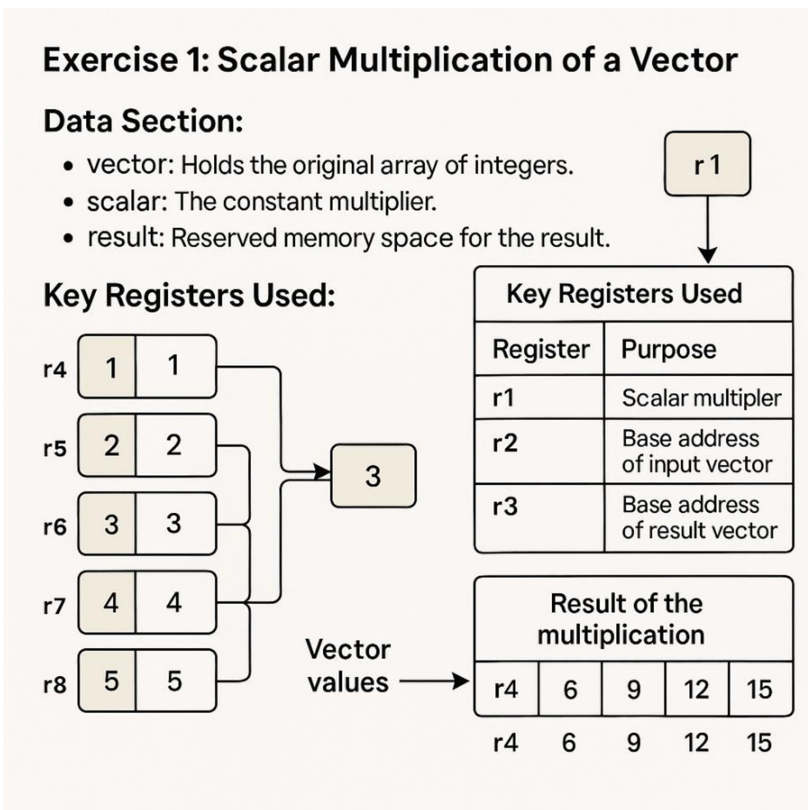


Fig. 1. Visual Diagram – Scalar Multiplication of a Vector

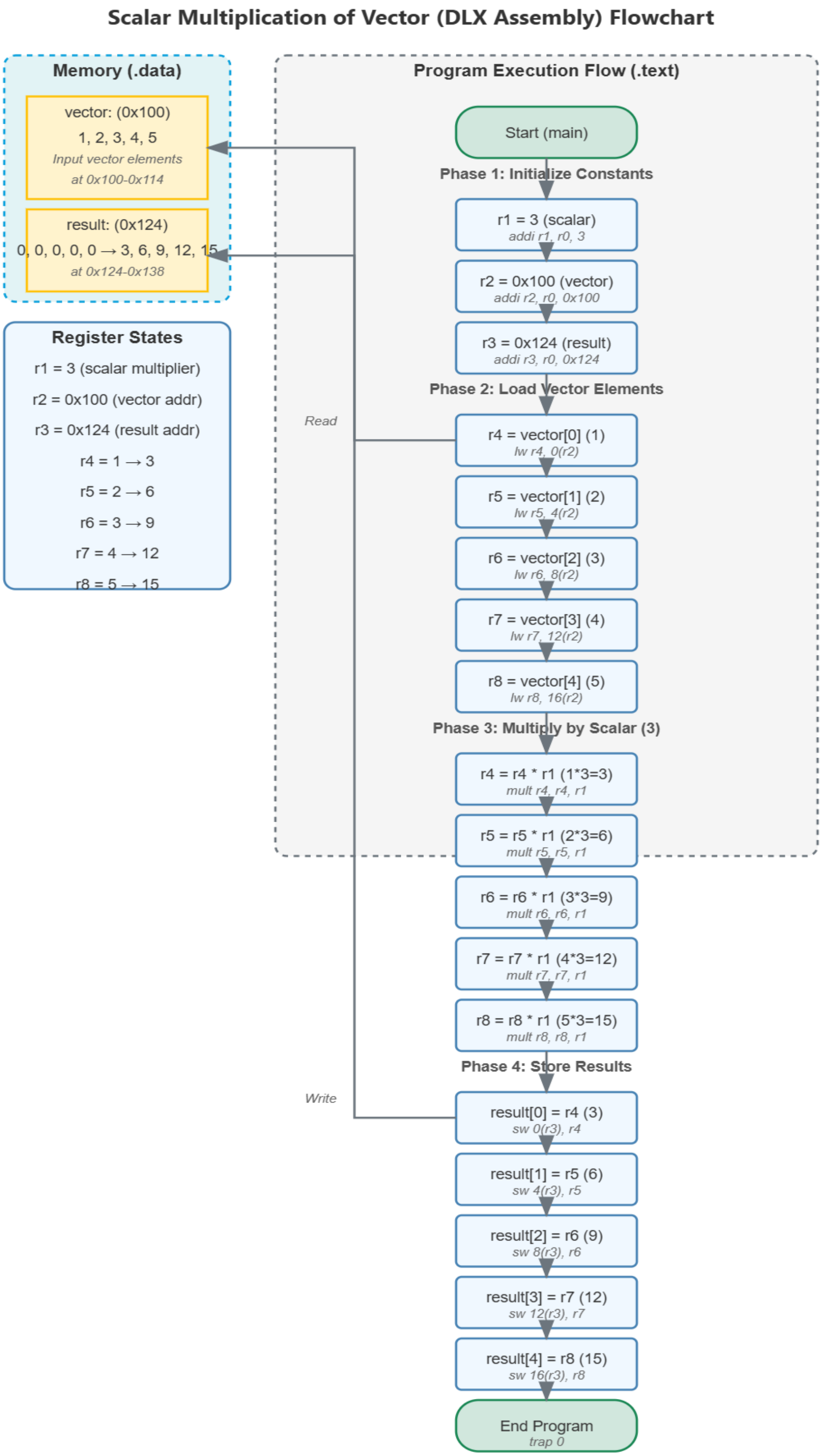
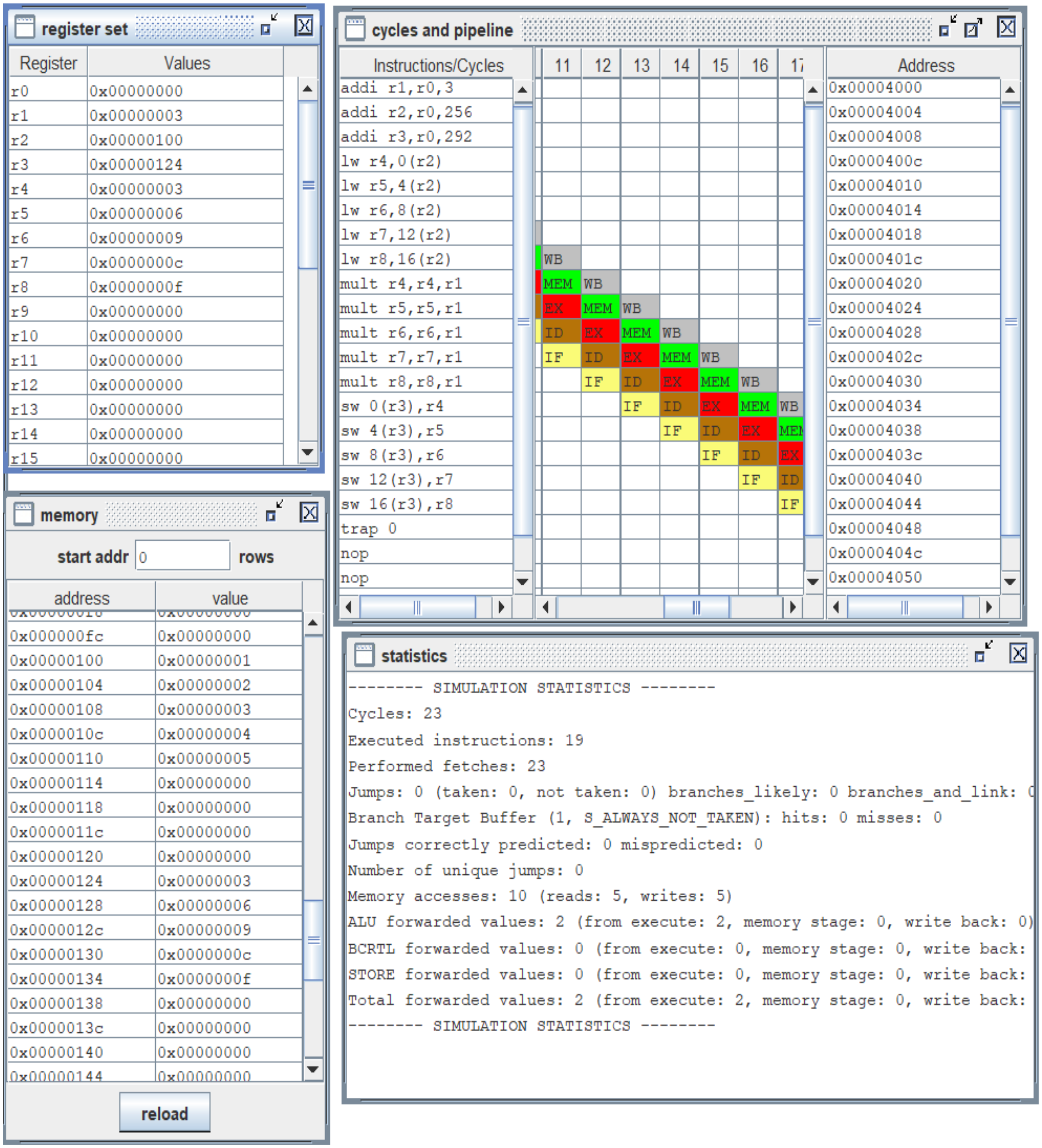


Fig.2. DLX Scalar Multiplication Flow Chart

Screenshots of the code:

coding frame

```
;------------------------------------------------------------
; Scalar Multiplication of a Vector (DLX Assembly)
; Performs scalar multiplication of a 5-element vector by 3
;------------------------------------------------------------

        .data
vector: .word 1, 2, 3, 4, 5       ; Input vector starting at 0x00000100
result: .word 0, 0, 0, 0, 0       ; Output vector starting at 0x00000124

        .text
        .global main
main:
    ;--------------------------------------
    ; Initialize constants and base addresses
    addi r1, r0, 3             ; r1 = scalar multiplier = 3
    addi r2, r0, 0x100         ; r2 = base address of input vector
    addi r3, r0, 0x124         ; r3 = base address of result vector

    ;--------------------------------------
    ; Load vector elements into registers
    lw r4, 0(r2)               ; r4 = vector[0]
    lw r5, 4(r2)               ; r5 = vector[1]
    lw r6, 8(r2)               ; r6 = vector[2]
    lw r7, 12(r2)              ; r7 = vector[3]
    lw r8, 16(r2)              ; r8 = vector[4]

    ;--------------------------------------
    ; Multiply each element by scalar
    mult r4, r4, r1            ; r4 = vector[0] * 3
    mult r5, r5, r1            ; r5 = vector[1] * 3
    mult r6, r6, r1            ; r6 = vector[2] * 3
    mult r7, r7, r1            ; r7 = vector[3] * 3
    mult r8, r8, r1            ; r8 = vector[4] * 3

    ;--------------------------------------
    ; Store results back to memory
    sw 0(r3), r4               ; result[0] = r4
    sw 4(r3), r5               ; result[1] = r5
    sw 8(r3), r6               ; result[2] = r6
    sw 12(r3), r7              ; result[3] = r7
    sw 16(r3), r8              ; result[4] = r8

    ;--------------------------------------
    ; Terminate program
    trap 0                     ; End execution
```


register set
Register | Values
r0 0x00000000
r1 0x00000003
r2 0x00000100
r3 0x00000124
r4 0x00000003
r5 0x00000006
r6 0x00000009
r7 0x0000000c
r8 0x0000000f
r9 0x00000000
r10 0x00000000
r11 0x00000000
r12 0x00000000
r13 0x00000000
r14 0x00000000
r15 0x00000000
cycles and pipeline
Instructions/Cycles
11 12 13 14 15 16 17
addi r1,r0,3
addi r2,r0,256
addi r3,r0,292
lw r4,0(r2)
lw r5,4(r2)
lw r6,8(r2)
lw r7,12(r2)
lw r8,16(r2)
mult r4,r4,r1
mult r5,r5,r1
mult r6,r6,r1
mult r7,r7,r1
mult r8,r8,r1
sw 0(r3),r4
sw 4(r3),r5
sw 8(r3),r6
sw 12(r3),r7
sw 16(r3),r8
trap 0
nop
nop
WB
MEM WB
EX MEM WB
ID EX MEM WB
IF ID EX MEM WB
IF ID EX MEM WB
IF ID EX MEM WB
IF ID EX MEM
IF ID EX
IF ID
IF
Address
0x00004000
0x00004004
0x00004008
0x0000400c
0x00004010
0x00004014
0x00004018
0x0000401c
0x00004020
0x00004024
0x00004028
0x0000402c
0x00004030
0x00004034
0x00004038
0x0000403c
0x00004040
0x00004044
0x00004048
0x0000404c
0x00004050
memory
start addr 0 rows
address | value
0x000000fc 0x00000000
0x00000100 0x00000001
0x00000104 0x00000002
0x00000108 0x00000003
0x0000010c 0x00000004
0x00000110 0x00000005
0x00000114 0x00000000
0x00000118 0x00000000
0x0000011c 0x00000000
0x00000120 0x00000000
0x00000124 0x00000003
0x00000128 0x00000006
0x0000012c 0x00000009
0x00000130 0x0000000c
0x00000134 0x0000000f
0x00000138 0x00000000
0x0000013c 0x00000000
0x00000140 0x00000000
0x00000144 0x00000000
reload
statistics
-------- SIMULATION STATISTICS --------
Cycles: 23
Executed instructions: 19
Performed fetches: 23
Jumps: 0 (taken: 0, not taken: 0) branches_likely: 0 branches_and_link:
Branch Target Buffer (1, S_ALWAYS_NOT_TAKEN): hits: 0 misses: 0
Jumps correctly predicted: 0 mispredicted: 0
Number of unique jumps: 0
Memory accesses: 10 (reads: 5, writes: 5)
ALU forwarded values: 2 (from execute: 2, memory stage: 0, write back: 0)
BCRTL forwarded values: 0 (from execute: 0, memory stage: 0, write back:
STORE forwarded values: 0 (from execute: 0, memory stage: 0, write back:
Total forwarded values: 2 (from execute: 2, memory stage: 0, write back:
-------- SIMULATION STATISTICS --------

3. Exercise 2: Caesar Cipher Encryption on a Character Vector

Objective:

Encrypt individual characters using the Caesar Cipher algorithm with a specified shift value, demonstrating character manipulation in assembly language.

Data Section:

- *char_K* and *char_M*: ASCII values of characters 'K' and 'M'
- *shift*: Caesar cipher shift amount (3)
- *encrypted_K, encrypted_M*: Memory locations reserved for the encrypted outputs

Key Registers Used:

Register	Purpose
r1	Input character ('K')
r2	Input character ('M')
r3	Encrypted result for 'K'
r4	Encrypted result for 'M'

Subroutine: encrypt (conceptual)

The Caesar cipher function checks if the input character is an uppercase letter (ASCII 65–90). If so, it applies the shift with a manual modulo 26 operation to wrap around the alphabet. Otherwise, the input character is returned unchanged.

Example Execution:

- 'K' (ASCII 75) becomes 'N' (ASCII 78)
- 'M' (ASCII 77) becomes 'P' (ASCII 80)

Register Result Storage:

- r4 stores 'N' (78)
- r5 stores 'P' (80)

Table of Values:

Character	ASCII Before	ASCII After	Encrypted Char
K	75	78	N
M	77	80	P

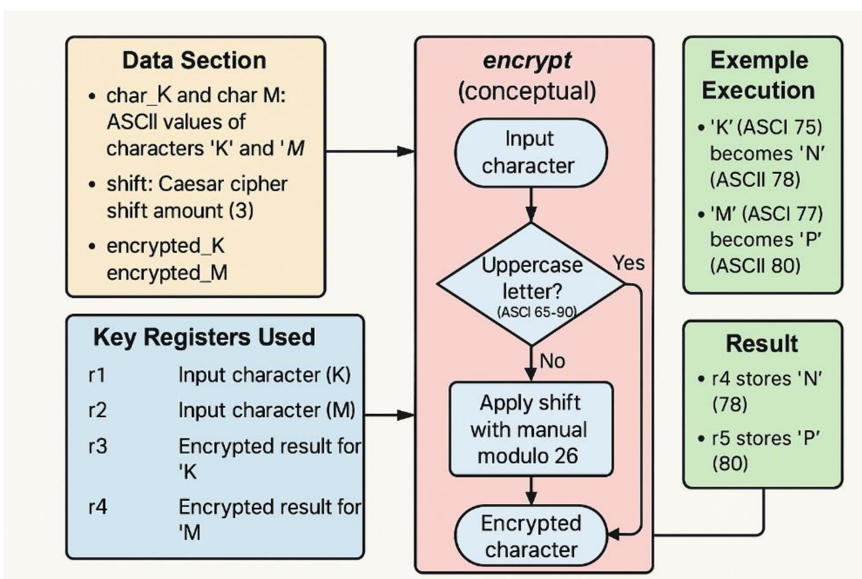


Fig.3. Visual Diagram – Caesar Cipher Encryption on a Character Vector

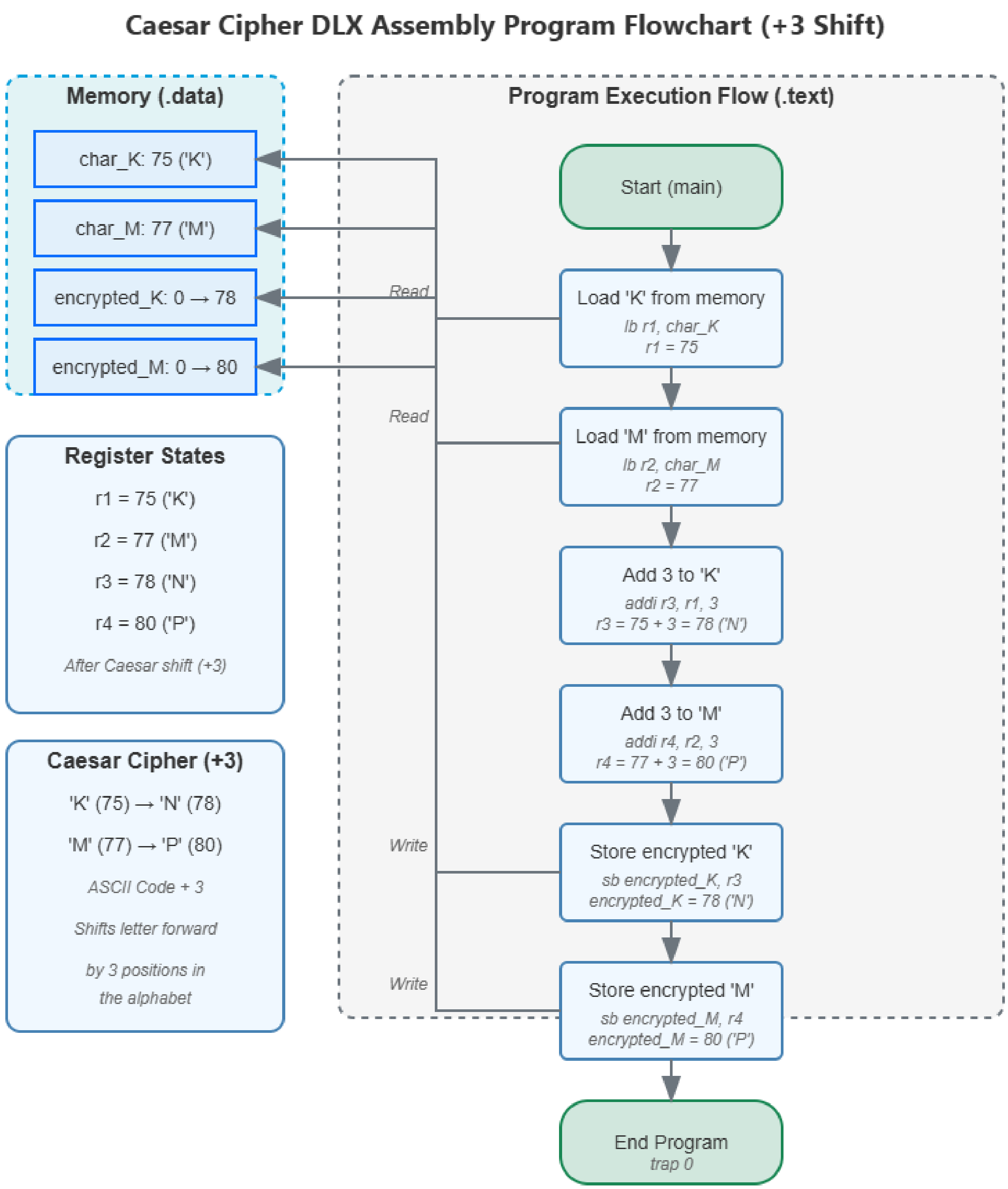
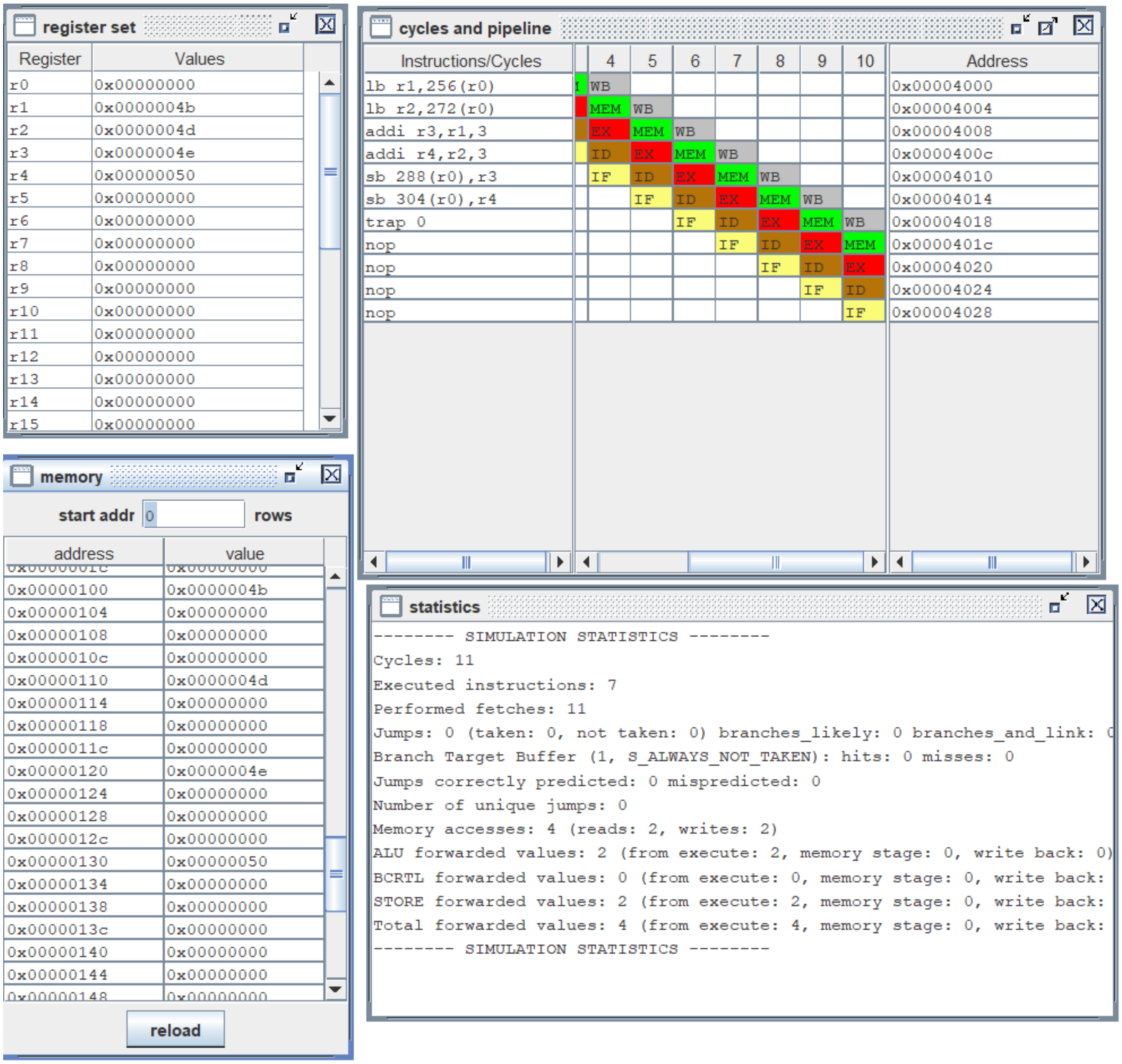


Fig.4. DLX Caesar Cipher Encryption on a Character Vector Flow Chart

Screenshots of the code:

```
;---------------------------------------------------------
; Caesar Cipher Implementation in DLX Assembly
; Encrypts characters 'K' and 'M' using a Caesar shift of +3
;---------------------------------------------------------

;------ Data Segment ------
        .data
; Store original characters in memory (ASCII codes)
char_K:     .byte 75           ; ASCII code for 'K'
            .align 4           ; Align to next word boundary for consistency
char_M:     .byte 77           ; ASCII code for 'M'
            .align 4           ; Align to next word boundary

; Reserve memory space for encrypted results
encrypted_K: .byte 0           ; Placeholder for encrypted 'K' (should be 'N' -> 78)
             .align 4          ; Align for word alignment
encrypted_M: .byte 0           ; Placeholder for encrypted 'M' (should be 'P' -> 80)
             .align 4          ; Align for word alignment

;------ Text Segment ------
        .text
        .global main
main:
        ; Load original character values into registers
        lb      r1, char_K         ; Load 'K' (ASCII 75) into register r1
        lb      r2, char_M         ; Load 'M' (ASCII 77) into register r2

        ; Apply Caesar shift (+3 positions forward in alphabet)
        addi    r3, r1, 3          ; r3 = 'K' + 3 => 'N' (ASCII 78)
        addi    r4, r2, 3          ; r4 = 'M' + 3 => 'P' (ASCII 80)

        ; Store the encrypted results in memory
        sb      encrypted_K, r3    ; Store 'N' into encrypted_K location
        sb      encrypted_M, r4    ; Store 'P' into encrypted_M location

        ; Terminate program
        trap    0                  ; Stop execution
```



4. Conclusion

The successful implementation of both scalar vector multiplication and Caesar Cipher encryption using DLX assembly highlights the power and precision of low-level programming in managing memory and processor operations. The project effectively demonstrates how elementary algorithms can be decomposed into a sequence of register operations and memory transactions within a reduced instruction set computing (RISC) environment. Beyond the technical execution, this work has deepened the conceptual grasp of assembly programming and reinforced best practices in code optimization, clarity, and logical structuring, skills fundamental to embedded systems and systems-level development.