CSCI 592

LAB ASSIGNMENT – 7

Written by

DINESH SEVETI

Date: 03-29-2025

**OBJECTIVE**

The objective of this project is to implement a simple convolution operation on a 3x3 image matrix using a 2x2 kernel in both Easy68k and RISC-V assembly languages, without utilizing loops. Each element of the result matrix is computed by directly performing the required operations (multiplication and addition) on the image and kernel matrix elements.

**TECHNOLOGY USED**

* Easy68k Assembly Language: Using the Easy68k simulator to execute the assembly code.
* RISC-V Assembly Language: The program is written for the RISC-V architecture, focusing on optimizing convolution calculations through register-based operations.

**PROCEDURE**

* Data Representation: Define a 3x3 image matrix and a 2x2 kernel matrix in the data section.
* Multiplication & Summation: Perform element-wise multiplication between the image and kernel matrices, summing the results manually for each element in the output matrix.
* Storing Results: Store the computed convolution results in the result matrix.
* Exit the Program: After all calculations, both programs exit gracefully with a system call.

**OPERATIONS**

* Matrix Multiplication: Multiply corresponding elements from the image and kernel matrices.
* Addition: Add the products to compute the result for each element of the result matrix.
* Memory Store: Store the resulting values into the result matrix.
* System Exit: Use a system call to exit the program after processing.

**ALGORITHM**

**The algorithm for calculating the convolution in both programs is as follows:**

* Initialize the result matrix to store the output values.
* For each element in the result matrix:
* Multiply the corresponding image and kernel elements.
* Sum the results of these multiplications to compute the result.
* Store the result in the result matrix.
* After processing all elements, exit the program.

**CODE LISTING**

ORG $1000 ; Start of the program

image: DC.W 1, 2, 3 ; 3x3 image row 1

DC.W 4, 5, 6 ; 3x3 image row 2

DC.W 7, 8, 9 ; 3x3 image row 3

kernel: DC.W 1, 0 ; 2x2 kernel row 1

DC.W 0, -1 ; 2x2 kernel row 2

result: DS.W 4 ; Space for 2x2 result matrix

; Calculate result[0][0]

MOVE.W image, D0 ; D0 = image[0][0] = 1

MOVE.W kernel, D1 ; D1 = kernel[0][0] = 1

MULS D1, D0 ; D0 = D0 \* D1 = 1 \* 1 = 1

MOVE.W image+2, D2 ; D2 = image[0][1] = 2

MOVE.W kernel+2, D3 ; D3 = kernel[0][1] = 0

MULS D3, D2 ; D2 = D2 \* D3 = 2 \* 0 = 0

ADD.W D2, D0 ; D0 = D0 + D2 = 1 + 0 = 1

MOVE.W image+6, D2 ; D2 = image[1][0] = 4

MOVE.W kernel+4, D3 ; D3 = kernel[1][0] = 0

MULS D3, D2 ; D2 = D2 \* D3 = 4 \* 0 = 0

ADD.W D2, D0 ; D0 = D0 + D2 = 1 + 0 = 1

MOVE.W image+8, D2 ; D2 = image[1][1] = 5

MOVE.W kernel+6, D3 ; D3 = kernel[1][1] = -1

MULS D3, D2 ; D2 = D2 \* D3 = 5 \* -1 = -5

ADD.W D2, D0 ; D0 = D0 + D2 = 1 + (-5) = -4

MOVE.W D0, result ; result[0][0] = -4

; Calculate result[0][1]

MOVE.W image+2, D0 ; D0 = image[0][1] = 2

MOVE.W kernel, D1 ; D1 = kernel[0][0] = 1

MULS D1, D0 ; D0 = D0 \* D1 = 2 \* 1 = 2

MOVE.W image+4, D2 ; D2 = image[0][2] = 3

MOVE.W kernel+2, D3 ; D3 = kernel[0][1] = 0

MULS D3, D2 ; D2 = D2 \* D3 = 3 \* 0 = 0

ADD.W D2, D0 ; D0 = D0 + D2 = 2 + 0 = 2

MOVE.W image+8, D2 ; D2 = image[1][1] = 5

MOVE.W kernel+4, D3 ; D3 = kernel[1][0] = 0

MULS D3, D2 ; D2 = D2 \* D3 = 5 \* 0 = 0

ADD.W D2, D0 ; D0 = D0 + D2 = 2 + 0 = 2

MOVE.W image+10, D2 ; D2 = image[1][2] = 6

MOVE.W kernel+6, D3 ; D3 = kernel[1][1] = -1

MULS D3, D2 ; D2 = D2 \* D3 = 6 \* -1 = -6

ADD.W D2, D0 ; D0 = D0 + D2 = 2 + (-6) = -4

MOVE.W D0, result+2 ; result[0][1] = -4

; Calculate result[1][0]

MOVE.W image+6, D0 ; D0 = image[1][0] = 4

MOVE.W kernel, D1 ; D1 = kernel[0][0] = 1

MULS D1, D0 ; D0 = D0 \* D1 = 4 \* 1 = 4

MOVE.W image+8, D2 ; D2 = image[1][1] = 5

MOVE.W kernel+2, D3 ; D3 = kernel[0][1] = 0

MULS D3, D2 ; D2 = D2 \* D3 = 5 \* 0 = 0

ADD.W D2, D0 ; D0 = D0 + D2 = 4 + 0 = 4

MOVE.W image+12, D2 ; D2 = image[2][0] = 7

MOVE.W kernel+4, D3 ; D3 = kernel[1][0] = 0

MULS D3, D2 ; D2 = D2 \* D3 = 7 \* 0 = 0

ADD.W D2, D0 ; D0 = D0 + D2 = 4 + 0 = 4

MOVE.W image+14, D2 ; D2 = image[2][1] = 8

MOVE.W kernel+6, D3 ; D3 = kernel[1][1] = -1

MULS D3, D2 ; D2 = D2 \* D3 = 8 \* -1 = -8

ADD.W D2, D0 ; D0 = D0 + D2 = 4 + (-8) = -4

MOVE.W D0, result+4 ; result[1][0] = -4

; Calculate result[1][1]

MOVE.W image+8, D0 ; D0 = image[1][1] = 5

MOVE.W kernel, D1 ; D1 = kernel[0][0] = 1

MULS D1, D0 ; D0 = D0 \* D1 = 5 \* 1 = 5

MOVE.W image+10, D2 ; D2 = image[1][2] = 6

MOVE.W kernel+2, D3 ; D3 = kernel[0][1] = 0

MULS D3, D2 ; D2 = D2 \* D3 = 6 \* 0 = 0

ADD.W D2, D0 ; D0 = D0 + D2 = 5 + 0 = 5

MOVE.W image+14, D2 ; D2 = image[2][1] = 8

MOVE.W kernel+4, D3 ; D3 = kernel[1][0] = 0

MULS D3, D2 ; D2 = D2 \* D3 = 8 \* 0 = 0

ADD.W D2, D0 ; D0 = D0 + D2 = 5 + 0 = 5

MOVE.W image+16, D2 ; D2 = image[2][2] = 9

MOVE.W kernel+6, D3 ; D3 = kernel[1][1] = -1

MULS D3, D2 ; D2 = D2 \* D3 = 9 \* -1 = -9

ADD.W D2, D0 ; D0 = D0 + D2 = 5 + (-9) = -4

MOVE.W D0, result+6 ; result[1][1] = -4

; End of program

SIMHALT ; Halt the simulator

**CODE LISTING**

.data

image: .word 1, 2, 3, 4, 5, 6, 7, 8, 9 # image 3x3 matrix

kernel: .word -1, -2, 0, 1 # kernel 2x2 matrix

result: .space 16 # space for result 2x2 matrix

.text

.globl \_start

\_start:

# Calculate result[0][0]

# Load image and kernel values into registers

la t0, image # t0 = base address of image

la t1, kernel # t1 = base address of kernel

lw t2, 0(t0)

lw t3, 0(t1)

mul t4, t2, t3

lw t2, 4(t0)

lw t3, 4(t1)

mul t5, t2, t3

add t4, t4, t5

lw t2, 12(t0)

lw t3, 8(t1)

mul t5, t2, t3

add t4, t4, t5

lw t2, 16(t0)

lw t3, 12(t1)

mul t5, t2, t3

add t4, t4, t5

la t6, result

sw t4, 0(t6) # Computes 0 and stores at result[0][0]

# Calculate result[0][1]

lw t2, 4(t0)

lw t3, 0(t1)

mul t4, t2, t3

lw t2, 8(t0)

lw t3, 4(t1)

mul t5, t2, t3

add t4, t4, t5

lw t2, 12(t0)

lw t3, 8(t1)

mul t5, t2, t3

add t4, t4, t5

lw t2, 16(t0)

lw t3, 12(t1)

mul t5, t2, t3

add t4, t4, t5

sw t4, 4(t6) # Computes -2 and stores at result[0][1]

# Calculate result[1][0]

lw t2, 12(t0)

lw t3, 0(t1)

mul t4, t2, t3

lw t2, 16(t0)

lw t3, 4(t1)

mul t5, t2, t3

add t4, t4, t5

lw t2, 24(t0)

lw t3, 8(t1)

mul t5, t2, t3

add t4, t4, t5

lw t2, 28(t0)

lw t3, 12(t1)

mul t5, t2, t3

add t4, t4, t5

sw t4, 8(t6) # Computes -6 and stores at result[1][0]

# Calculate result[1][1]

lw t2, 16(t0)

lw t3, 0(t1)

mul t4, t2, t3

lw t2, 20(t0)

lw t3, 4(t1)

mul t5, t2, t3

add t4, t4, t5

lw t2, 24(t0)

lw t3, 8(t1)

mul t5, t2, t3

add t4, t4, t5

lw t2, 32(t0)

lw t3, 12(t1)

mul t5, t2, t3

add t4, t4, t5

sw t4, 12(t6) # Computes -8 and stores at result[1][1]

# End of program

li a7, 10 # Exit system call

ecall

**DESCRIPTION**

* Easy68k: The Easy68k program calculates the convolution by manually performing element-wise multiplication and addition for each element in the result matrix. No loops are used; instead, each calculation is explicitly coded.
* RISC-V: Similarly, the RISC-V program calculates the convolution manually, without using loops. It performs individual multiplications and additions to compute each element of the result matrix.

**OBSERVATIONS**

* Easy68k: This implementation manually handles each element in the result matrix, multiplying corresponding elements from the image and kernel matrices. It does not use loops or iterations.
* RISC-V: The approach is like Easy68k, and while it uses more registers, it still avoids loops and manually processes each element of the result matrix.

**RESULTS**

**CONCLUSIONS**

In conclusion, both the Easy68k and RISC-V assembly programs effectively perform the convolution operation on a 3x3 image matrix using a 2x2 kernel matrix without utilizing loops, demonstrating a manual approach to matrix operations. By manually calculating the element-wise multiplication and addition for each position in the result matrix, the programs highlight a fundamental understanding of convolution. The results show that despite the absence of loops, the computations are accurate and provide insight into low-level programming techniques for image processing tasks. This approach not only reinforces fundamental assembly programming skills but also showcases the versatility of both Easy68k and RISC-V architectures in handling mathematical operations.