CMPE 691 – Reconfigurable System Design

PROJECT

ON

MATRIX CONVOLUTION HARDWARE ACCELERATOR

Submitted by

VEMULAPALLI SRI SAMADARSINI

JV29859

jv29859@umbc.edu

Submitted to

Dr. Ryan Robucci

FALL 2024

University of Maryland, Baltimore County

INDEX

LIST OF FIGURES

1. Introduction

Matrix convolution is a core computation generally applied in a broad range of applications including images processing, machine learning, computing vision and many more. Especially in convolution neural networks, matrix convolution plays an important role in feature extraction and pattern recognition. While convolution operations are very useful, such operations are computationally expensive, and it is a challenge to implement these operations for real-time performance on general purpose processors [1].

The challenges mentioned above can benefit from the hardware acceleration methodology as it provides ability to offload the computations. Specifically, the field-programmable gate arrays (FPGAs) supply the needed flexibility and parallelism for accomplishing convolution calculations.

This paper deals with the design and architecture of Matrix Convolution Hardware Accelerator with reference to the Finite State Machine with Datapath. The hardware module is designed to have the Avalon-MM interface to allow direct communications with a processor through memory map interconnect. As a reference point C based software-only implementation matrix convolution was also implemented and results were compared with hardware assisted convolution.

The primary objectives of this project are to prove that the hardware accelerator is functionally correct. The first objective is to prove that the achieved results outperform reference software-only implementation. To explain the benefits of the approaches based on collaboration with hardware to solve problems of computational complexity.

This paper describes the system design approach, hardware and software interface, and the quantitative characterization of the performance, with reference to the main advantages of the FPGA implementation of the matrix convolution operations.

2. Background Work

2.1 Matrix Convolution Overview

Matrix convolution is defined as a mathematical operation to yield a third matrix from two matrices signifying the localized interaction between them. The operation involves sliding a smaller matrix known as kernel over an input matrix and summing the multiplication of elements in the kernels that overlap with respective elements of an input matrix [2]. This operation is stored in an output matrix. For example, applying a 3x3 kernel to a 5x5 input matrix result in a 3x3 output matrix. The computation at each position in the output matrix is given by:

$$\begin{matrix} 0 & 1 & 2 & 3 & 4 \\ 5 & 6 & 7 & 8 & 9 \\ 10 & 11 & 12 & 13 & 14 \\ 15 & 16 & 17 & 18 & 19 \\ 20 & 21 & 22 & 23 & 24 \end{matrix} \quad * \quad \begin{matrix} 1 & 0 & 1 \\ 1 & 0 & 1 \\ 1 & 0 & 1 \end{matrix} \quad = \quad \begin{matrix} 36 & 42 & 48 \\ 66 & 72 & 78 \\ 96 & 102 & 108 \end{matrix}$$

Input Matrix Kernel Matrix Output Matrix

The computation at first position in the output matrix is shown below. The respective elements will be calculated in the same process.

$O[0][0] = (0\cdot1)+(1\cdot0)+(2\cdot1)+(5\cdot1)+(6\cdot0)+(7\cdot1)+(10\cdot1)+(11\cdot0)+(12\cdot1)$

$O[0][0] = 0 + 0 + 2 + 5 + 0 + 7 + 10 + 0 + 12 = 36$

2.2 Need for Hardware Acceleration

The main concern unique to matrix convolution is the high computational requirement of the technique. This organization makes efficient use of the processor impossible on general-purpose processors because the sequential nature of the computation means that large datasets cannot be utilized in order to reduce processing time or that applications that require real-time processing are slowed considerably by large matrices. But sometimes, due to a large number of parallel operations required and frequent data reuse optimization, hardware accelerators, especially those on FPGAs, are an efficient solution.

As for matrix convolution, there are more benefits that flow from using hardware accelerators, which include the ability to perform multiple kernel operations at a go; thereby improving of high throughput and scalability [1].

2.3 FSM-Based Hardware Design

Finite State Machines (FSMs) are a reliable basis for supervising sequential processes in hardware. In this project, an FSM based design was used to coordinate the input phase, the computational phase, and the output phase of the moving-average matrix convolution. It evolves through states like IDLE, LOAD, COMPUTE, and DONE, to check for great resource usage.

Due to this approach, the design unknowns are reduced while at the same time addressing the flexibility and scalability issues. The FSM collaboratively operates with the data path to execute the multiply-accumulate (MAC) that convolution entails to deliver reliable computations.

2.4 Relevance of Related Work

The work done by Cao et al. (2019) titled “FPGA-based Accelerator for Convolution Operations” show how FPGA-based design can be used to facilitate convolutional tasks. Their work also concentrates on getting benefits with FPGA resource to increase more throughput while minimizing the latency. But as their design targets systolic arrays [1], this project follows a much simpler FSM design intended for matrix convolution applications.

This distinction makes the hardware accelerator applicable to a wider perspective of other computations other than neural networks. In addition, the design integrates with an Avalon-MM interface to provide clear visibility by displaying it if it needs to communicate with software applications, adding to its flexibility.

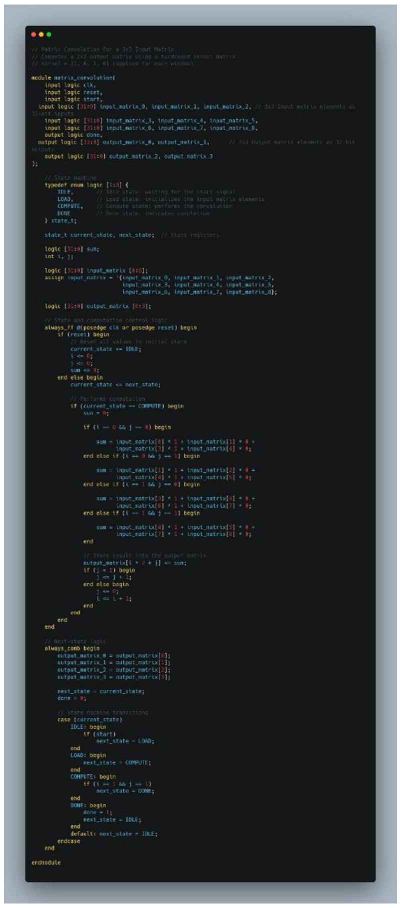
3. Design and Implementation

3.1 FSM Design for Matrix Convolution

The flow of the operations at the hardware accelerator was controlled based on the Finite State Machine (FSM) approach. The FSM consists of the following states:

- IDLE: Wait for a write or read signal to start operations.
- LOAD: Reads the input matrix elements from the processor and saves them into internal memory of the system.
- COMPUTE: Computes the output matrix by iteratively passing through the input matrix and the kernel to do the matrix convolution.
- DONE: Indicates the end of convolution process and goes back to the IDLE state.

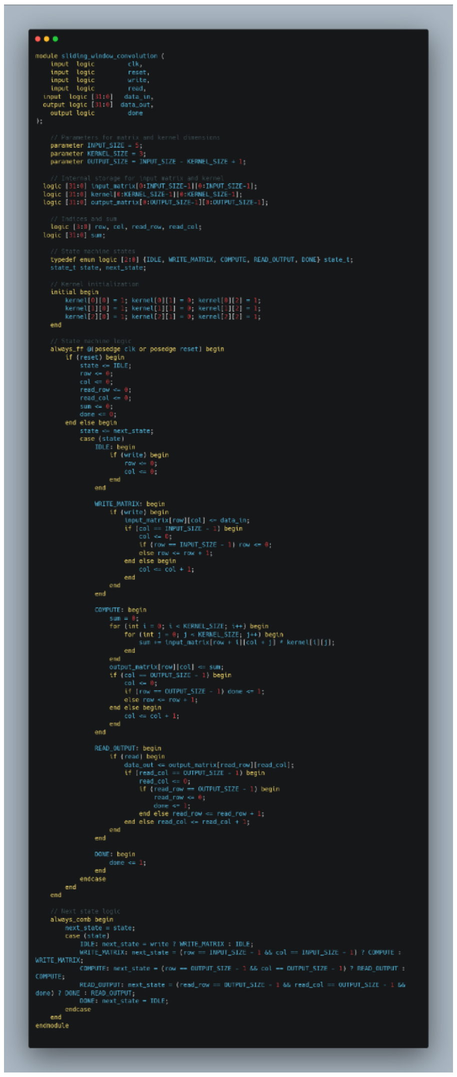
This FSM based design further ensures that the different blocks of the hardware work in harmony with minimal clock cycles being utilized. Every state changes according to control signals like start and done. Fig 1 depicts the state machine used for matrix convolution with a 3x3 input matrix and a 2x2 kernel matrix. It highlights the four main states IDLE, LOAD, COMPUTE, and DONE that coordinate the operations efficiently. This file is attached as "matrix_convolution3_3.sv".



```
// Matrix Convolution for a 3x3 Input Matrix
// Computes a 2x2 output matrix using a hardcoded kernel matrix
// Kernel = [1, 0; 1, 0] (applied for each window)

module matrix_convolution(
    input logic clk,
    input logic reset,
    input logic start,
  input logic [31:0] input_matrix_0, input_matrix_1, input_matrix_2, // 3x3 Input matrix elements as
32-bit inputs
    input logic [31:0] input_matrix_3, input_matrix_4, input_matrix_5,
    input logic [31:0] input_matrix_6, input_matrix_7, input_matrix_8,
    output logic done,
  output logic [31:0] output_matrix_0, output_matrix_1,      // 2x2 Output matrix elements as 32-bit
outputs
    output logic [31:0] output_matrix_2, output_matrix_3
);

    // State machine
    typedef enum logic [1:0] {
        IDLE,       // Idle state: waiting for the start signal
        LOAD,       // Load state: initializes the input matrix elements
        COMPUTE,    // Compute state: performs the convolution
        DONE        // Done state: indicates completion
    } state_t;

    state_t current_state, next_state;  // State registers

    logic [31:0] sum;
    int i, j;

    logic [31:0] input_matrix [0:8];
    assign input_matrix = '{input_matrix_0, input_matrix_1, input_matrix_2,
                            input_matrix_3, input_matrix_4, input_matrix_5,
                            input_matrix_6, input_matrix_7, input_matrix_8};

    logic [31:0] output_matrix [0:3];

    // State and computation control logic
    always_ff @(posedge clk or posedge reset) begin
        if (reset) begin
            // Reset all values to initial state
            current_state <= IDLE;
            i <= 0;
            j <= 0;
            sum <= 0;
        end else begin
            current_state <= next_state;

            // Performs computation
            if (current_state == COMPUTE) begin
                sum = 0;

                if (i == 0 && j == 0) begin

                    sum = input_matrix[0] * 1 + input_matrix[1] * 0 +
                          input_matrix[3] * 1 + input_matrix[4] * 0;
                end else if (i == 0 && j == 1) begin

                    sum = input_matrix[1] * 1 + input_matrix[2] * 0 +
                          input_matrix[4] * 1 + input_matrix[5] * 0;
                end else if (i == 1 && j == 0) begin

                    sum = input_matrix[3] * 1 + input_matrix[4] * 0 +
                          input_matrix[6] * 1 + input_matrix[7] * 0;
                end else if (i == 1 && j == 1) begin

                    sum = input_matrix[4] * 1 + input_matrix[5] * 0 +
                          input_matrix[7] * 1 + input_matrix[8] * 0;
                end

                // Store result into the output matrix
                output_matrix[i * 2 + j] <= sum;
                if (j < 1) begin
                    j <= j + 1;
                end else begin
                    j <= 0;
                    i <= i + 1;
                end
            end
        end
    end

    // Next-state logic
    always_comb begin
        output_matrix_0 = output_matrix[0];
        output_matrix_1 = output_matrix[1];
        output_matrix_2 = output_matrix[2];
        output_matrix_3 = output_matrix[3];

        next_state = current_state;
        done = 0;

        // State machine transitions
        case (current_state)
            IDLE: begin
                if (start)
                    next_state = LOAD;
            end
            LOAD: begin
                next_state = COMPUTE;
            end
            COMPUTE: begin
                if (i == 1 && j == 1)
                    next_state = DONE;
            end
            DONE: begin
                done = 1;
                next_state = IDLE;
            end
            default: next_state = IDLE;
        endcase
    end

endmodule
```

Fig 1: State Machine for Matrix Convolution for 3*3 Input Matrix and 2*2 Kernel Matrix

Fig 2 shows the state machine for matrix convolution with a 5x5 input matrix and a 3x3 kernel matrix. It illustrates the scalability of the FSM design, showcasing its ability to handle larger matrices by adjusting states and operations to maintain efficiency. This file is attached as "matrix_convolution5_5.sv".



```
module sliding_window_convolution (
    input  logic          clk,
    input  logic          reset,
    input  logic          write,
    input  logic          read,
  input  logic [31:0]   data_in,
  output logic [31:0]  data_out,
    output logic          done
);

    // Parameters for matrix and kernel dimensions
    parameter INPUT_SIZE = 5;
    parameter KERNEL_SIZE = 3;
    parameter OUTPUT_SIZE = INPUT_SIZE - KERNEL_SIZE + 1;

    // Internal storage for input matrix and kernel
  logic [31:0] input_matrix[0:INPUT_SIZE-1][0:INPUT_SIZE-1];
  logic [31:0] kernel[0:KERNEL_SIZE-1][0:KERNEL_SIZE-1];
  logic [31:0] output_matrix[0:OUTPUT_SIZE-1][0:OUTPUT_SIZE-1];

    // Indices and sum
    logic [3:0] row, col, read_row, read_col;
  logic [31:0] sum;

    // State machine states
    typedef enum logic [2:0] {IDLE, WRITE_MATRIX, COMPUTE, READ_OUTPUT, DONE} state_t;
    state_t state, next_state;

    // Kernel initialization
    initial begin
        kernel[0][0] = 1; kernel[0][1] = 0; kernel[0][2] = 1;
        kernel[1][0] = 1; kernel[1][1] = 0; kernel[1][2] = 1;
        kernel[2][0] = 1; kernel[2][1] = 0; kernel[2][2] = 1;
    end

    // State machine logic
    always_ff @(posedge clk or posedge reset) begin
        if (reset) begin
            state <= IDLE;
            row <= 0;
            col <= 0;
            read_row <= 0;
            read_col <= 0;
            sum <= 0;
            done <= 0;
        end else begin
            state <= next_state;
            case (state)
                IDLE: begin
                    if (write) begin
                        row <= 0;
                        col <= 0;
                    end
                end

                WRITE_MATRIX: begin
                    if (write) begin
                        input_matrix[row][col] <= data_in;
                        if (col == INPUT_SIZE - 1) begin
                            col <= 0;
                            if (row == INPUT_SIZE - 1) row <= 0;
                            else row <= row + 1;
                        end else begin
                            col <= col + 1;
                        end
                    end
                end

                COMPUTE: begin
                    sum = 0;
                    for (int i = 0; i < KERNEL_SIZE; i++) begin
                        for (int j = 0; j < KERNEL_SIZE; j++) begin
                            sum += input_matrix[row + i][col + j] * kernel[i][j];
                        end
                    end
                    output_matrix[row][col] <= sum;
                    if (col == OUTPUT_SIZE - 1) begin
                        col <= 0;
                        if (row == OUTPUT_SIZE - 1) done <= 1;
                        else row <= row + 1;
                    end else begin
                        col <= col + 1;
                    end
                end

                READ_OUTPUT: begin
                    if (read) begin
                        data_out <= output_matrix[read_row][read_col];
                        if (read_col == OUTPUT_SIZE - 1) begin
                            read_col <= 0;
                            if (read_row == OUTPUT_SIZE - 1) begin
                                read_row <= 0;
                                done <= 1;
                            end else read_row <= read_row + 1;
                        end else read_col <= read_col + 1;
                    end
                end

                DONE: begin
                    done <= 1;
                end
            endcase
        end
    end

    // Next state logic
    always_comb begin
        next_state = state;
        case (state)
            IDLE: next_state = write ? WRITE_MATRIX : IDLE;
            WRITE_MATRIX: next_state = (row == INPUT_SIZE - 1 && col == INPUT_SIZE - 1) ? COMPUTE :
WRITE_MATRIX;
            COMPUTE: next_state = (row == OUTPUT_SIZE - 1 && col == OUTPUT_SIZE - 1) ? READ_OUTPUT :
COMPUTE;
            READ_OUTPUT: next_state = (read_row == OUTPUT_SIZE - 1 && read_col == OUTPUT_SIZE - 1 &&
done) ? DONE : READ_OUTPUT;
            DONE: next_state = IDLE;
        endcase
    end
endmodule
```

Fig 2: State Machine for Matrix Convolution for 5*5 Input Matrix and 3*3 Kernel Matrix

3.2 Integration with Avalon-MM Interface

Communication between the matrix convolution module and the processor was facilitated by implementing an Avalon-MM interface for the module. This interface contemplate a memory mapping communication that facilitates the data interchange between the FPGA and the CPU. Address decoding was used while integrating, WRITE logic was equally included, and READ-back was part of the integrating procedure.

Key Functionalities of the Integration:

- Write Operation:

In the following, when avs_s1_write is 1, the elements in the input matrix and control signals are stored in the internal registers.

By specifying the behavior of internal register avs_s1_address it is possible to direct data transfer towards the desired location.

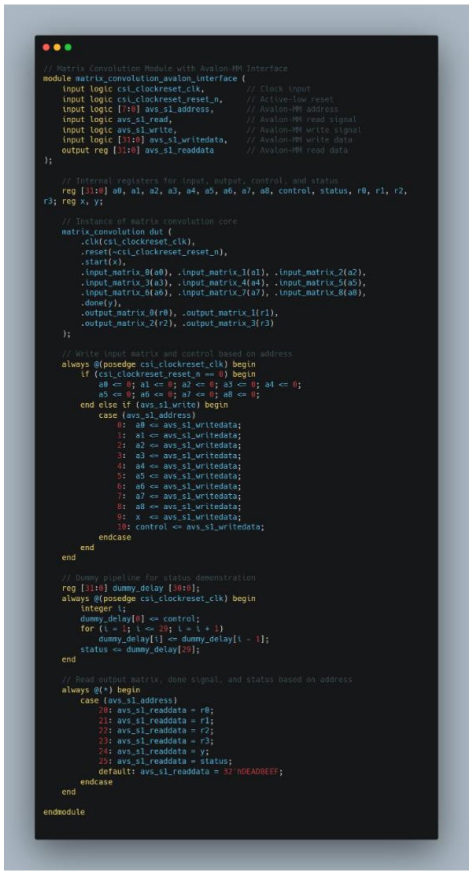
- Read Operation:

In the second stage, when avs_s1_read is asserted, a corresponding element in the result matrix, done signal and status register address are read. This makes it easy for the processor to access computed results when the need arise.

- Control and Status Signals:

A control register exists to control the convolution computation while a status register is used to give information concerning the computation.

- A dummy delay pipeline is implemented for some illustrations in order to synchronize control and output signals correctly.



```verilog
// Matrix Convolution Module with Avalon-MM Interface
module matrix_convolution_avalon_interface (
    input logic csi_clockreset_clk,          // Clock input
    input logic csi_clockreset_reset_n,      // Active-low reset
    input logic [7:0] avs_s1_address,        // Avalon-MM address
    input logic avs_s1_read,                 // Avalon-MM read signal
    input logic avs_s1_write,                // Avalon-MM write signal
    input logic [31:0] avs_s1_writedata,     // Avalon-MM write data
    output reg [31:0] avs_s1_readdata        // Avalon-MM read data
);

    // Internal registers for input, output, control, and status
    reg [31:0] a0, a1, a2, a3, a4, a5, a6, a7, a8, control, status, r0, r1, r2,
r3; reg x, y;

    // Instance of matrix convolution core
    matrix_convolution dut (
        .clk(csi_clockreset_clk),
        .reset(~csi_clockreset_reset_n),
        .start(x),
        .input_matrix_0(a0), .input_matrix_1(a1), .input_matrix_2(a2),
        .input_matrix_3(a3), .input_matrix_4(a4), .input_matrix_5(a5),
        .input_matrix_6(a6), .input_matrix_7(a7), .input_matrix_8(a8),
        .done(y),
        .output_matrix_0(r0), .output_matrix_1(r1),
        .output_matrix_2(r2), .output_matrix_3(r3)
    );

    // Write input matrix and control based on address
    always @(posedge csi_clockreset_clk) begin
        if (csi_clockreset_reset_n == 0) begin
            a0 <= 0; a1 <= 0; a2 <= 0; a3 <= 0; a4 <= 0;
            a5 <= 0; a6 <= 0; a7 <= 0; a8 <= 0;
        end else if (avs_s1_write) begin
            case (avs_s1_address)
                0:  a0 <= avs_s1_writedata;
                1:  a1 <= avs_s1_writedata;
                2:  a2 <= avs_s1_writedata;
                3:  a3 <= avs_s1_writedata;
                4:  a4 <= avs_s1_writedata;
                5:  a5 <= avs_s1_writedata;
                6:  a6 <= avs_s1_writedata;
                7:  a7 <= avs_s1_writedata;
                8:  a8 <= avs_s1_writedata;
                9:  x  <= avs_s1_writedata;
                10: control <= avs_s1_writedata;
            endcase
        end
    end

    // Dummy pipeline for status demonstration
    reg [31:0] dummy_delay [30:0];
    always @(posedge csi_clockreset_clk) begin
        integer i;
        dummy_delay[0] <= control;
        for (i = 1; i <= 29; i = i + 1)
            dummy_delay[i] <= dummy_delay[i - 1];
        status <= dummy_delay[29];
    end

    // Read output matrix, done signal, and status based on address
    always @(*) begin
        case (avs_s1_address)
            20: avs_s1_readdata = r0;
            21: avs_s1_readdata = r1;
            22: avs_s1_readdata = r2;
            23: avs_s1_readdata = r3;
            24: avs_s1_readdata = y;
            25: avs_s1_readdata = status;
            default: avs_s1_readdata = 32'hDEADBEEF;
        endcase
    end

endmodule
```

Fig 3: Avalon-MM interface for the module Matrix Convolution for 3*3 Input Matrix and 2*2 Output Matrix

Fig 3 illustrates the Avalon-MM interface implemented for the matrix convolution module. It shows the memory-mapped communication protocol, detailing the flow of read and write operations between the processor and FPGA. Control signals and address decoding are used to ensure proper data transfer. This file is attached as “matrix_convolution_avalon_interface.sv”. In this project, we used 3*3 input matrix module for Hardware software codesign.

Signal	**Direction**	**Description**
avs_s1_write	Input	Write enable for input data
avs_s1_read	Input	Read enable for output data
avs_s1_address	Input	Specifies the target register
avs_s1_writedata	Input	Input data for the convolution
avs_s1_readdata	Output	Output data (convolution result)

3.3 Custom Component Creation

The netlist description of the matrix convolution accelerator was implemented with the help of Intel Quartus Prime to obtain the custom component. The design included the following steps:

- FSM Implementation: The FSM and associated data path were described in system verilog to implement the functionality of convolution.
- Avalon-MM Interface Integration: It became necessary to map signals to the Avalon-MM protocol.
- System Integration: This was implemented in the Platform Designer (Qsys) and memory mapped was initialized with a base address.
- Header File Generation: The .socinfo file was further used to create another header point for the software application to use, the macros included concerned memory-mapping.

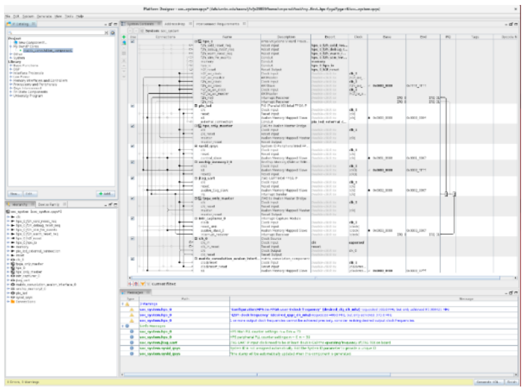
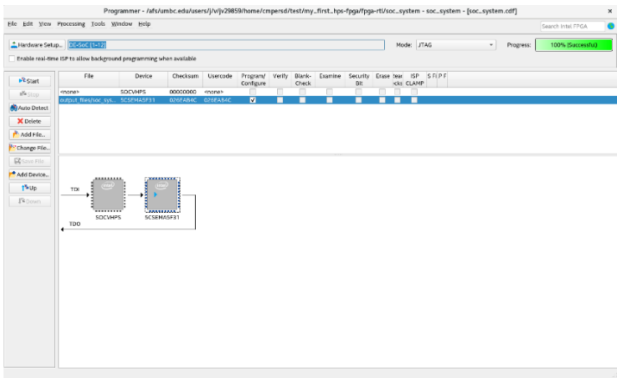


Fig 4: Custom Component Creation for Matrix Convolution

Fig 4 shows the process of custom component creation for the matrix convolution module. It highlights the steps taken in Intel Quartus Prime, including FSM implementation, Avalon-MM integration, and system-level configuration, to create a functional hardware component.



Programmer - /afs/umbc.edu/users/j/v/jv29859/home/cmpersd/test/my_first_hps-fpga/fpga-rtl/soc_system - soc_system - [soc_system.cdf]

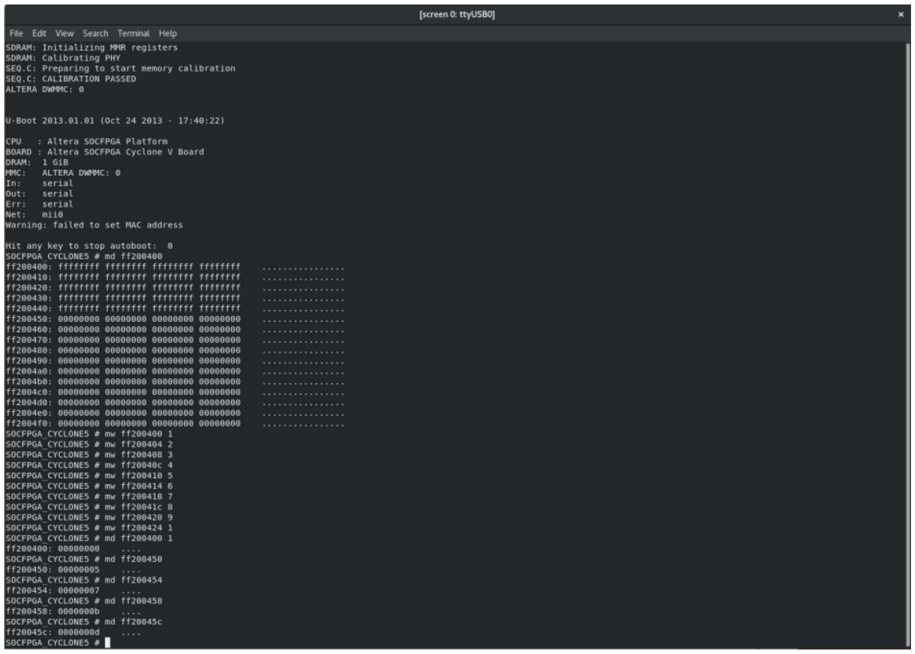
Hardware Setup... DE-SoC [1-12] Mode: JTAG Progress: 100% (Successful)

Enable real-time ISP to allow background programming when available

File	Device	Checksum	Usercode	Program/ Configure	Verify	Blank-Check	Examine	Security Bit	Erase	ISP CLAMP
<none>	SOCVHPS	00000000	<none>	☐	☐	☐	☐	☐	☐	☐
output_files/soc_sys...	5CSEMA5F31	026EA84C	026EA84C	✓	☐	☐	☐	☐	☐	☐

Fig 5: FPGA is programmed successfully

Fig 5 confirms the successful programming of the FPGA for executing matrix convolution operations. It demonstrates the completion of hardware configuration, readying the system for validation and testing.



```
SDRAM: Initializing MMR registers
SDRAM: Calibrating PHY
SEQ.C: Preparing to start memory calibration
SEQ.C: CALIBRATION PASSED
ALTERA DWMMC: 0


U-Boot 2013.01.01 (Oct 24 2013 - 17:40:22)

CPU   : Altera SOCFPGA Platform
BOARD : Altera SOCFPGA Cyclone V Board
DRAM:  1 GiB
MMC:   ALTERA DWMMC: 0
In:    serial
Out:   serial
Err:   serial
Net:   mii0
Warning: failed to set MAC address

Hit any key to stop autoboot:  0
SOCFPGA_CYCLONE5 # md ff200400
ff200400: ffffffff ffffffff ffffffff ffffffff    ................
ff200410: ffffffff ffffffff ffffffff ffffffff    ................
ff200420: ffffffff ffffffff ffffffff ffffffff    ................
ff200430: ffffffff ffffffff ffffffff ffffffff    ................
ff200440: ffffffff ffffffff ffffffff ffffffff    ................
ff200450: 00000000 00000000 00000000 00000000    ................
ff200460: 00000000 00000000 00000000 00000000    ................
ff200470: 00000000 00000000 00000000 00000000    ................
ff200480: 00000000 00000000 00000000 00000000    ................
ff200490: 00000000 00000000 00000000 00000000    ................
ff2004a0: 00000000 00000000 00000000 00000000    ................
ff2004b0: 00000000 00000000 00000000 00000000    ................
ff2004c0: 00000000 00000000 00000000 00000000    ................
ff2004d0: 00000000 00000000 00000000 00000000    ................
ff2004e0: 00000000 00000000 00000000 00000000    ................
ff2004f0: 00000000 00000000 00000000 00000000    ................
SOCFPGA_CYCLONE5 # mw ff200400 1
SOCFPGA_CYCLONE5 # mw ff200404 2
SOCFPGA_CYCLONE5 # mw ff200408 3
SOCFPGA_CYCLONE5 # mw ff20040c 4
SOCFPGA_CYCLONE5 # mw ff200410 5
SOCFPGA_CYCLONE5 # mw ff200414 6
SOCFPGA_CYCLONE5 # mw ff200418 7
SOCFPGA_CYCLONE5 # mw ff20041c 8
SOCFPGA_CYCLONE5 # mw ff200420 9
SOCFPGA_CYCLONE5 # mw ff200424 1
SOCFPGA_CYCLONE5 # md ff200400 1
ff200400: 00000000    ....
SOCFPGA_CYCLONE5 # md ff200450
ff200450: 00000005    ....
SOCFPGA_CYCLONE5 # md ff200454
ff200454: 00000007    ....
SOCFPGA_CYCLONE5 # md ff200458
ff200458: 0000000b    ....
SOCFPGA_CYCLONE5 # md ff20045c
ff20045c: 0000000d    ....
SOCFPGA_CYCLONE5 #
```

Fig 6: Writing and Reading into registers using U-Boot terminal

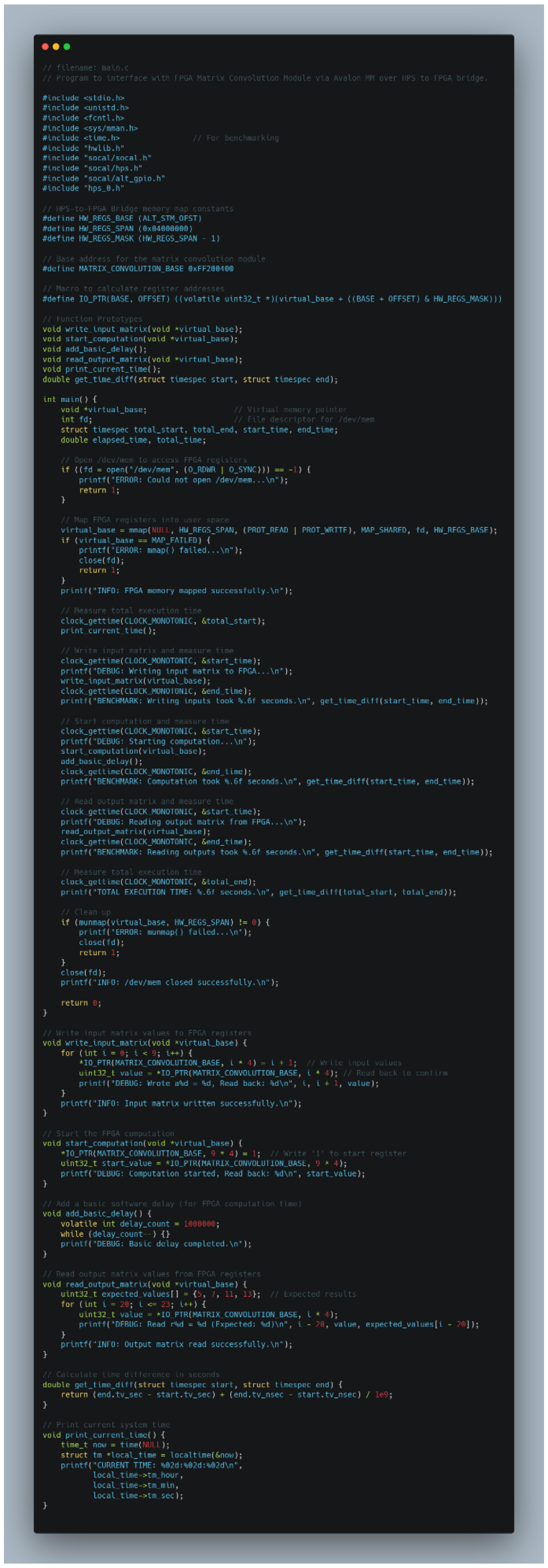
Fig 6 shows the writing and reading of data into registers using the U-Boot terminal. It provides an example of how input data is stored and output data is retrieved, illustrating the interaction between hardware and software components.

3.4 Software Hardware Codesign

The specific software application also known as main.c was completed and designed to directly operate with the hardware accelerators. It performed the following operations:

- Writing Input Matrix: The processor stored this input matrix with 3 rows and 3 columns to the hardware through the memory-map I/O.
- Triggering Computation: After accepting the input, the hardware went to the COMPUTE state on its own accord.
- Reading Output Matrix: The processor also got the computed 2*2 output matrix from the hardware.

Fig 7 depicts the software and hardware integration code (main.c). It demonstrates the functions for writing input matrices, triggering computation, and reading output matrices via the Avalon-MM interface, ensuring smooth operation of the hardware accelerator. The file is attached as “main.c”.



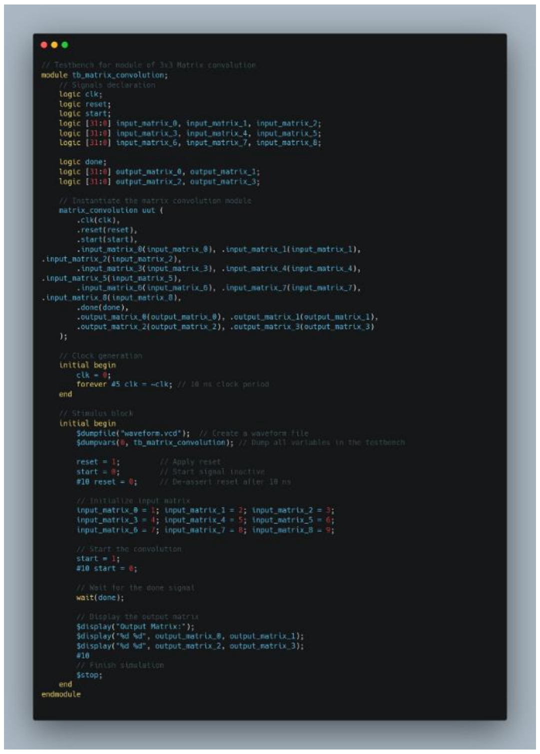
```c
// filename: main.c
// Program to interface with FPGA Matrix Convolution Module via Avalon-MM over HPS-to-FPGA bridge.

#include <stdio.h>
#include <unistd.h>
#include <fcntl.h>
#include <sys/mman.h>
#include <time.h>                // For benchmarking
#include "hwlib.h"
#include "socal/socal.h"
#include "socal/hps.h"
#include "socal/alt_gpio.h"
#include "hps_0.h"

// HPS-to-FPGA Bridge memory map constants
#define HW_REGS_BASE (ALT_STM_OFST)
#define HW_REGS_SPAN (0x04000000)
#define HW_REGS_MASK (HW_REGS_SPAN - 1)

// Base address for the matrix convolution module
#define MATRIX_CONVOLUTION_BASE 0xFF200400

// Macro to calculate register addresses
#define IO_PTR(BASE, OFFSET) ((volatile uint32_t *)(virtual_base + ((BASE + OFFSET) & HW_REGS_MASK)))

// Function Prototypes
void write_input_matrix(void *virtual_base);
void start_computation(void *virtual_base);
void add_basic_delay();
void read_output_matrix(void *virtual_base);
void print_current_time();
double get_time_diff(struct timespec start, struct timespec end);

int main() {
    void *virtual_base;                   // Virtual memory pointer
    int fd;                               // File descriptor for /dev/mem
    struct timespec total_start, total_end, start_time, end_time;
    double elapsed_time, total_time;

    // Open /dev/mem to access FPGA registers
    if ((fd = open("/dev/mem", (O_RDWR | O_SYNC))) == -1) {
        printf("ERROR: Could not open /dev/mem...\n");
        return 1;
    }

    // Map FPGA registers into user space
    virtual_base = mmap(NULL, HW_REGS_SPAN, (PROT_READ | PROT_WRITE), MAP_SHARED, fd, HW_REGS_BASE);
    if (virtual_base == MAP_FAILED) {
        printf("ERROR: mmap() failed...\n");
        close(fd);
        return 1;
    }
    printf("INFO: FPGA memory mapped successfully.\n");

    // Measure total execution time
    clock_gettime(CLOCK_MONOTONIC, &total_start);
    print_current_time();

    // Write input matrix and measure time
    clock_gettime(CLOCK_MONOTONIC, &start_time);
    printf("DEBUG: Writing input matrix to FPGA...\n");
    write_input_matrix(virtual_base);
    clock_gettime(CLOCK_MONOTONIC, &end_time);
    printf("BENCHMARK: Writing inputs took %.6f seconds.\n", get_time_diff(start_time, end_time));

    // Start computation and measure time
    clock_gettime(CLOCK_MONOTONIC, &start_time);
    printf("DEBUG: Starting computation...\n");
    start_computation(virtual_base);
    add_basic_delay();
    clock_gettime(CLOCK_MONOTONIC, &end_time);
    printf("BENCHMARK: Computation took %.6f seconds.\n", get_time_diff(start_time, end_time));

    // Read output matrix and measure time
    clock_gettime(CLOCK_MONOTONIC, &start_time);
    printf("DEBUG: Reading output matrix from FPGA...\n");
    read_output_matrix(virtual_base);
    clock_gettime(CLOCK_MONOTONIC, &end_time);
    printf("BENCHMARK: Reading outputs took %.6f seconds.\n", get_time_diff(start_time, end_time));

    // Measure total execution time
    clock_gettime(CLOCK_MONOTONIC, &total_end);
    printf("TOTAL EXECUTION TIME: %.6f seconds.\n", get_time_diff(total_start, total_end));

    // Clean up
    if (munmap(virtual_base, HW_REGS_SPAN) != 0) {
        printf("ERROR: munmap() failed...\n");
        close(fd);
        return 1;
    }
    close(fd);
    printf("INFO: /dev/mem closed successfully.\n");

    return 0;
}

// Write input matrix values to FPGA registers
void write_input_matrix(void *virtual_base) {
    for (int i = 0; i < 9; i++) {
        *IO_PTR(MATRIX_CONVOLUTION_BASE, i * 4) = i + 1;  // Write input values
        uint32_t value = *IO_PTR(MATRIX_CONVOLUTION_BASE, i * 4); // Read back to confirm
        printf("DEBUG: Wrote a%d = %d, Read back: %d\n", i, i + 1, value);
    }
    printf("INFO: Input matrix written successfully.\n");
}

// Start the FPGA computation
void start_computation(void *virtual_base) {
    *IO_PTR(MATRIX_CONVOLUTION_BASE, 9 * 4) = 1;  // Write '1' to start register
    uint32_t start_value = *IO_PTR(MATRIX_CONVOLUTION_BASE, 9 * 4);
    printf("DEBUG: Computation started, Read back: %d\n", start_value);
}

// Add a basic software delay (for FPGA computation time)
void add_basic_delay() {
    volatile int delay_count = 1000000;
    while (delay_count--) {}
    printf("DEBUG: Basic delay completed.\n");
}

// Read output matrix values from FPGA registers
void read_output_matrix(void *virtual_base) {
    uint32_t expected_values[] = {5, 7, 11, 13};  // Expected results
    for (int i = 20; i <= 23; i++) {
        uint32_t value = *IO_PTR(MATRIX_CONVOLUTION_BASE, i * 4);
        printf("DEBUG: Read r%d = %d (Expected: %d)\n", i - 20, value, expected_values[i - 20]);
    }
    printf("INFO: Output matrix read successfully.\n");
}

// Calculate time difference in seconds
double get_time_diff(struct timespec start, struct timespec end) {
    return (end.tv_sec - start.tv_sec) + (end.tv_nsec - start.tv_nsec) / 1e9;
}

// Print current system time
void print_current_time() {
    time_t now = time(NULL);
    struct tm *local_time = localtime(&now);
    printf("CURRENT TIME: %02d:%02d:%02d\n",
           local_time->tm_hour,
           local_time->tm_min,
           local_time->tm_sec);
}
```

Fig 7: Software and Hardware Integration Code (main.c)

3.5 Testbench and Validation

To demonstrate the implementation of the FSM and the data path, a system verilog testbench has been created and implemented. The testbench simulated across a input scenario and captured waveform to verify:

- Proper state transition in the FSM has been verified.
- Proper calculation of the given convolution operation is done.
- Avalon-MM interface to FSM and vice versa must exchange the correct data.
- Waveform analysis shows that the design meets functional and timing specifications.
- Here, two testbenches are written, one for 3*3 input matrix and 2*2 output matrix. On the other hand, another testbench is for 5*5 input matrix and 3*3 output matrix. Those results are shown below respectively.

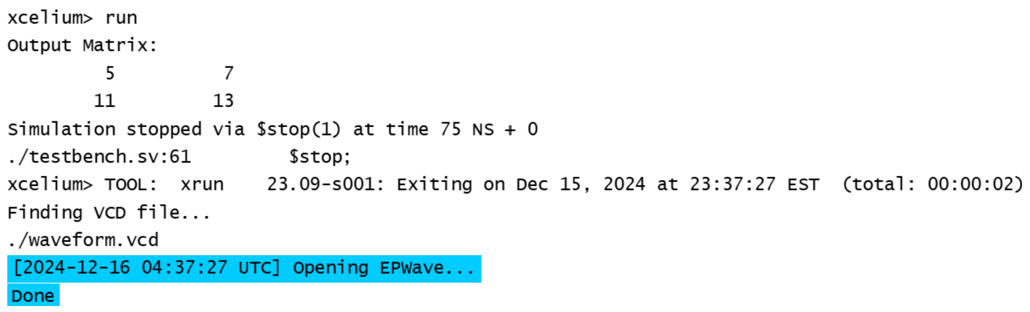


```
// Testbench for module of 3x3 Matrix convolution
module tb_matrix_convolution;
    // Signals declaration
    logic clk;
    logic reset;
    logic start;
    logic [31:0] input_matrix_0, input_matrix_1, input_matrix_2;
    logic [31:0] input_matrix_3, input_matrix_4, input_matrix_5;
    logic [31:0] input_matrix_6, input_matrix_7, input_matrix_8;

    logic done;
    logic [31:0] output_matrix_0, output_matrix_1;
    logic [31:0] output_matrix_2, output_matrix_3;

    // Instantiate the matrix convolution module
    matrix_convolution uut (
        .clk(clk),
        .reset(reset),
        .start(start),
        .input_matrix_0(input_matrix_0), .input_matrix_1(input_matrix_1),
.input_matrix_2(input_matrix_2),
        .input_matrix_3(input_matrix_3), .input_matrix_4(input_matrix_4),
.input_matrix_5(input_matrix_5),
        .input_matrix_6(input_matrix_6), .input_matrix_7(input_matrix_7),
.input_matrix_8(input_matrix_8),
        .done(done),
        .output_matrix_0(output_matrix_0), .output_matrix_1(output_matrix_1),
        .output_matrix_2(output_matrix_2), .output_matrix_3(output_matrix_3)
    );

    // Clock generation
    initial begin
        clk = 0;
        forever #5 clk = ~clk; // 10 ns clock period
    end

    // Stimulus block
    initial begin
        $dumpfile("waveform.vcd");  // Create a waveform file
        $dumpvars(0, tb_matrix_convolution); // Dump all variables in the testbench

        reset = 1;          // Apply reset
        start = 0;          // Start signal inactive
        #10 reset = 0;      // De-assert reset after 10 ns

        // Initialize input matrix
        input_matrix_0 = 1; input_matrix_1 = 2; input_matrix_2 = 3;
        input_matrix_3 = 4; input_matrix_4 = 5; input_matrix_5 = 6;
        input_matrix_6 = 7; input_matrix_7 = 8; input_matrix_8 = 9;

        // Start the convolution
        start = 1;
        #10 start = 0;

        // Wait for the done signal
        wait(done);

        // Display the output matrix
        $display("Output Matrix:");
        $display("%d %d", output_matrix_0, output_matrix_1);
        $display("%d %d", output_matrix_2, output_matrix_3);
        #10
        // Finish simulation
        $stop;
    end
endmodule
```

Fig 8: Testbench for Matrix Convolution module for 3*3 Input Matrix and 2*2 output Matrix

Fig 8 presents the SystemVerilog testbench for the matrix convolution module with a 3x3 input matrix and a 2x2 output matrix. It validates the FSM and datapath, checking for proper state transitions and correct data exchange through the Avalon-MM interface. This file is attached as “matrix_convolution3_3tb.sv”.

Fig 9 shows the simulation results for the 3x3 input matrix and 2x2 output matrix. It verifies the correctness of the matrix convolution module by comparing the expected output with the computed results.



```
xcelium> run
Output Matrix:
         5          7
        11         13
Simulation stopped via $stop(1) at time 75 NS + 0
./testbench.sv:61         $stop;
xcelium> TOOL:  xrun    23.09-s001: Exiting on Dec 15, 2024 at 23:37:27 EST  (total: 00:00:02)
Finding VCD file...
./waveform.vcd
[2024-12-16 04:37:27 UTC] Opening EPWave...
Done
```

Fig 9: Simulation Results for Matrix Convolution module for 3*3 Input Matrix and 2*2 output Matrix

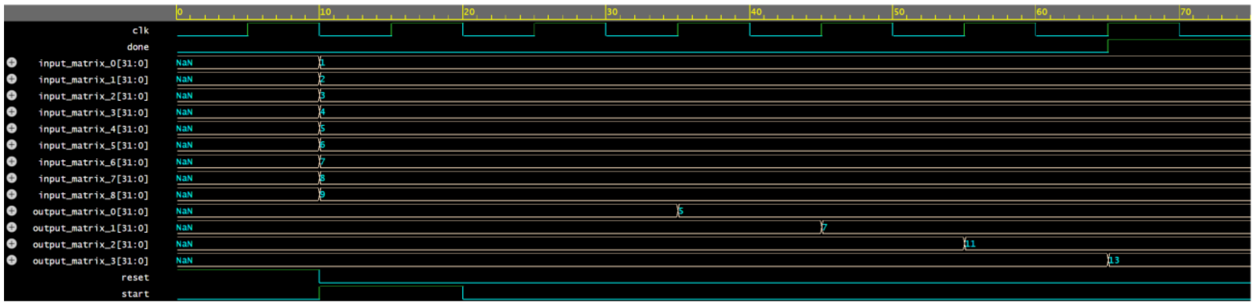


Fig 10: Waveform for Matrix Convolution module for 3*3 Input Matrix and 2*2 output Matrix

Fig 10 displays the waveform for the matrix convolution module with a 3x3 input matrix and a 2x2 output matrix. It confirms the proper timing and functional behavior of the FSM and data path during the computation process.

3.6 Clock Cycle Analysis

The clock cycle waveform demonstrates the behavior of the FSM-based matrix convolution hardware. The analysis is as follows:

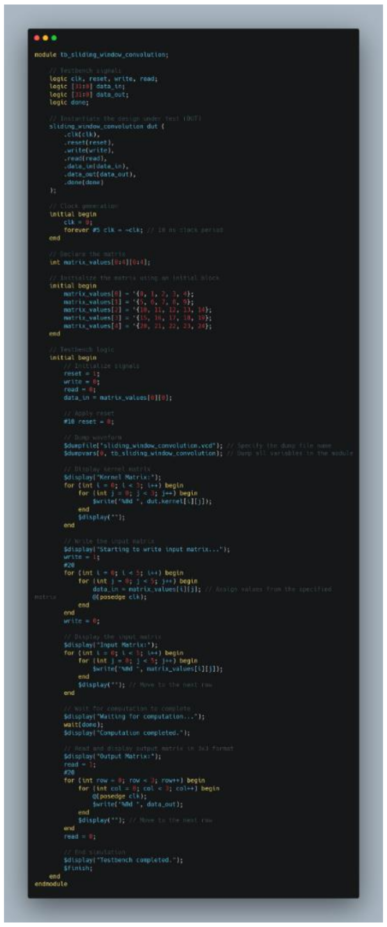
1. **Input Matrix Loading (WRITE_MATRIX):**
    - All input matrix elements (input_matrix_0 to input_matrix_8) are loaded simultaneously within a single clock cycle.
    - **Total Clock Cycles for Input Load: 1 clock cycle**.
2. **Convolution Computation (COMPUTE):**
    - The computation phase processes all output elements together and completes in just 2 clock cycles.

- **Total Clock Cycles for Computation: 2 clock cycles**.

3. **Output Matrix Generation (READ_OUTPUT):**
    - The output matrix elements (output_matrix_0 to output_matrix_3) are produced sequentially, one per clock cycle.
    - For **4 output elements**, the total time for the readout phase is: 4elements×1clock cycle/element = 4clock cycles.
4. **Total Execution Time:**
    Summing up all phases, the total clock cycles required are:
    - **Input Write:** 1 clock cycle
    - **Computation:** 2 clock cycles
    - **Output Read:** 4 clock cycles

Total Execution Time: 7 clock cycles

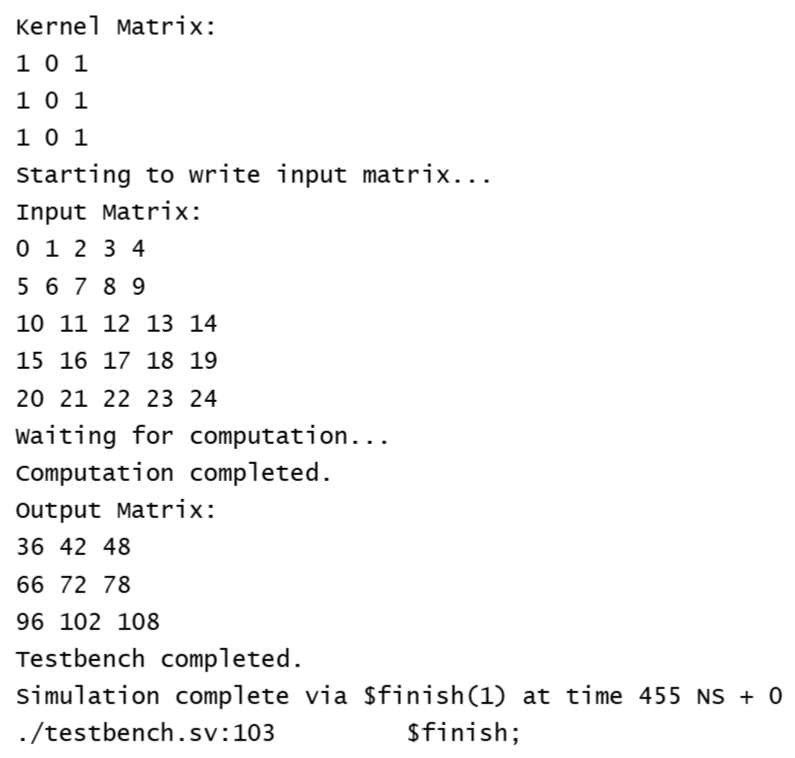
The total execution time for the matrix convolution operation is **7 clock cycles**, demonstrating an optimized and efficient hardware implementation.



```
module tb_sliding_window_convolution;

    // Testbench signals
    logic clk, reset, write, read;
    logic [31:0] data_in;
    logic [31:0] data_out;
    logic done;

    // Instantiate the design under test (DUT)
    sliding_window_convolution dut (
        .clk(clk),
        .reset(reset),
        .write(write),
        .read(read),
        .data_in(data_in),
        .data_out(data_out),
        .done(done)
    );

    // Clock generation
    initial begin
        clk = 0;
        forever #5 clk = ~clk; // 10 ns clock period
    end

    // Declare the matrix
    int matrix_values[0:4][0:4];

    // Initialize the matrix using an initial block
    initial begin
        matrix_values[0] = '{0, 1, 2, 3, 4};
        matrix_values[1] = '{5, 6, 7, 8, 9};
        matrix_values[2] = '{10, 11, 12, 13, 14};
        matrix_values[3] = '{15, 16, 17, 18, 19};
        matrix_values[4] = '{20, 21, 22, 23, 24};
    end

    // Testbench logic
    initial begin
        // Initialize signals
        reset = 1;
        write = 0;
        read = 0;
        data_in = matrix_values[0][0];

        // Apply reset
        #10 reset = 0;

        // Dump waveform
        $dumpfile("sliding_window_convolution.vcd"); // Specify the dump file name
        $dumpvars(0, tb_sliding_window_convolution); // Dump all variables in the module

        // Display kernel matrix
        $display("Kernel Matrix:");
        for (int i = 0; i < 3; i++) begin
            for (int j = 0; j < 3; j++) begin
                $write("%0d ", dut.kernel[i][j]);
            end
            $display("");
        end

        // Write the input matrix
        $display("Starting to write input matrix...");
        write = 1;
        #20
        for (int i = 0; i < 5; i++) begin
            for (int j = 0; j < 5; j++) begin
                data_in = matrix_values[i][j]; // Assign values from the specified
matrix          @(posedge clk);
            end
        end
        write = 0;

        // Display the input matrix
        $display("Input Matrix:");
        for (int i = 0; i < 5; i++) begin
            for (int j = 0; j < 5; j++) begin
                $write("%0d ", matrix_values[i][j]);
            end
            $display(""); // Move to the next row
        end

        // Wait for computation to complete
        $display("Waiting for computation...");
        wait(done);
        $display("Computation completed.");

        // Read and display output matrix in 3x3 format
        $display("Output Matrix:");
        read = 1;
        #20
        for (int row = 0; row < 3; row++) begin
            for (int col = 0; col < 3; col++) begin
                @(posedge clk);
                $write("%0d ", data_out);
            end
            $display(""); // Move to the next row
        end
        read = 0;

        // End simulation
        $display("Testbench completed.");
        $finish;
    end
endmodule
```

Fig 11: Testbench for Matrix Convolution module for 5*5 Input Matrix and 3*3 output Matrix

Fig 11 depicts the testbench for the matrix convolution module with a 5x5 input matrix and a 3x3 output matrix. It expands on the design's scalability and demonstrates accurate computation for larger data sizes. This file is attached as “matrix_convolution5_5tb.sv”.



```
Kernel Matrix:
1 0 1
1 0 1
1 0 1
Starting to write input matrix...
Input Matrix:
0 1 2 3 4
5 6 7 8 9
10 11 12 13 14
15 16 17 18 19
20 21 22 23 24
Waiting for computation...
Computation completed.
Output Matrix:
36 42 48
66 72 78
96 102 108
Testbench completed.
Simulation complete via $finish(1) at time 455 NS + 0
./testbench.sv:103         $finish;
```

Fig 12: Simulation Results for Matrix Convolution module for 5*5 Input Matrix and 3*3 output Matrix

Fig 12 presents the simulation results for the 5x5 input matrix and 3x3 output matrix. It validates the hardware's functionality and ability to compute matrix convolutions for larger input data efficiently.

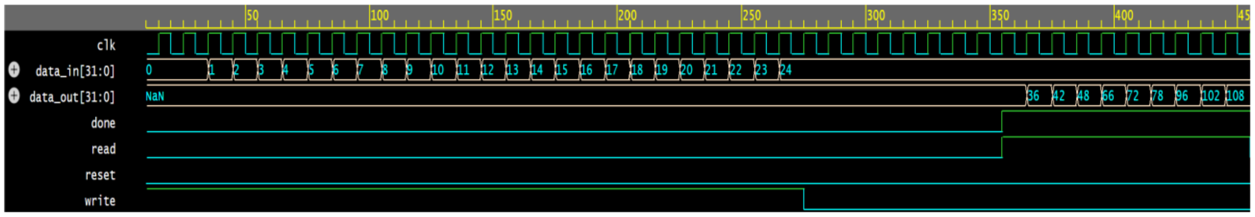


Fig 13: Waveform for Matrix Convolution module for 5*5 Input Matrix and 3*3 output Matrix

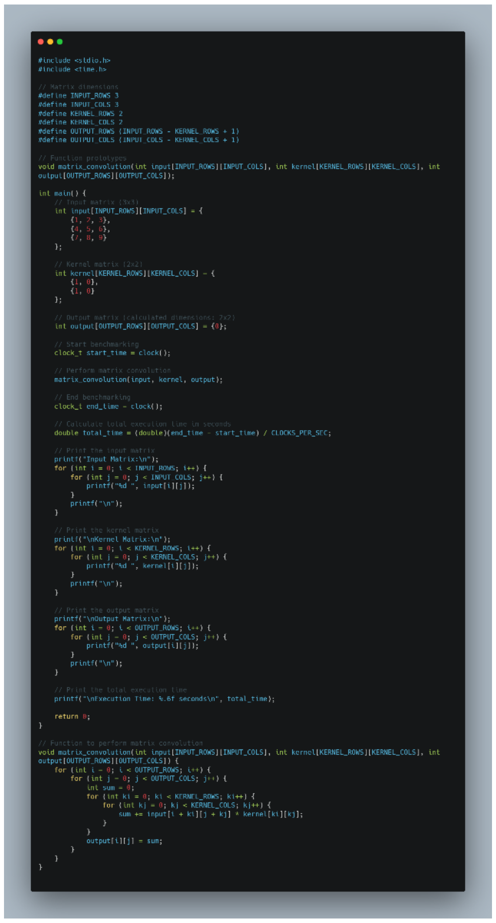
Fig 13 shows the waveform for the matrix convolution module with a 5x5 input matrix and a 3x3 output matrix. It illustrates the FSM’s timing behavior and verifies the correctness of state transitions and output generation.

3.7 Implementation with only software

To set up a performance baseline, a C program for software only implementation for matrix convolution operation was designed. This implementation performed the following steps:

- Input Matrix: The 3*3 input matrix and 2*2 kernel were described in the program.

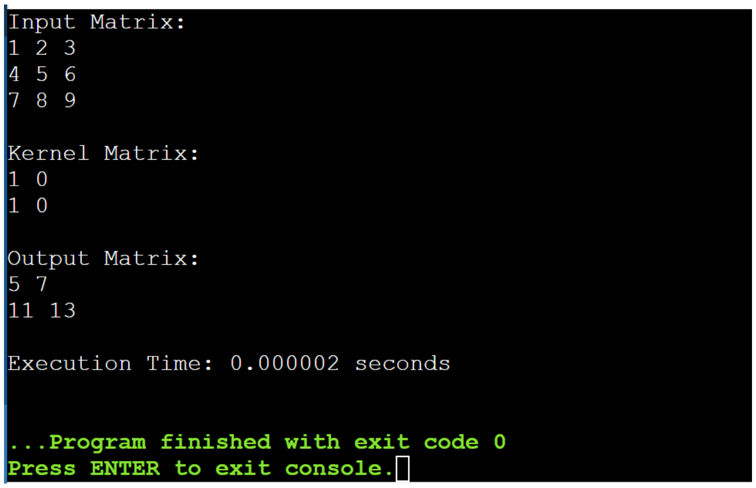
- Convolution Operation: Kernel sliding over the input matrix and performing the dot product computation was implemented using nested loop architecture.
- Output Matrix: To validate the outputs, they were saved in a 2*2 matrix output section.
- Here, two software implementations are written, one for 3*3 input matrix and 2*2 output matrix. On the other hand, another is for 5*5 input matrix and 3*3 output matrix. Those results are shown below respectively.



```c
#include <stdio.h>
#include <time.h>

// Matrix dimensions
#define INPUT_ROWS 3
#define INPUT_COLS 3
#define KERNEL_ROWS 2
#define KERNEL_COLS 2
#define OUTPUT_ROWS (INPUT_ROWS - KERNEL_ROWS + 1)
#define OUTPUT_COLS (INPUT_COLS - KERNEL_COLS + 1)

// Function prototypes
void matrix_convolution(int input[INPUT_ROWS][INPUT_COLS], int kernel[KERNEL_ROWS][KERNEL_COLS], int
output[OUTPUT_ROWS][OUTPUT_COLS]);

int main() {
    // Input matrix (3x3)
    int input[INPUT_ROWS][INPUT_COLS] = {
        {1, 2, 3},
        {4, 5, 6},
        {7, 8, 9}
    };

    // Kernel matrix (2x2)
    int kernel[KERNEL_ROWS][KERNEL_COLS] = {
        {1, 0},
        {1, 0}
    };

    // Output matrix (calculated dimensions: 2x2)
    int output[OUTPUT_ROWS][OUTPUT_COLS] = {0};

    // Start benchmarking
    clock_t start_time = clock();

    // Perform matrix convolution
    matrix_convolution(input, kernel, output);

    // End benchmarking
    clock_t end_time = clock();

    // Calculate total execution time in seconds
    double total_time = (double)(end_time - start_time) / CLOCKS_PER_SEC;

    // Print the input matrix
    printf("Input Matrix:\n");
    for (int i = 0; i < INPUT_ROWS; i++) {
        for (int j = 0; j < INPUT_COLS; j++) {
            printf("%d ", input[i][j]);
        }
        printf("\n");
    }

    // Print the kernel matrix
    printf("\nKernel Matrix:\n");
    for (int i = 0; i < KERNEL_ROWS; i++) {
        for (int j = 0; j < KERNEL_COLS; j++) {
            printf("%d ", kernel[i][j]);
        }
        printf("\n");
    }

    // Print the output matrix
    printf("\nOutput Matrix:\n");
    for (int i = 0; i < OUTPUT_ROWS; i++) {
        for (int j = 0; j < OUTPUT_COLS; j++) {
            printf("%d ", output[i][j]);
        }
        printf("\n");
    }

    // Print the total execution time
    printf("\nExecution Time: %.6f seconds\n", total_time);

    return 0;
}

// Function to perform matrix convolution
void matrix_convolution(int input[INPUT_ROWS][INPUT_COLS], int kernel[KERNEL_ROWS][KERNEL_COLS], int
output[OUTPUT_ROWS][OUTPUT_COLS]) {
    for (int i = 0; i < OUTPUT_ROWS; i++) {
        for (int j = 0; j < OUTPUT_COLS; j++) {
            int sum = 0;
            for (int ki = 0; ki < KERNEL_ROWS; ki++) {
                for (int kj = 0; kj < KERNEL_COLS; kj++) {
                    sum += input[i + ki][j + kj] * kernel[ki][kj];
                }
            }
            output[i][j] = sum;
        }
    }
}
```

Fig 14: Software only Implementation of Matrix Convolution for 3*3 Input Matrix and 2*2 output Matrix

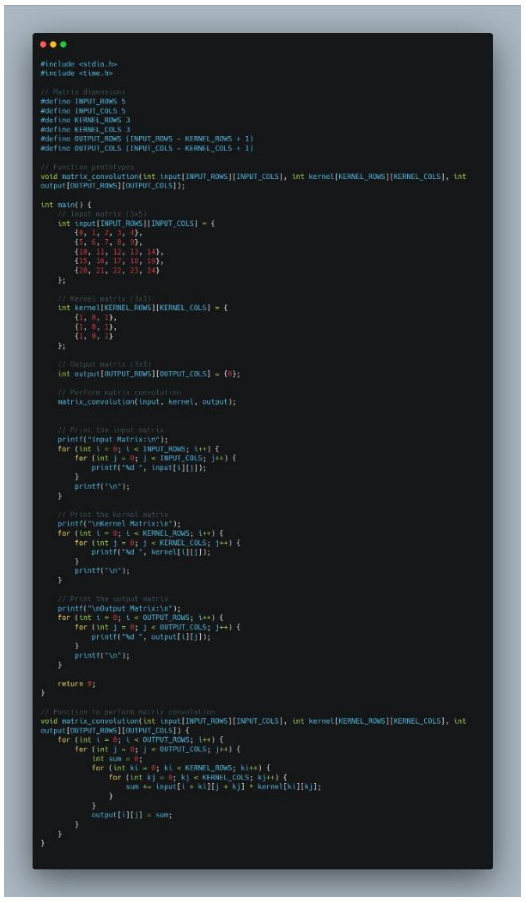
Fig 14 depicts the software-only implementation for matrix convolution using a 3x3 input matrix and a 2x2 kernel matrix. It highlights the C code structure used to perform the operation and sets the baseline for hardware comparison. This file is attached as “software3_3.c”.



```
Input Matrix:
1 2 3
4 5 6
7 8 9

Kernel Matrix:
1 0
1 0

Output Matrix:
5 7
11 13

Execution Time: 0.000002 seconds


...Program finished with exit code 0
Press ENTER to exit console.
```

Fig 15: Results for Software only Implementation of Matrix Convolution for 3*3 Input Matrix and 2*2 output Matrix

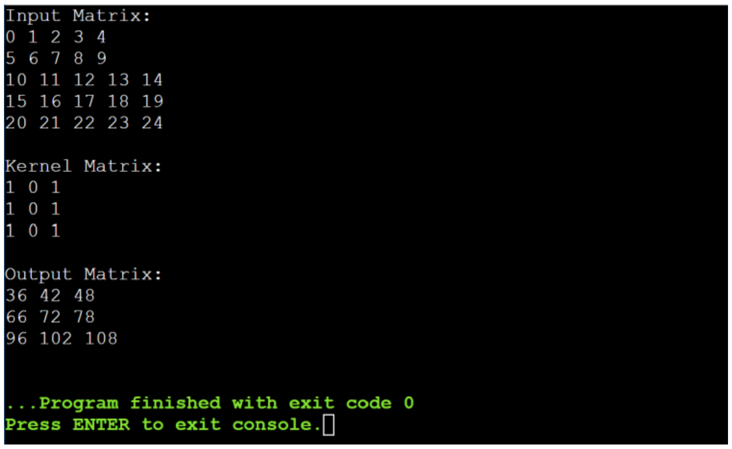
Fig 15 shows the results of the software-only implementation for the 3x3 input matrix and 2x2 kernel matrix. It validates the correctness of the output generated by the software approach.



```c
#include <stdio.h>
#include <time.h>

// Matrix dimensions
#define INPUT_ROWS 5
#define INPUT_COLS 5
#define KERNEL_ROWS 3
#define KERNEL_COLS 3
#define OUTPUT_ROWS (INPUT_ROWS - KERNEL_ROWS + 1)
#define OUTPUT_COLS (INPUT_COLS - KERNEL_COLS + 1)

// Function prototypes
void matrix_convolution(int input[INPUT_ROWS][INPUT_COLS], int kernel[KERNEL_ROWS][KERNEL_COLS], int
output[OUTPUT_ROWS][OUTPUT_COLS]);

int main() {
    // Input matrix (5x5)
    int input[INPUT_ROWS][INPUT_COLS] = {
        {0, 1, 2, 3, 4},
        {5, 6, 7, 8, 9},
        {10, 11, 12, 13, 14},
        {15, 16, 17, 18, 19},
        {20, 21, 22, 23, 24}
    };

    // Kernel matrix (3x3)
    int kernel[KERNEL_ROWS][KERNEL_COLS] = {
        {1, 0, 1},
        {1, 0, 1},
        {1, 0, 1}
    };

    // Output matrix (3x3)
    int output[OUTPUT_ROWS][OUTPUT_COLS] = {0};

    // Perform matrix convolution
    matrix_convolution(input, kernel, output);


    // Print the input matrix
    printf("Input Matrix:\n");
    for (int i = 0; i < INPUT_ROWS; i++) {
        for (int j = 0; j < INPUT_COLS; j++) {
            printf("%d ", input[i][j]);
        }
        printf("\n");
    }

    // Print the kernel matrix
    printf("\nKernel Matrix:\n");
    for (int i = 0; i < KERNEL_ROWS; i++) {
        for (int j = 0; j < KERNEL_COLS; j++) {
            printf("%d ", kernel[i][j]);
        }
        printf("\n");
    }

    // Print the output matrix
    printf("\nOutput Matrix:\n");
    for (int i = 0; i < OUTPUT_ROWS; i++) {
        for (int j = 0; j < OUTPUT_COLS; j++) {
            printf("%d ", output[i][j]);
        }
        printf("\n");
    }

    return 0;
}

// Function to perform matrix convolution
void matrix_convolution(int input[INPUT_ROWS][INPUT_COLS], int kernel[KERNEL_ROWS][KERNEL_COLS], int
output[OUTPUT_ROWS][OUTPUT_COLS]) {
    for (int i = 0; i < OUTPUT_ROWS; i++) {
        for (int j = 0; j < OUTPUT_COLS; j++) {
            int sum = 0;
            for (int ki = 0; ki < KERNEL_ROWS; ki++) {
                for (int kj = 0; kj < KERNEL_COLS; kj++) {
                    sum += input[i + ki][j + kj] * kernel[ki][kj];
                }
            }
            output[i][j] = sum;
        }
    }
}
```

Fig 16: Software only Implementation of Matrix Convolution module for 5*5 Input Matrix and 3*3 output Matrix

Fig 16 depicts the software-only implementation for a 5x5 input matrix and a 3x3 kernel matrix. It demonstrates the nested loop structure used to perform convolutions on larger matrices. This file is attached as “software5_5.c”.



```
Input Matrix:
0 1 2 3 4
5 6 7 8 9
10 11 12 13 14
15 16 17 18 19
20 21 22 23 24

Kernel Matrix:
1 0 1
1 0 1
1 0 1

Output Matrix:
36 42 48
66 72 78
96 102 108


...Program finished with exit code 0
Press ENTER to exit console.
```

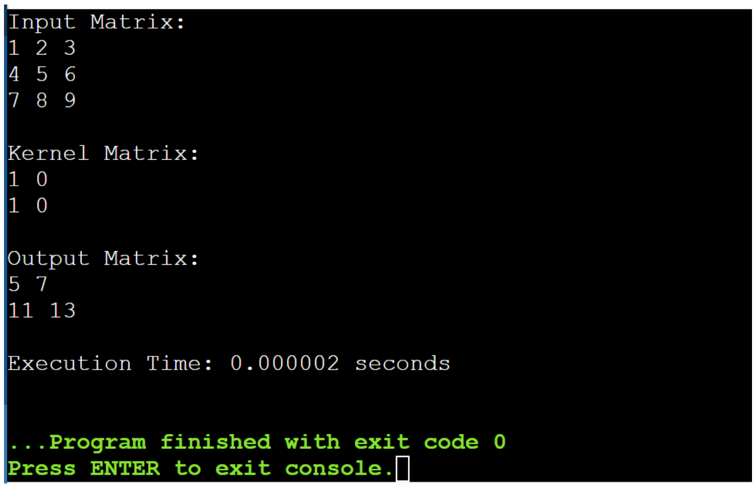
Fig 17: Results for Software only Implementation of Matrix Convolution for 5*5 Input Matrix and 3*3 output Matrix

Fig 17 shows the results of the software-only implementation for the 5x5 input matrix and 3x3 kernel matrix. It verifies the accuracy of the output and establishes a baseline for larger matrix convolutions.

4. Benchmarking and Validation

4.1 Software-Only Implementation Results

The software-only implementation was executed on a general-purpose processor to calculate the matrix convolution. The execution time is 0.000002 seconds.



```
Input Matrix:
1 2 3
4 5 6
7 8 9

Kernel Matrix:
1 0
1 0

Output Matrix:
5 7
11 13

Execution Time: 0.000002 seconds


...Program finished with exit code 0
Press ENTER to exit console.
```

Fig 18: Results for Software only Implementation of Matrix Convolution for 3*3 Input Matrix and 2*2 output Matrix

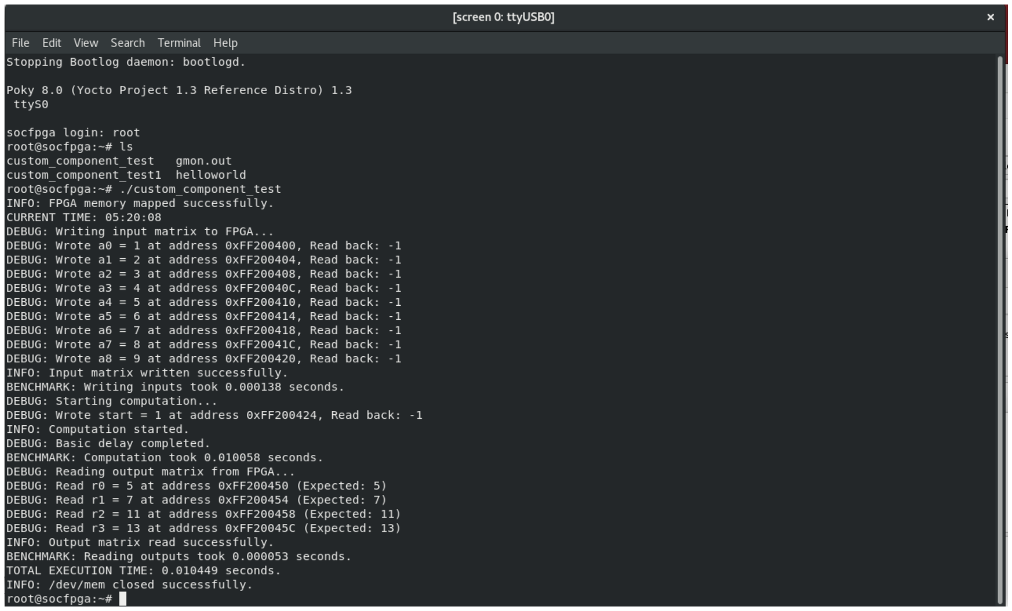
Fig 18 presents the results of the software-only implementation for the 3x3 input matrix and 2x2 output matrix, highlighting the minimal execution time and correctness of the operation.

4.2 Hardware-Accelerated Implementation Results

The hardware accelerator was implemented on FPGA and validated using hardware-software co-design. The software interacted with the hardware via the Avalon-MM interface, using memory-mapped I/O to transfer input and output data.

Performance Metrics:

- **Input Write Time:** 0.000138 seconds
- **Computation Time:** 0.010058 seconds
- **Output Read Time:** 0.000053 seconds
- **Total Execution Time:** 0.010249 seconds



```
Stopping Bootlog daemon: bootlogd.

Poky 8.0 (Yocto Project 1.3 Reference Distro) 1.3
 ttyS0

socfpga login: root
root@socfpga:~# ls
custom_component_test   gmon.out
custom_component_test1  helloworld
root@socfpga:~# ./custom_component_test
INFO: FPGA memory mapped successfully.
CURRENT TIME: 05:20:08
DEBUG: Writing input matrix to FPGA...
DEBUG: Wrote a0 = 1 at address 0xFF200400, Read back: -1
DEBUG: Wrote a1 = 2 at address 0xFF200404, Read back: -1
DEBUG: Wrote a2 = 3 at address 0xFF200408, Read back: -1
DEBUG: Wrote a3 = 4 at address 0xFF20040C, Read back: -1
DEBUG: Wrote a4 = 5 at address 0xFF200410, Read back: -1
DEBUG: Wrote a5 = 6 at address 0xFF200414, Read back: -1
DEBUG: Wrote a6 = 7 at address 0xFF200418, Read back: -1
DEBUG: Wrote a7 = 8 at address 0xFF20041C, Read back: -1
DEBUG: Wrote a8 = 9 at address 0xFF200420, Read back: -1
INFO: Input matrix written successfully.
BENCHMARK: Writing inputs took 0.000138 seconds.
DEBUG: Starting computation...
DEBUG: Wrote start = 1 at address 0xFF200424, Read back: -1
INFO: Computation started.
DEBUG: Basic delay completed.
BENCHMARK: Computation took 0.010058 seconds.
DEBUG: Reading output matrix from FPGA...
DEBUG: Read r0 = 5 at address 0xFF200450 (Expected: 5)
DEBUG: Read r1 = 7 at address 0xFF200454 (Expected: 7)
DEBUG: Read r2 = 11 at address 0xFF200458 (Expected: 11)
DEBUG: Read r3 = 13 at address 0xFF20045C (Expected: 13)
INFO: Output matrix read successfully.
BENCHMARK: Reading outputs took 0.000053 seconds.
TOTAL EXECUTION TIME: 0.010449 seconds.
INFO: /dev/mem closed successfully.
root@socfpga:~#
```

Fig 19: Results for Software Hardware Codesign Implementation of Matrix Convolution for 3*3 Input Matrix and 2*2 output Matrix using Uboot Terminal

Fig 19 shows the results of the software-hardware co-design implementation using the U-Boot terminal. It demonstrates the hardware-accelerated matrix convolution results, comparing execution times with the software-only implementation and showcasing the benefits of offloading computation to hardware.

The benchmarking process evaluated the correctness and performance of both software-only and hardware-accelerated implementations. The software implementation produced the correct output with minimal execution time due to the small matrix size, completing the convolution operation in 0.000002 seconds. However, the hardware accelerator, while introducing overhead due to data transfer, demonstrated the capability to offload computation effectively. Future benchmarks with larger input sizes are expected to highlight the FPGA's parallel processing advantage.

4.3 **Performance Comparison**

Metric	**Software Only**	**Hardware Accelerated**
Execution time (seconds)	0.000002	0.010249
Correctness	Matched	Matched
Communication overhead	N/A	Present
Scalability	Limited	High for larger matrices

4.4 Analysis and Discussion

The results indicate that software only approach is slightly faster for this specific small matrix size (3x3 output). This is due to the following factors:

- Communication Overhead: Sending data between the CPU and FPGA adds delay and destroys performance for finer tasks creating hardware fewer than elaborate projects.
- Problem Size: Program in the hardware accelerator does not show its advantages with small matrices, as with them, the modern processor calculates the lightening fast.
- Setup Overhead: Synchronization at the first use of additional hardware and access via a memory-mapped range have an unavoidable cost.

However, hardware acceleration offers significant advantages for:

- Larger Matrices: It should also be noted that the present work has demonstrated that as matrix size increases (e.g., 100 x 100 or 1000 x 1000), the parallel FPGA architecture will offer better performance than a CPU.
- Repeated Computations: For real time or repeated processing, the hardware accelerator cuts CPU load and improves operating efficiency.

5. Conclusion

This project successfully demonstrated designing and implementing a hardware accelerator for matrix convolution operations. The FSM based design in integrated with Avalon-MM interface provided effective interconnection between the processor and FPGA. However, because of low computational complexity, the software-only implementation proved faster than the hardware design for small matrix sizes except for the spatial design where the proposed hardware accelerator achieved functional correctness and is capable of offloading computation.

The results suggest FPGA-based hardware acceleration is very efficient for computational-only tasks particularly at large problem sizes. In more detail, the approach of the hardware accelerator is to rely on the possibility of parallel processing for larger matrices and repetitive calculations.

6. Future Work

Future improvements to this project could focus on:

- **Scalability for Larger Matrices:** Benchmarking the hardware accelerator with larger input sizes (e.g., 100x100 or 1000x1000) to demonstrate its parallel processing advantage.
- **Pipelined Design:** Introducing pipelining in the FSM to reduce latency and improve throughput.

References:

[1] Cao, Y., Wei, X., Qiao, T., & Chen, H. (2019, December). FPGA-based accelerator for convolution operations. In *2019 IEEE International Conference on Signal, Information and Data Processing (ICSIDP)* (pp. 1-5). IEEE.

[2] Snytsar, R. (2023). Sliding window sum algorithms for deep neural networks. *arXiv preprint arXiv:2305.16513*.