STMicroelectronics

Systolic Array for Applying Matrix Multiplication

Muhammed Adel Ahmed El-Sayed

Table of Contents

Table of Snappets

Table of Figures

Introduction

In this document, I will show the architecture and operation of a systolic array designed for matrix multiplication. Systolic arrays are specialized parallel computing architectures that excel at computationally intensive tasks like matrix operations by employing a network of interconnected processing elements (PEs) that process data in a pipelined fashion. This design leverages the inherent parallelism of matrix multiplication to achieve high throughput and efficiency. As shown in gif 1, an abstract overview about the operation of systolic array.

Design Implementation

Processing Element (PE) Design

Processing Element (PE), a fundamental building block responsible for performing the multiply-accumulate operation. Each PE is designed to handle a single element multiplication and accumulate the result, while also passing data to its neighboring PEs. As shown in Fig the top-level PE design indicating the interface I/O ports.

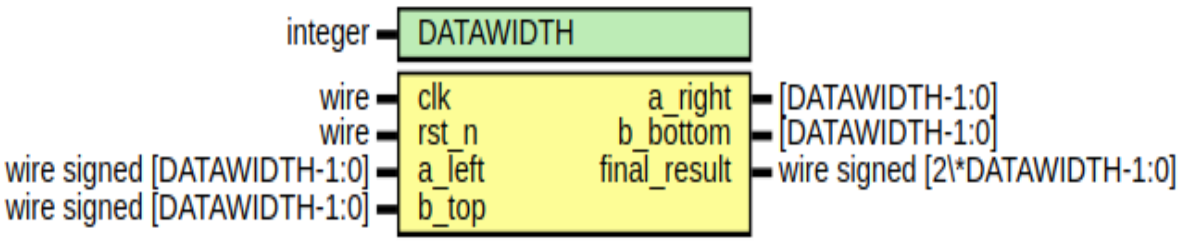


Figure 1 :Top-level PE

PE Inputs and Outputs

The following table indicate the interface ports with an abstract description of its operation.

Signal	Direction	Width	Description
clk	Input	1 bit	Global clock signal that synchronizes all operations within the PE.
rst_n	Input	1 bit	Active-low asynchronous reset. When asserted, initializes the PE's internal registers to zero.
a_left	Input	DATAWIDTH bits	Signed input from the PE on the left; carries an element of matrix A.
b_top	Input	DATAWIDTH bits	Signed input from the PE above; carries an element of matrix B.
a_right	Output	DATAWIDTH bits	Signed output to the PE on the right; forwards a_left after a one-clock delay.
b_bottom	Output	DATAWIDTH bits	Signed output to the PE below; forwards b_top after a one-clock delay.
final_result	Output	2 × DATAWIDTH bits	Signed accumulated partial-sum (product) for this PE, contributing to the final C matrix element.

Table 1: PE I/O Ports

As shown in snippet code 1, the interface ports parametrized with DATAWIDTH 16-bit.

```
// Systolic Processing Element (PE)
// This module performs multiply-accumulate operations and forwards data to
neighbors.
module systolic_pe #(
    parameter integer DATAWIDTH = 16  // Width of input operands
)(
    input  wire                          clk,        // Clock signal
    input  wire                          rst_n,      // Active-low reset
    input  wire signed [DATAWIDTH-1:0] a_left,     // Input data from left neighbor
    input  wire signed [DATAWIDTH-1:0] b_top,      // Input data from top neighbor
    output reg  signed [DATAWIDTH-1:0] a_right,    // Output data to right neighbor
    output reg  signed [DATAWIDTH-1:0] b_bottom,   // Output data to bottom neighbor
    output wire signed [2*DATAWIDTH-1:0] final_result  // Output accumulated result
);
```

Snippet 1: PE I/O interface

PE Internal Operation

As shown in snippet code , the primary internal component of the PE is the accumulator register, which is double wide to prevent overflow during the multiplication and accumulation.

The operation of the PE is governed by a synchronous always_ff.

When assert (rst_n), all outputs (a_right , b_bottom) and the cleared to zero thus the PE zero.

In normal operation (when accumulator is rst_n is high), the PE performs two main functions:

1. Data Forwarding: The a_left input is registered and the b_top input is registered then passed to a_right , and b_bottom . This ensures that data propagates, maintaining synchronization across all PEs.

2. Multiply-Accumulate (MAC) Operation: This involve multiplying the current a_left and b_top inputs and adding the product to the accumulator . As shown in the snippet code. This operation is fundamental to matrix multiplication.

```
    // Internal accumulator for partial sums (size double DATAWIDTH)
    reg signed [2*DATAWIDTH-1:0] accumulator;
    // Connect internal accumulator to final output
    assign final_result = accumulator;
    // Sequential logic: on clock edge or reset
    always_ff @(posedge clk or negedge rst_n) begin
        if (!rst_n) begin
            // On reset, clear outputs and accumulator
            a_right <= 0;
            b_bottom <= 0;
            accumulator <= 0;
        end else begin
            // Forward input data to neighboring PEs
            a_right <= a_left;
            b_bottom <= b_top;
            // Perform multiply-accumulate: update dot-product term
            accumulator <= accumulator + (a_left * b_top);
        end
    end
```

Snippet 2: PE internal operation

The final_result output is assigned the value of the accumulator. The design implies that the accumulation continues over multiple clock cycles as new a_left and b_top values arrive.

The schematic shown in fig to show the architecture detailed component.

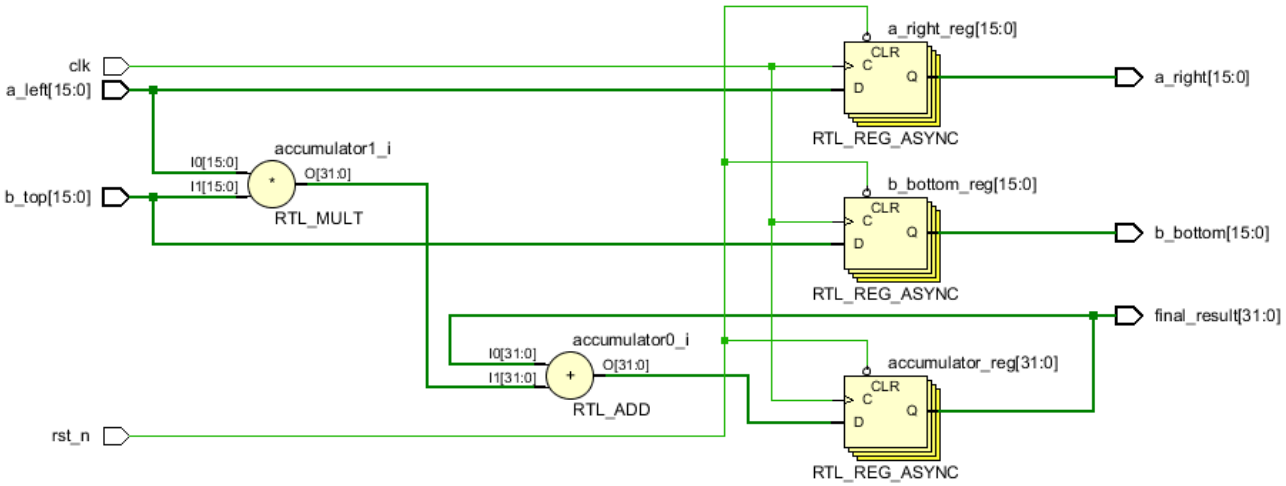


Figure 2: Hardware structure of PE

As shown in fig, the Architecture diagram shows the architecture of the implementation of PEs.

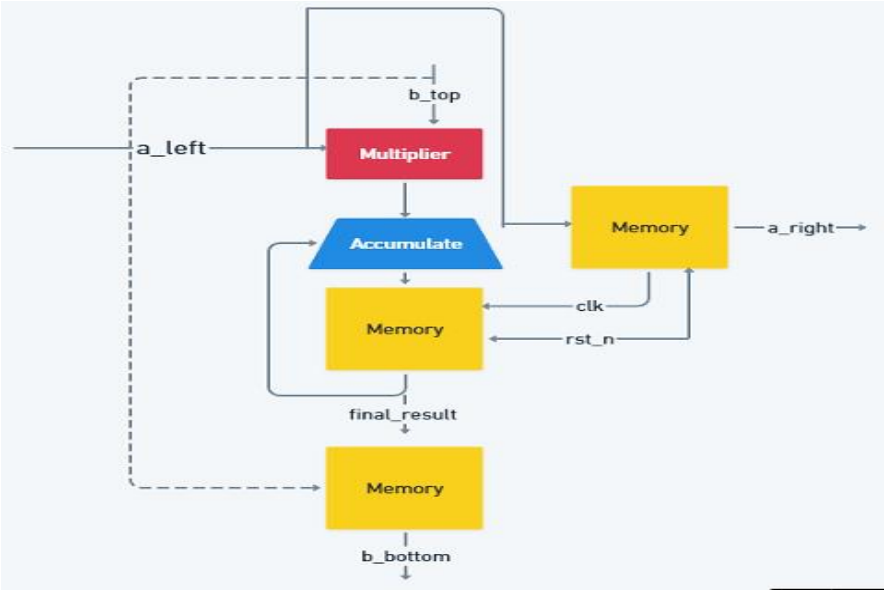


Figure 3:Block Diagram of a PE

Systolic Array Module Design

The systolic_array orchestrates the interaction of multiple systolic_pe instances to perform matrix multiplication. It handles the input distribution of matrices A and B, manages the data flow through the array, and collects the final results to form matrix C. The given fig shows the top-level of the design indicating the I/O ports interface.

Diagram

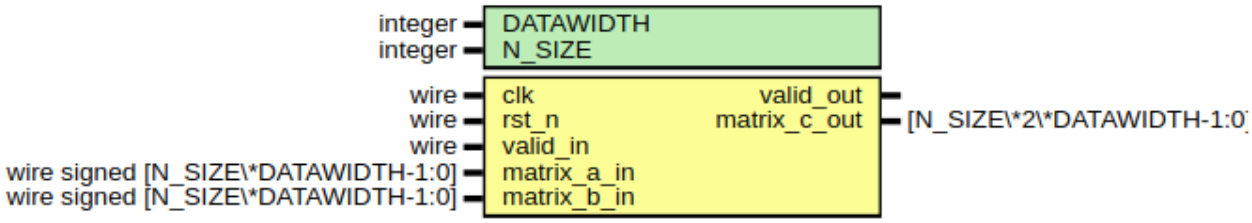
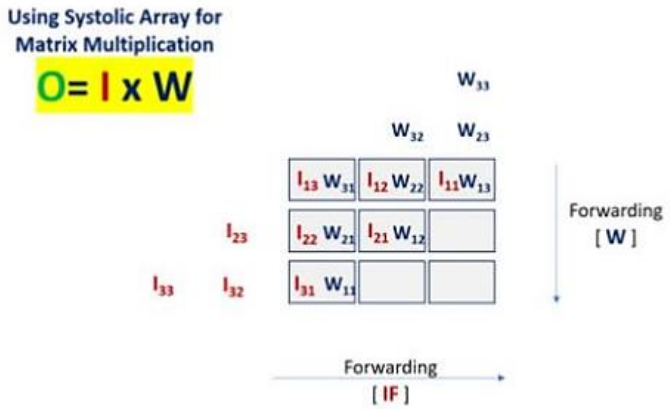


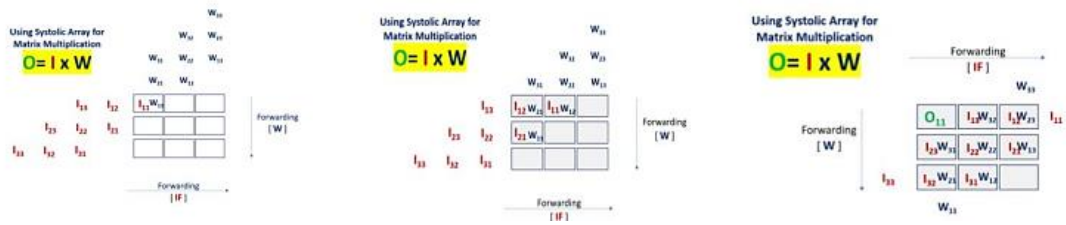
Figure 4: SA Top-Level

Systolic Data Flow Principles

1. **Talk Only to Your Neighbors:** Each unit sends and receives data just from the ones right beside it.
2. **Clock-Driven Data:** Every tick, data moves one step through the grid like a pulse.
3. **Continuous Pipelining:** New data enters each cycle while old data keeps flowing out—no idle time.
4. **Combine as You Go:** Partial results pass along and get added up without big memories.

Figure 5: SA Data Flow





Systolic Array Interface and internal buses

The following table indicate the interface ports with an abstract description of its operation.

Signal	Direction	Width	Description
clk	Input	1 bit	Global clock signal.
rst_n	Input	1 bit	Global active-low asynchronous reset signal.
valid_in	Input	1 bit	Control signal indicating that new input data for matrix_b_in is available this cycle.
matrix_a_in	Input	N_SIZE × DATAWIDTH bits	Carries a row or column of matrix A (distribution depends on your tiling strategy).
matrix_b_in	Input	N_SIZE × DATAWIDTH bits	Carries a row or column of matrix B.
valid_out	Output	1 bit	Pulses high for one cycle when the entire array has finished computing and matrix_c_out holds a valid result.
matrix_c_out	Output	N_SIZE × 2 × DATAWIDTH bits	Carries the resulting matrix C; the 2 × DATAWIDTH factor accounts for bit-width growth due to accumulation in the PEs.

Table 2: SA I/O Ports

As shown in snippet code , the interface ports parametrized with DATAWIDTH 16-bit and N_SIZE for matrixes dimension.

```
module systolic_array #(
    parameter integer DATAWIDTH = 16, // Bit-width for each PE
    parameter integer N_SIZE    = 5   // Matrix dimension (N x N)
)(
    input  wire                             clk,            // Clock signal
    input  wire                             rst_n,          // Active-low reset
    // Input: new row/column injected when valid_in is high
    input  wire                            valid_in,
    input  wire signed [N_SIZE*DATAWIDTH-1:0] matrix_a_in,
    input  wire signed [N_SIZE*DATAWIDTH-1:0] matrix_b_in,
    output logic                            valid_out,
    output logic signed [N_SIZE*2*DATAWIDTH-1:0] matrix_c_out
);
```

Snippet 3: SA Interface

The design utilizes several internal wires and registers to manage data flow:

- $a\_bus[0:N\_SIZE][0:N\_SIZE-1]$ → 2D array of wires carrying matrix A elements horizontally between PEs, As shown in fig.
- $b\_bus[0:N\_SIZE-1][0:N\_SIZE]:$ → 2D array of wires carrying matrix B elements vertically between PEs, As shown in fig.

Snippet 5: Connections between buses and the wires which hold input matrixes

```
    genvar i, j;
    generate
        for (i = 0; i < N_SIZE; i = i + 1) begin : INPUT_CONNECTIONS
            // Left boundary feeds A values into first column
            assign a_bus[i][0] = valid_in ? a_slice[i] : '0;
            // Top boundary feeds B values into first row
            assign b_bus[0][i] = valid_in ? b_slice[i] : '0;
        end
    endgenerate
```

- $result\_bus[0:N\_SIZE-1][0:N\_SIZE-1]$ → 2D array of wires carrying the final_result from each PE. These results are eventually combined to form matrix_c_out .
- $a\_input\_regs\ and\ b\_input\_regs$ → Registers used to temporarily hold and distribute input data to the first row and column of PEs.
- $delay$ → A counter used to manage the timing and synchronization of input data distribution, ensuring that data arrives at the correct PE at the correct time for the systolic operation.
- $a\_slice\ and\ b\_slice$ → Helper wires used for bit-slicing the wide and matrix_b_in buses into individual matrix_a_in DATAWIDTH-bit elements for distribution to the PEs.

```
    // Generate bit slicing for input matrices
    genvar k;
    generate
        for (k = 0; k < N_SIZE; k = k + 1) begin : SLICE_GEN
            // Extract each DATAWIDTH-bit element from flattened inputs
            assign a_slice[k] = matrix_a_in[(k+1)*DATAWIDTH-1:k*DATAWIDTH];
            assign b_slice[k] = matrix_b_in[(k+1)*DATAWIDTH-1:k*DATAWIDTH];
        end
    endgenerate
```

Snippet 4: The slices wires that hold the inputs

Systolic PE Array Generation

As shown in snippet code , the core of the systolic_array module is a nested N_SIZE by N_SIZE that instantiates PEs modules. Each PE (pe_inst) is connected to its neighbors using the a_bus and b_bus wires, forming a grid. The at (i, j) is connected to a_right of b_bottom of a_left input of a PE (i, j-1), and b_top is connected to (i-1, j) .

```
    generate
        for (i = 0; i < N_SIZE; i = i+1) begin : ROWS
            for (j = 0; j < N_SIZE; j = j+1) begin : COLS
                systolic_pe #(
                    .DATAWIDTH(DATAWIDTH)
                ) pe_inst (
                    .clk         (clk),
                    .rst_n       (rst_n),
                    .a_left      (a_bus[i][j]),     // Input from left
                    .b_top       (b_bus[i][j]),     // Input from top
                    .a_right     (a_bus[i][j+1]),   // Output to right
                    .b_bottom    (b_bus[i+1][j]),   // Output to bottom
                    .final_result(result_bus[i][j]) // Partial dot-product
                );
            end   end  endgenerate
```

Snippet 6: Generation of PEs inside the SA

As shown in fig, this clarifies the connection between PEs through a_bus and b_bus thus this creates the characteristic systolic data flow.

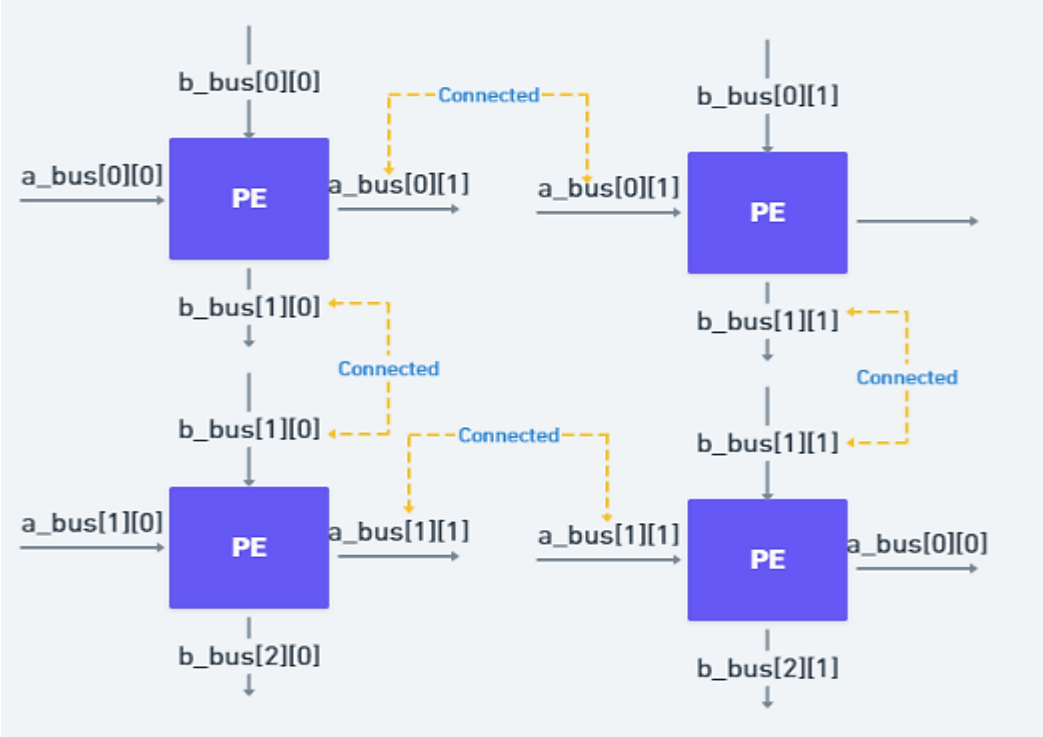


Figure 6: An explanation of the connections happens during the generation of PEs

Input Distribution Logic

This logic is responsible for feeding the matrix_a_in and matrix_b_in data into the array in a staggered fashion, which is critical for the correct operation of a systolic array for matrix multiplication.
During the initial cycles (rst_n is asserted), I initialize the registers that used to temporarily hold and distribute input data to the first row and column of PEs, initialize the timing counter delay which make the first element of matrix_a_in enter the PEs and delay the other elements and so on as shown in fig and initialize the counter c which will enable the valid_out after N_SIZE +1 thus a valid output you can take is available.

```
// Main control for distributing inputs and generating valid_out
always @(posedge clk or negedge rst_n) begin
    if (!rst_n) begin // Initialize registers and counters on reset
        for (integer i = 0; i < N_SIZE; i = i + 1) begin
            a_input_regs[i] <= 0;
            b_input_regs[i] <= 0;
        end
        valid_d <= 0;
        delay <= 1;
        c <= N_SIZE+1;
    end
```

Snippet 7: The initialization of signals during negedge of rst_n

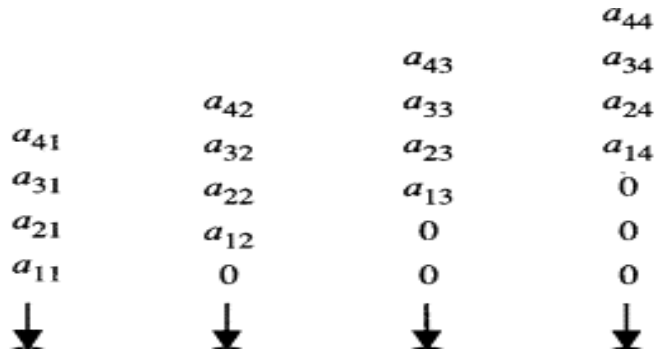


Figure 7: An explanation of how inputs enter the PE

When valid_in is asserted, the module processes the incoming matrix_a_in and matrix_b_in by slicing them into individual elements and directing them to the appropriate a_input_regs and b_input_regs based on the current delay count so this ensures that the data is correctly aligned for the PEs. ($controlled\ by\ delay\ <=\ N\_SIZE$), elements are loaded into the PEs along the anti-diagonal at which the elements in the column loaded one by one each cycle meaning that at first cycle in 3x3 matiexs let say 1 0 0 and delay = 1, in the next cycle 1 1 0 delay =2, third cycle 1 1 1 delay = 3 so the next cycle we expected to fill the zeros before the elements so let say 0 1 1 after that 0 0 1 then 0 0 0, as shown in code snippet.

Shifting Pattern: In subsequent cycles (controlled by $delay\ <=\ 2 * N\_SIZE - 1$), the input data shifts, allowing new elements to enter the array and propagate through the PEs. The shift_offset variable helps in correctly indexing the a_slice and b_slice to feed the PEs, as shown in fig that illustrate that when we enter 4x4 matric so we expect that we will shift by 7 cycles.

```
            // Phase 1: load diagonal elements in cycles 1 to N_SIZE
            if (delay <= N_SIZE) begin
                for (integer i = 0; i < delay; i = i + 1) begin
                    a_input_regs[i] <= a_slice[i];
                    b_input_regs[i] <= b_slice[i];
                end
                valid_d <= 0; // No valid output yet
                c <= N_SIZE+1;
            end
            // Phase 2: shift inputs in cycles N_SIZE+1 to 2*N_SIZE-1
            else if (delay <= 2*N_SIZE-1) begin
                shift_offset = delay - N_SIZE;
                for (integer i = shift_offset; i < N_SIZE; i = i + 1) begin
                    a_input_regs[i] <= a_slice[i-shift_offset];
                    b_input_regs[i] <= b_slice[i-shift_offset];
                end
            end
```

Snippet 8: Shifting Pattern

As shown in snippet code, when a new data arrives so clear all input temporally regs as to hold the new data from the input's matrixes.

```
else if (valid_in) begin
            // When new data arrives, clear input regs first
            for (integer i = 0; i < N_SIZE; i = i + 1) begin
                a_input_regs[i] <= '0;
                b_input_regs[i] <= '0;
            end
```

Snippet 9: The initialization when new input data

When the counter delay reach N_SIZE +1 so we expect that we finish inputting the two matrix and its time to disenable the valid_in to enable the valid_out so we can get the output row by row by each cycle in the same time decreasing c counter as after reach the final row we disenable the output as the design finish its computation at this time we initialize the counter again.

```
        else if (delay == N_SIZE+1) begin
            // After loading, generate valid pulses and track rows
            if (c > 1) begin
                c <= c-1;
                valid_d <= 1; // Pulse valid while draining results
            end
            else begin
                // Reset for next operation
                delay <= 1;
                valid_d <= 0;
                c <= N_SIZE+1;
            end
```

Snippet 10: The snippet that indicate what happen when we finish the inputs

The result_bus collects the results, final_result results from each PE. An always_comb block is used to combine these individual PE results into the output matrix_c_out bus when valid_out is asserted. The valid_out signal enabled after the entire array has completed its computation, indicating that, valid result of the matrix multiplication.

```
// Assemble final output matrix when valid_out is high
always_comb begin
    if (valid_out) begin
        for (integer s = 0; s < N_SIZE; s = s + 1) begin
            matrix_c_out[s*2*DATAWIDTH +: 2*DATAWIDTH] = result_bus[N_SIZE-c][s];
        end
    end
    else begin
        matrix_c_out = '0;  // Default output when not valid
    end
end
```

Snippet 11: The assignment of matric_c_out

Systolic Matrix Multiplication Example (N= 3)

I'll illustrate with a 3x3 matrix multiplication. The input data streams for A and B are fed into the array in a staggered manner to ensure that the correct elements arrive at the correct PEs.
Consider the elements aij from matrix A and bij from matrix B. The PE at position (i, j) will compute the sum of products that contributes to Cij.

- Initial Phase: The first elements (e.g., a00 and b00) enter the top-left PE (0,0). They are multiplied, and the product is added to the accumulator of PE(0 ,0).
- Propagation: In the next clock cycle, a move to PE(0 ,1) and b moves to PE(1 ,0). Simultaneously, new elements (e.g., a01 and b10) enter PE(0,0) and PE(1,0) respectively, and a10 and b01 enter PE(0,1) and PE(1 ,0).
- Staggered Input: The valid_in signal and the internal that the inputs matrix_a_in and delay counter ensure matrix_b_in are sliced and fed into the array such that elements that need to interact arrive at the same PE at the same time.
- The final_result from each PE, the results for matrix C will emerge from the PEs in a staggered fashion. The valid_out signal indicates when the complete matrix_c_out is available.

Architecture Diagram

As shown in fig, an abstraction block diagram that show:

- Components in the design.
- Output signals.
- Input signals.
- Abstracted Data Flow.
- PE structure component.

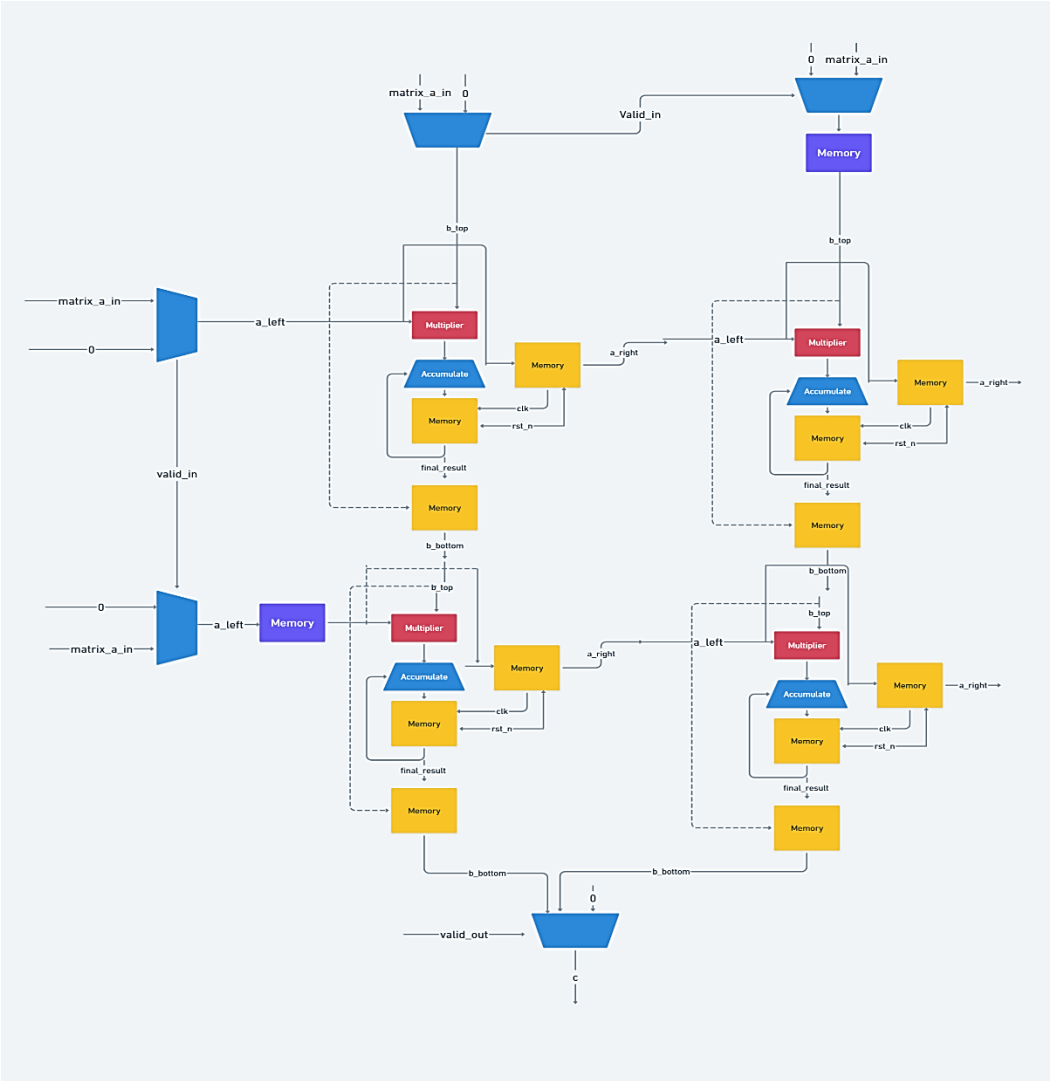


Figure 8 :Block diagram of simple 2x2 Systolic Array

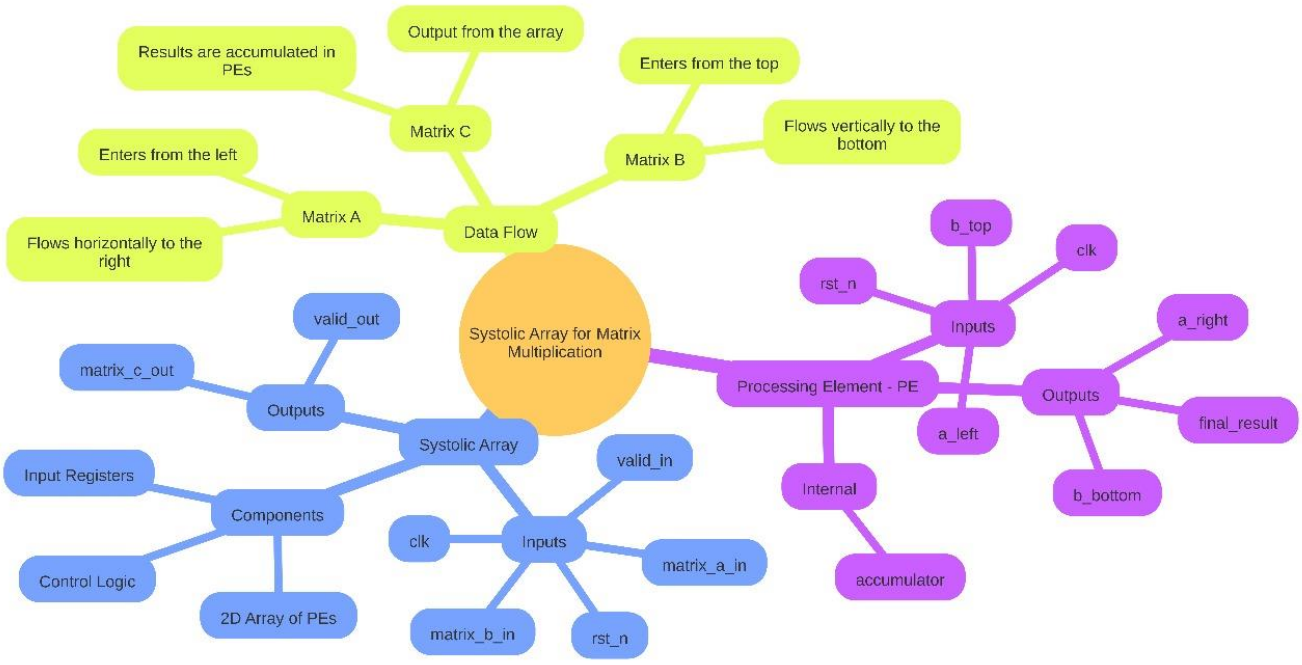


Figure 9: An abstraction of everything about the design

Design Verification

Verification Objectives

- Verify that the systolic array performs accurate matrix multiplication for all supported matrix sizes (2x2, 3x3, 4x4, 5x5, .etc.) based on N_SIZE, Confirming that the design scales correctly for different N_SIZE parameters.
- Ensure proper data flow synchronization across all processing elements (PEs) in the array.
- Validate correct handshaking between input/output interfaces and internal processing units.

Verification Methodology

Directed Testing : Specific test cases targeting known corner cases and boundary conditions

```
// Generic task to initialize test matrices
    task automatic init_test_matrices;
        integer i, j;
        begin
            // Matrix A: Sequential values starting from 1
            for (i = 0; i < N_SIZE; i = i + 1) begin
                for (j = 0; j < N_SIZE; j = j + 1) begin
                    matrix_a[i][j] = i * N_SIZE + j + 1;
                end
            end

            // Matrix B: Reverse sequential values
            for (i = 0; i < N_SIZE; i = i + 1) begin
                for (j = 0; j < N_SIZE; j = j + 1) begin
                    matrix_b[i][j] = N_SIZE * N_SIZE - (i * N_SIZE + j);
                end
            end
        end
    endtask
```

Snippet 12: Initialization Test

Test Cases

1. 2x2 Matrices

As shown in fig , when the user has to apply the inputs matrixes then the valid_in will be enabled so at each posedge of cycles the inputs will entering.
When the inputs are finished so the valid_in will dis enabled

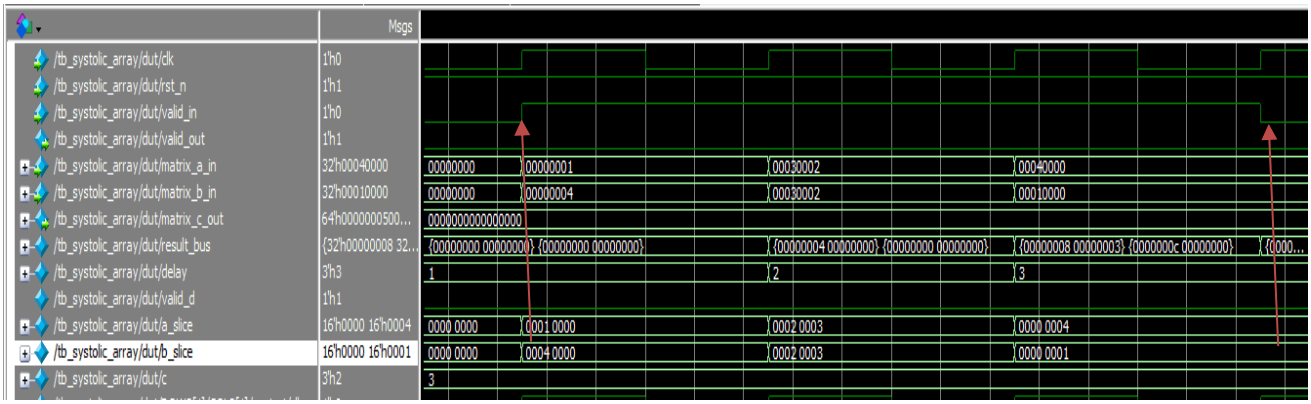
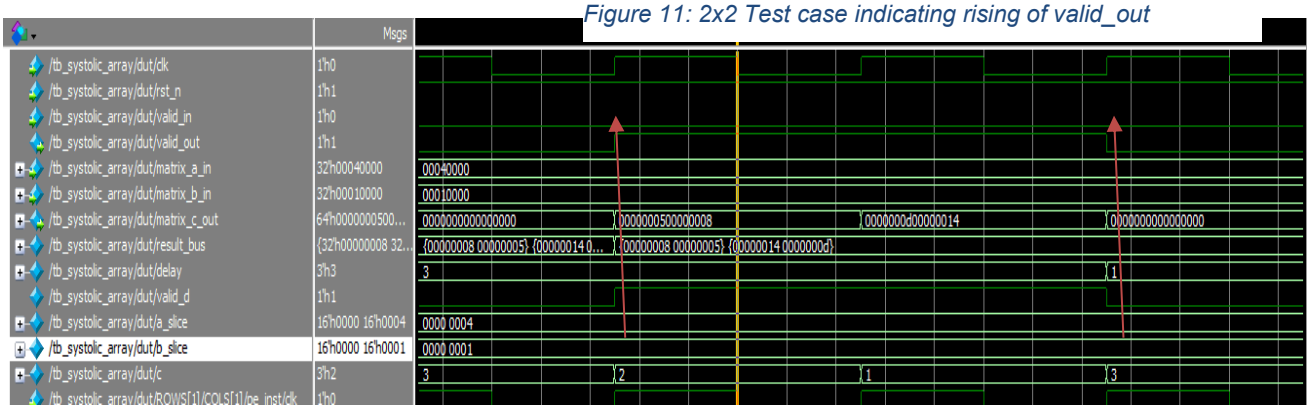


Figure 10: 2x2 Test case indicating the enable of valid_in and data entry flow

As shown in fig , when the inputs are finished the valid_out will be enabled so that the design will output matrix c row by row at each posedge then when the output finished its valid output multiplied then the valid_out will dis enabled again.

Figure 11: 2x2 Test case indicating rising of valid_out



The Output Results as shown in the log file indicating the inputs matrixes and the expected and actual results so correct output results.

```
# --- Test Case: 2x2 Matrix Multiplication ---
# Matrix A:
#      1    2
#      3    4
# Matrix B:
#      4    3
#      2    1
# Expected Result (A * B):
#          8       5
#         20      13
#
# Collecting matrix outputs...
# Row 0 collected: 0000000500000008
# Row 1 collected: 0000000d00000014
# All 2 rows collected successfully!
# === Output Results ===
# Collected Matrix Rows (Raw Hex):
#   Row 0: 0000000500000008
#   Row 1: 0000000d00000014
# Actual Result Matrix:
#          8       5
#         20      13
# SUCCESS: All results match expected values!
```

Snippet 13: Log of 2x2

2. 3x3 Matrices

As shown in fig , when the user has to apply the inputs matrixes then the valid_in will be enabled so at each posedge of cycles the inputs will entering.

When the inputs are finished so the valid_in will dis enabled

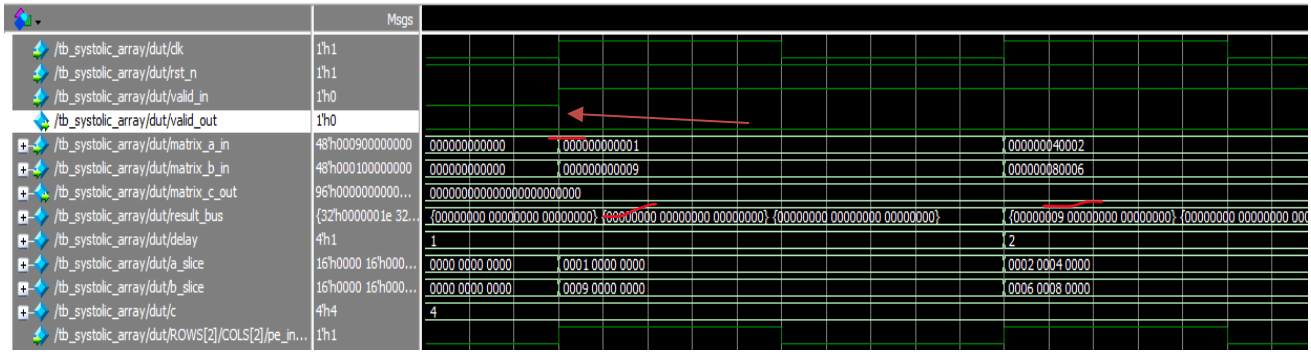


Figure 12: 3x3 Test case indicating the enable of valid_in and data entry flow

As shown in fig, the inputs keep entering while valid_in enabled and valid_out disenabled.

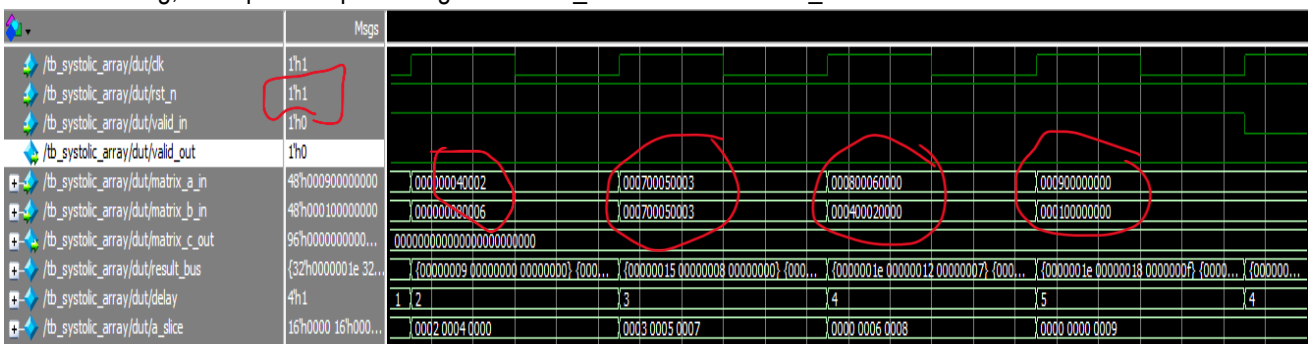


Figure 13: 3x3 Test case indicating rising of valid_out

As shown in fig , when the inputs are finished the valid_out will be enabled so that the design will output matrix c row by row at each posedge then when the output finished its valid output multiplied then the valid_out will dis enabled again.

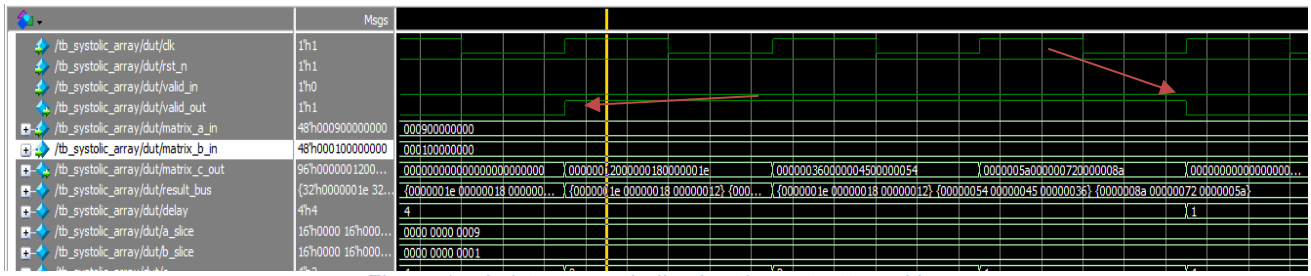


Figure 14: 3x3 test case indicating the output transition

The rows are got in hex in the fig:

- 0000001200000018000000 1e → 30 24 18
- 0000003600000045000000 54→ 84 69 54
- 0000005a000000720000008a→ 138 114 90

Which are the expected results.
The Output Results as shown in the log file indicating the inputs matrixes and the expected and actual results so correct output results.

```
# --- Test Case: 3x3 Matrix Multiplication ---
# Matrix A:
#     1   2   3
#     4   5   6
#     7   8   9
# Matrix B:
#     9   8   7
#     6   5   4
#     3   2   1
# Expected Result (A * B):
#       30      24      18
#       84      69      54
#      138     114      90
# === Output Results ===
# Actual Result Matrix:
#       30      24      18
#       84      69      54
#      138     114      90
# --- Verification ---
# SUCCESS: All results match expected values!
```

Snippet 14: 3x3 Log

3. 4x4 Matrices

As shown in fig , when the user has to apply the inputs matrixes then the valid_in will be enabled so at each posedge of cycles the inputs will entering.

When the inputs are finished so the valid_in will dis enabled

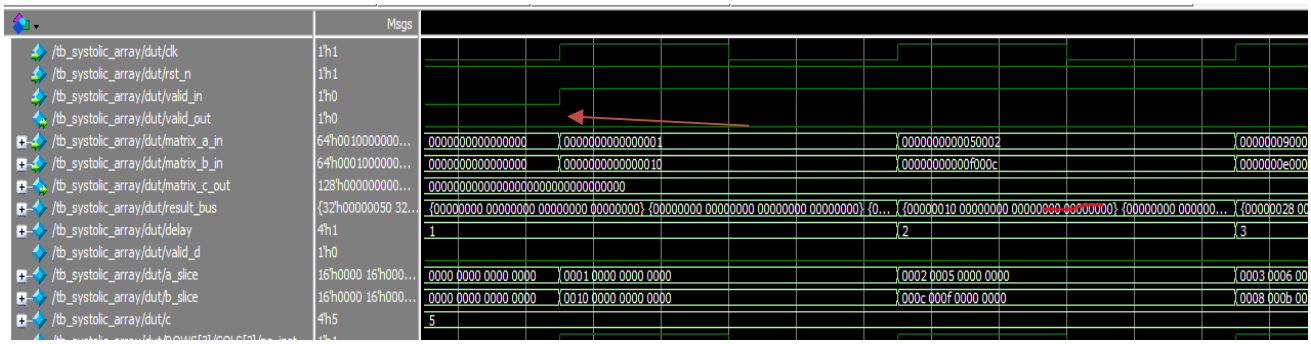


Figure 15:4x4 Test case indicating the enable of valid_in and data entry flow

As shown in fig, the inputs keep entering while valid_in enabled and valid_out disenabled.

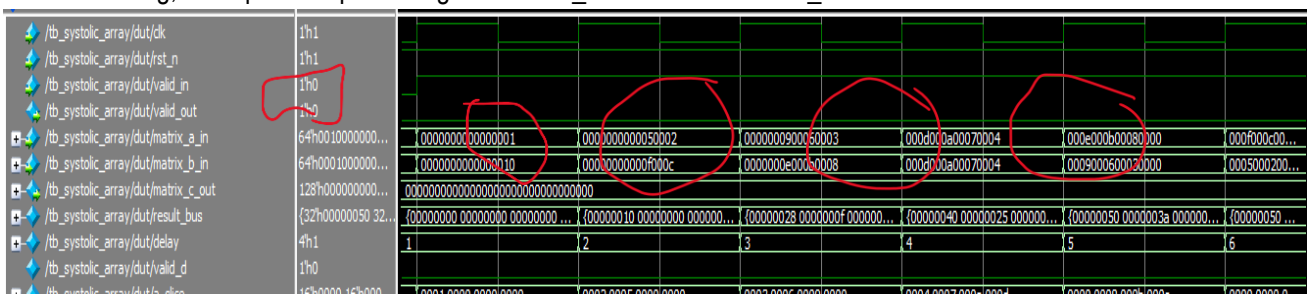


Figure 16: 4x4 Test case indicating data entry flow

As shown in fig , when the inputs are finished the valid_out will be enabled so that the design will output matrix c row by row at each posedge then when the output finished its valid output multiplied then the valid_out will dis enabled again.

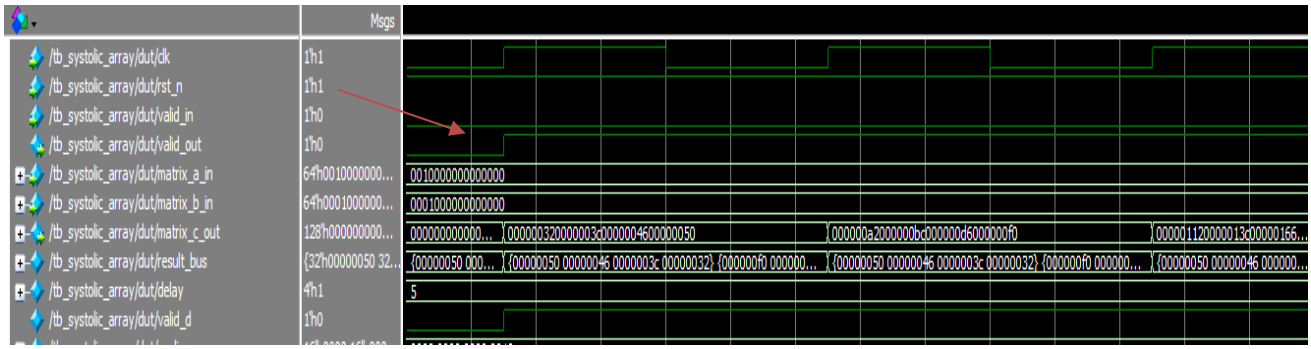


Figure 17: 4x4 test case indicating the output transition

The rows are got in hex in the fig:

- 000000320000003c0000004600000050 → 80 70 60 50
- 000000a2000000bc000000d6000000f0 → 240 214 188 162
- 000001120000013c0000016600000190 → 400 358 316 274
- 00000182000001bc000001f600000230 → 560 502 444 386

Which are the expected results. The Output Results as shown in the log file indicating the inputs matrixes and the expected and actual results so correct output results.

```
# --- Test Case: 4x4 Matrix Multiplication ---
# Matrix A:
#     1   2   3   4
#     5   6   7   8
#     9  10  11  12
#    13  14  15  16
# Matrix B:
#    16  15  14  13
#    12  11  10   9
#     8   7   6   5
#     4   3   2   1
# Expected Result (A * B):
#        80      70      60      50
#       240     214     188     162
#       400     358     316     274
#       560     502     444     386
# Collecting matrix outputs...
# Row 0 collected: 000000320000003c0000004600000050
# Row 1 collected: 000000a2000000bc000000d6000000f0
# Row 2 collected: 000001120000013c0000016600000190
# Row 3 collected: 00000182000001bc000001f600000230
# All 4 rows collected successfully!
# === Output Results ===
# Actual Result Matrix:
#        80      70      60      50
#       240     214     188     162
#       400     358     316     274
#       560     502     444     386
# --- Verification ---
# SUCCESS: All results match expected values!
```

Snippet 15: 4x4 Log file

4. 5x5 Matrices

As shown in fig , when the user has to apply the inputs matrixes then the valid_in will be enabled so at each posedge of cycles the inputs will entering.

When the inputs are finished so the valid_in will dis enabled

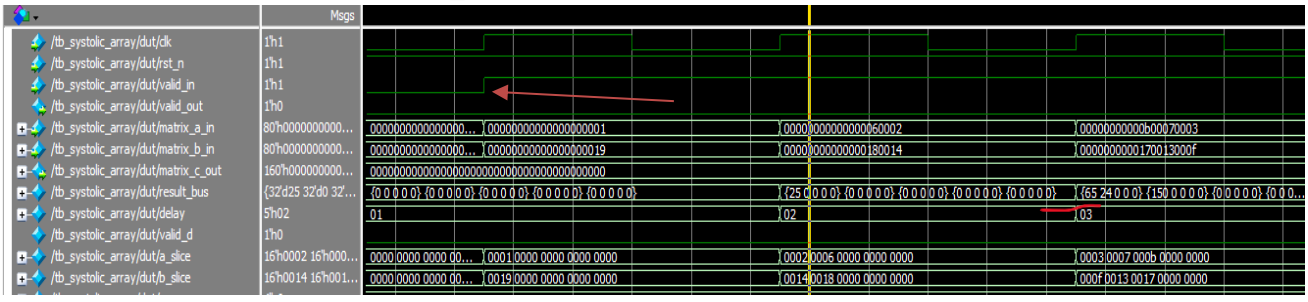


Figure 18:5x5 Test case indicating the enable of valid_in and data entry flow

As shown in fig , when the inputs are finished the valid_out will be enabled so that the design will output matrix c row by row at each posedge then when the output finished its valid output multiplied then the valid_out will dis enabled again.

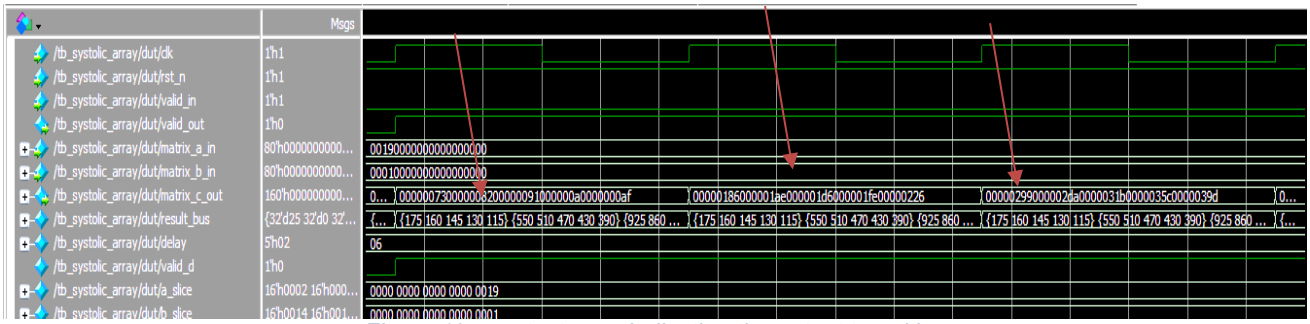


Figure 19: 5x5 test case indicating the output transition

The Output Results as shown in the log file indicating the inputs matrixes and the expected and actual results so correct output results.

```
# Matrix A:
#     1    2    3    4    5
#     6    7    8    9   10
#    11   12   13   14   15
#    16   17   18   19   20
#    21   22   23   24   25
# Matrix B:
#    25   24   23   22   21
#    20   19   18   17   16
#    15   14   13   12   11
#    10    9    8    7    6
#     5    4    3    2    1
# Expected Result (A * B):
#       175      160      145      130      115
#       550      510      470      430      390
#       925      860      795      730      665
#      1300     1210     1120     1030      940
#      1675     1560     1445     1330     1215
# Row 0 collected: 0000007300000082000000910000000a0000000af
# Row 1 collected: 00000186000001ae000001d6000001fe00000226
# Row 2 collected: 00000299000002da0000031b0000035c0000039d
# Row 3 collected: 000003ac0000040600000460000004ba00000514
# Row 4 collected: 000004bf00000532000005a5000006180000068b
# === Output Results ===
# Actual Result Matrix:
#       175      160      145      130      115
#       550      510      470      430      390
#       925      860      795      730      665
#      1300     1210     1120     1030      940
#      1675     1560     1445     1330     1215
# SUCCESS: All results match expected values!
```

Snippet 16: 5x5 Log file

And so on for any N_SIZE * N_SIZE matrixes thus it is generic parametrized with any dimension you need.

Conclusion

The systolic array architecture presented here offers an efficient and scalable solution for hardware-accelerated matrix multiplication. By leveraging a network of simple, interconnected processing elements and a carefully data flow, it achieves high parallelism and throughput, making it ideal for applications requiring intensive linear algebra computations. The modular design, with parameterized DATAWIDTH and N_SIZE , allows for flexibility in adapting the array to various matrix dimensions and data precision requirements.