STMicroelectronics

Systolic Array for Applying Matrix Multiplication

Submitted to:

Eng. Ahmed Abdelsalam

Date of Submission:

July 27th, 2025

Report by: Ammar Ahmed Wahidi

Contents

Introduction

The term *systolic* originates from biology, describing the rhythmic contraction of the heart. In computing, it refers to the regular and synchronized movement of data through a network of processing elements. This concept is the foundation of systolic array architectures, where data flows steadily across processing units in a fixed pattern, enabling continuous computation.

A **systolic array** is a parallel computing architecture composed of a network of processing elements (PEs) arranged in a regular grid. Each PE performs simple operations such as multiplication and accumulation while passing data rhythmically to neighbouring elements in a synchronized manner, much like the heartbeat — hence the term “systolic.”

In the context of digital signal processing and matrix-based computations, systolic arrays are highly efficient for implementing **matrix multiplication**, **convolution operations**, and other **linear algebra tasks** due to their pipelined, dataflow-driven structure. They are especially well-suited for hardware acceleration in AI and ML workloads, such as neural network inference, where large volumes of matrix operations are performed repetitively.

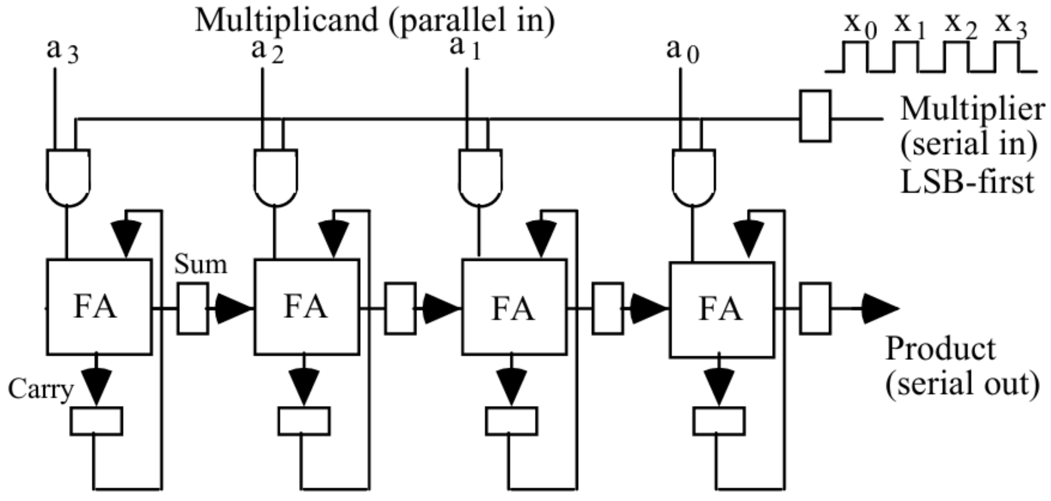
By enabling continuous data movement through the array and local computation in each PE, systolic arrays reduce the need for frequent memory accesses and allow for high throughput with deterministic latency. This makes them a core component in modern **AI accelerators**, **DSP processors**, and **custom ASICs** designed for compute-intensive tasks.

This lab aims to implement a parameterized systolic array for matrix multiplication using SystemVerilog, demonstrating our ability to design hardware-efficient, parallel dataflow architectures and verify their functionality through simulation.

Overview of Systolic Concept

Systolic arrays are specialized parallel computing architectures designed to perform efficient, pipelined computations, particularly for matrix operations. The term "systolic" draws from the biological analogy of the heart's rhythmic contractions, reflecting the synchronized, rhythmic flow of data through a grid of processing elements (PEs). This section introduces the concept of systolic and semi-systolic architectures, focusing on their application in 1d vector multiplication and the role of retiming in optimizing circuit performance.

Semi-Systolic Serial Parallel Multiplier



Semi-systolic circuit 4x4 Multiplier in 8 clock cycles.

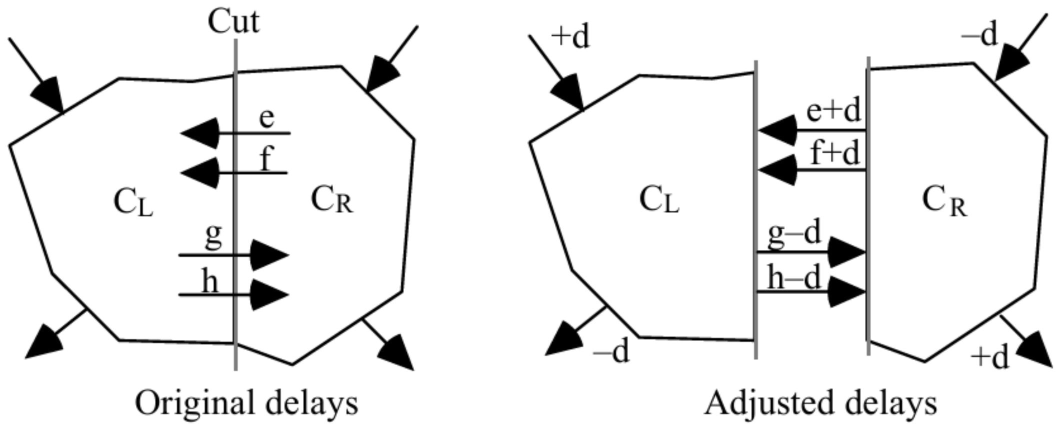
A semi-systolic multiplier is a precursor to a fully systolic design, combining serial and parallel data processing. For example, a 4x4 semi-systolic multiplier completes multiplication in 8 clock cycles. In this architecture, one operand is input bit-serially (one bit per clock cycle, starting with the least significant bit), while the other is provided in parallel, typically stored in registers or memory. The core operation resembles a carry-save adder, where partial products are computed and accumulated.

For instance, consider a serial input X_0 stored in a flip-flop and multiplied by a parallel operand A using AND gates. The outputs of these gates represent the partial product A*X_0However, scaling this design to larger bit-widths, such as 64 bits, introduces challenges like high fanout and long wire delays. In a semi-systolic design, signals from A and X_0may drive numerous AND gates, causing significant load and latency issues. These limitations in VLSI designs necessitate the adoption of fully systolic architectures to mitigate fanout and wire-length problems.

Systolic Retiming as a Design Tool

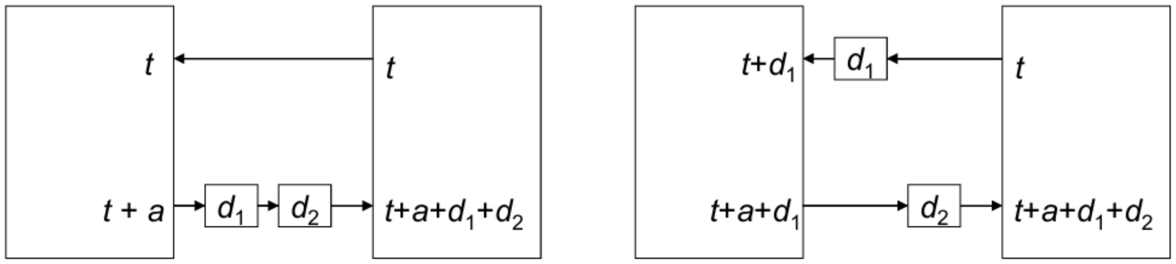
Retiming is a critical technique for transforming a semi-systolic circuit into a fully systolic one, optimizing performance without altering functionality. Retiming involves strategically inserting or removing delays (via flip-flops) to adjust signal propagation while preserving the circuit's external behavior. For example, inputs to a circuit segment can be delayed by d units, and outputs advanced by the same d units, ensuring that the relative timing of signals remains unchanged.

Example of retiming by delaying the inputs to C_L and advancing the outputs from C_R by d units.



Consider a circuit with signals e and f, representing delays of 2 and 3 flip-flops, respectively, for data moving left-to-right, and signals g and f for the opposite direction. By adding d flip-flops to left-to-right paths (increasing delay to e+d) and subtracting d from right-to-left paths (reducing delay to g−d, assuming g>d), the circuit's external behavior remains consistent, as the total delay (e+d+g−d=e+g) is unchanged. This approach eliminates long wire paths and high fanout, enhancing scalability.

Another Explanation of systolic retiming

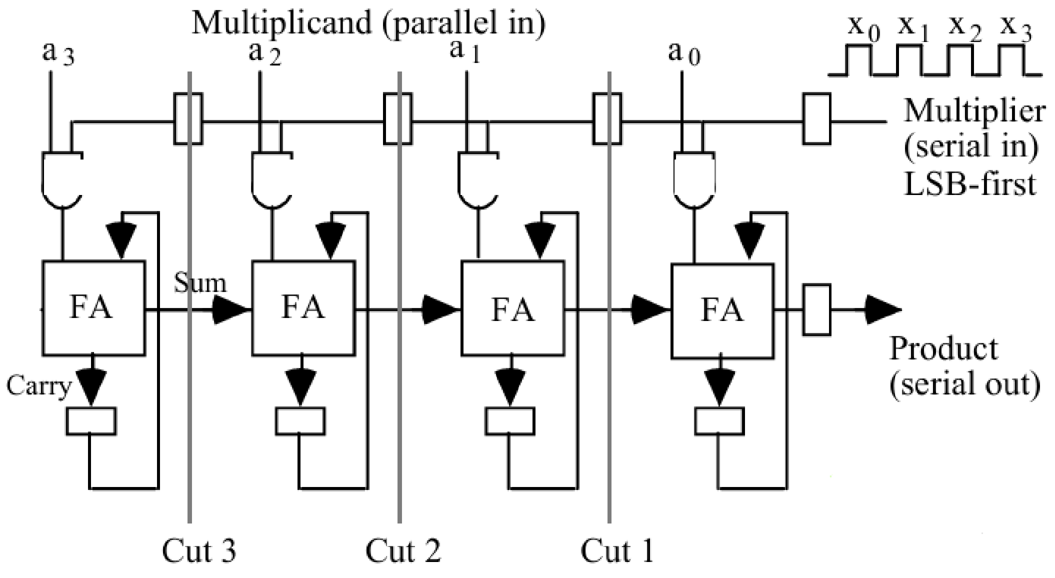


In Zero delay path, if I put latches (units delay paths), then I won't have long wire and I won't have fanout problem, so I basically in systolic circuit I try avoiding zero delay path.

Transferring delays from the outputs of a subsystem to its inputs does not change the behavior of the overall system.

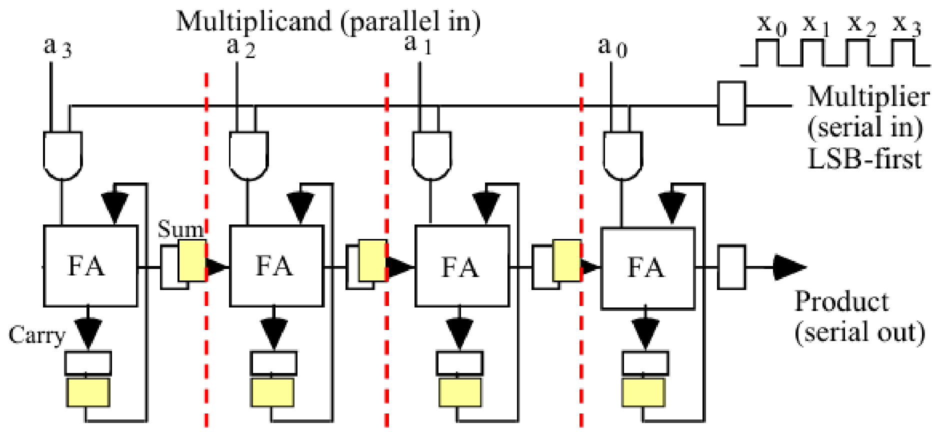
A key objective in systolic design is to avoid zero-delay paths, which occur when signals propagate through combinational logic without intervening registers. Such paths lead to long wire delays and fanout issues. By inserting latches to create unit-delay paths, systolic circuits ensure that data moves rhythmically through the PEs, improving timing reliability and reducing signal integrity problems.

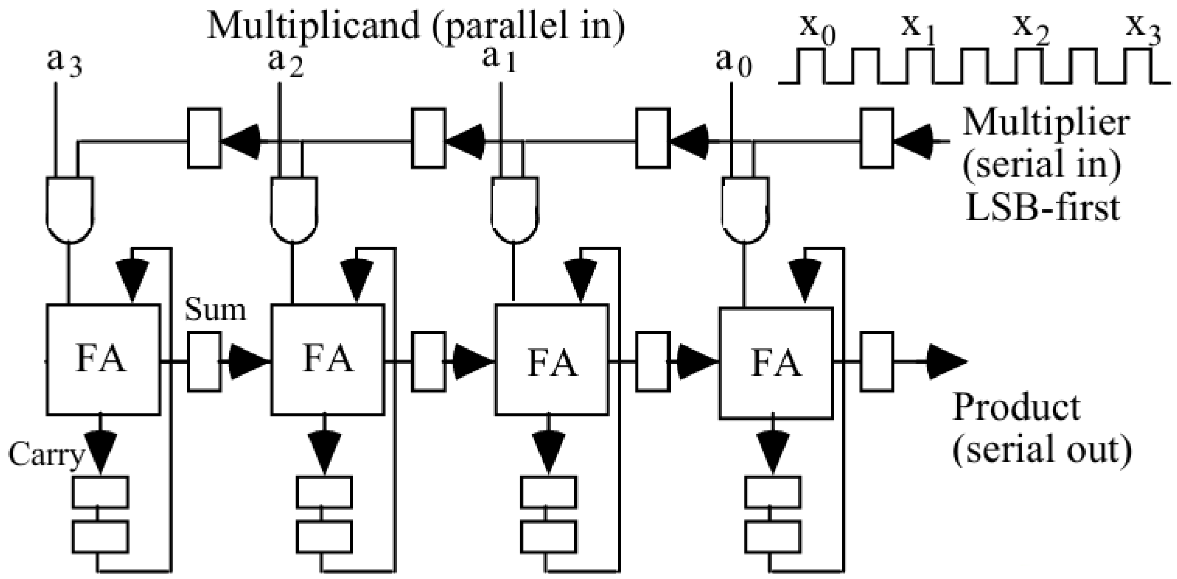
A First Attempt at Retiming



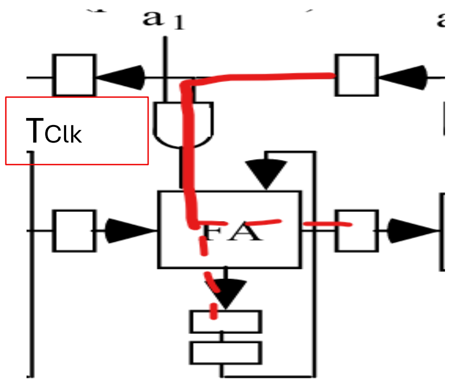
An initial retiming effort for a semi-systolic multiplier addresses fanout and wire-length issues by introducing delays. However, challenges such as zero-delay paths in the sum computation may persist, requiring high clock frequencies to meet timing constraints. A solution involves doubling the delay in the carry path and introducing new bits over additional clock cycles to shorten the critical path in the sum computation, paving the way for a fully systolic design.

Deriving a Fully Systolic Multiplier





Systolic circuit for 4×4 multiplication in 15 cycles.



A fully systolic 4x4 multiplier, achieved through iterative retiming, completes 1d Vector multiplication in 15 clock cycles. By carefully pipelining data through a grid of PEs, each performing multiply-accumulate operations, the design ensures synchronized data flow, eliminates zero-delay paths, and optimizes throughput. This architecture is ideal for high-performance applications like AI accelerators, where matrix multiplication is a core operation.

Architecture

Top-Level Architecture Overview

Design Inputs & Output Signals

Parameter Name	Type	Default Value	Description
DATAWIDTH	Integer	16	Datawidth of elements in the input matrices
N_SIZE	Integer	5	Size of the resulting matrix; also the number of PEs in each row and column (square matrix assumed)

Port Name	Direction	Width	Description
clk	Input	1-bit	Positive-edge clock signal
rst_n	Input	1-bit	Active-low reset signal
valid_in	Input	1-bit	Set to 1 when valid data is present on \`matrix_a_in\` and \`matrix_b_in\`
matrix_a_in	Input	N_SIZE × DATAWIDTH	Column-wise input of matrix A elements into the systolic array rows
matrix_b_in	Input	N_SIZE × DATAWIDTH	Row-wise input of matrix B elements into the systolic array columns
valid_out	Output	1-bit	Set to 1 when a valid row of output matrix is available on \`matrix_c_out\`
matrix_c_out	Output	N_SIZE × 2 × DATAWIDTH	Array of outputs representing one row of the resulting matrix C

Local Parameters

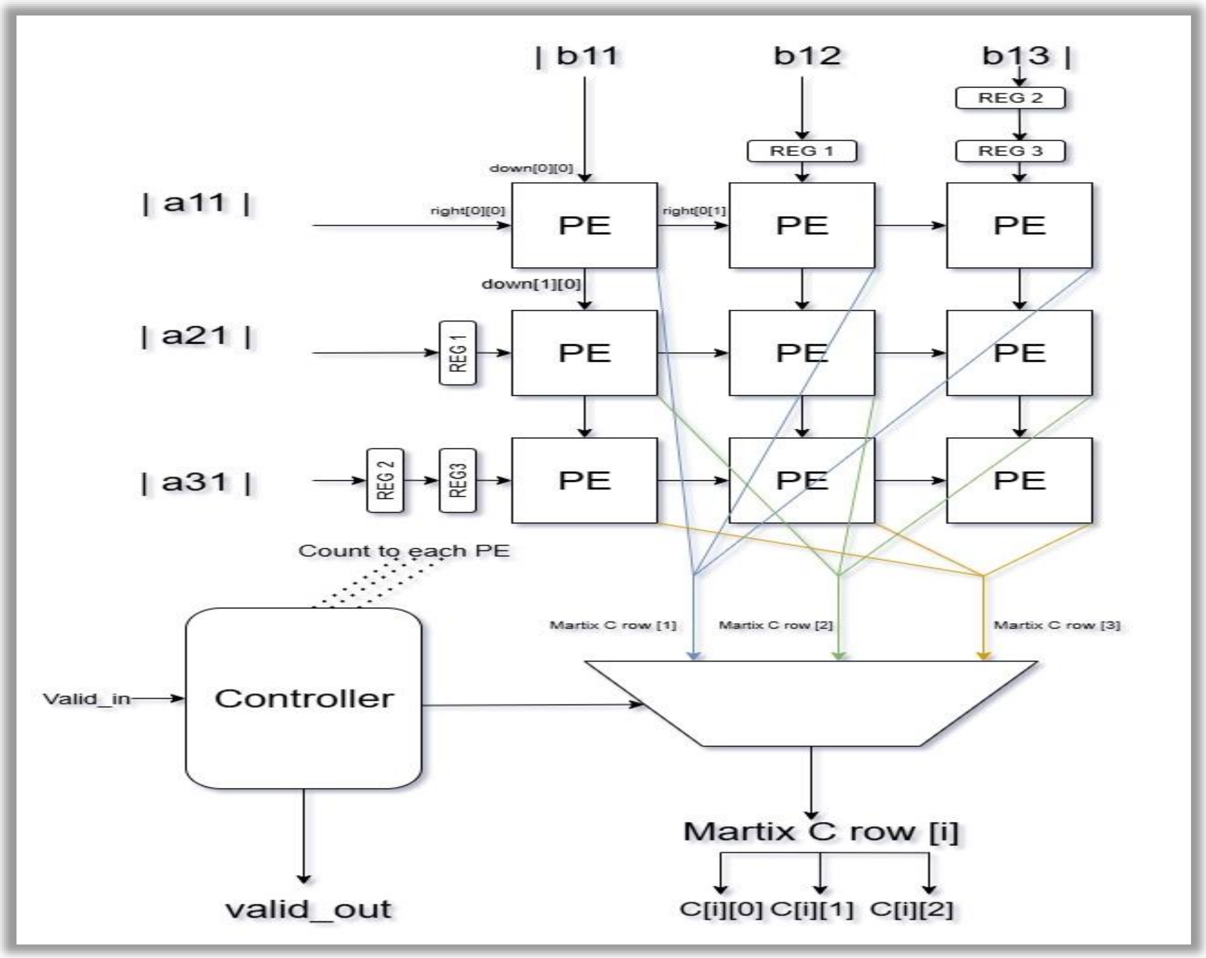
```
localparam        N_SEL           =    $clog2(N_SIZE)        ;
localparam        NUM_OF_REGS     =    ((N_SIZE1)*N_SIZE)/2  ;
```

N_SEL: Number of bits needed to select among N_SIZE options (used for mux/select logic)

NUM_OF_REGS: Total number of intermediate registers required for data shifting in triangular reg logic (based on the sum of arithmetic series 1 + 2 + ... + (N_SIZE - 1))

Block Diagram

The top-level architecture of the systolic array design integrates multiple submodules to perform efficient parallel matrix multiplication. The "`systolic_array`" module acts as the main controller, interfacing with input matrices, coordinating data movement across a grid of processing elements (PEs), and producing the output matrix. Key subcomponents include the PE array, a control unit (Controller), and a multiplexer (mux_out) for selecting the appropriate result. The design is fully parameterized to support scalable matrix sizes and data widths.



Data Flow

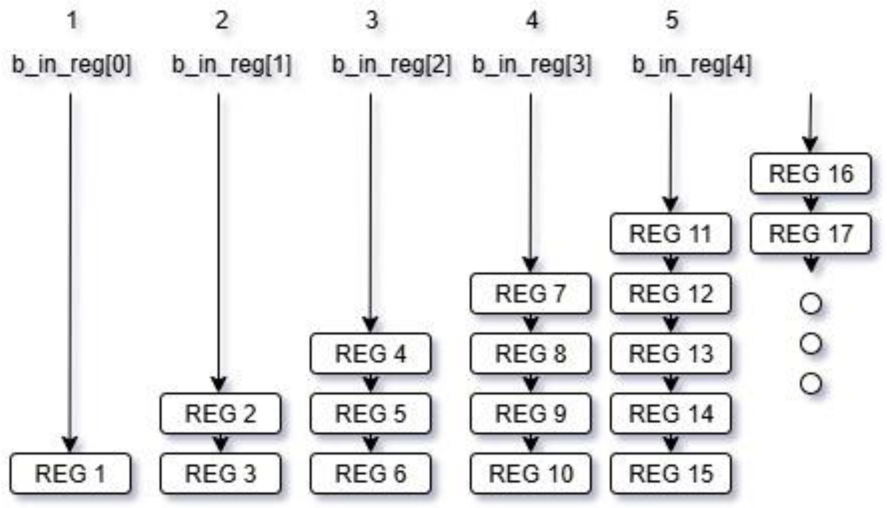
Storing Inputs Before Systolic Processing

```
reg     [DATAWIDTH1:0]            a_in_reg            [N_SIZE]                 ;
reg     [DATAWIDTH1:0]            b_in_reg            [N_SIZE]                 ;
// ==========================
// Register input matrices row-wise and column-wise
// ==========================
generate;
    for (i = 0 ; i<N_SIZE ; i=i+1)
    begin
        always @(posedge clk or negedge rst_n)
        begin
            if (~rst_n)
            begin
                a_in_reg[i]    <=  0             ;
                b_in_reg[i]    <=  0             ;
            end
            else if (valid_in)
            begin
                a_in_reg[i]    <=  matrix_a_in[i]   ;
                b_in_reg[i]    <=  matrix_b_in[i]   ;
            end
            else
            begin
                a_in_reg[i]     <= 0   ;
                b_in_reg[i]     <= 0   ;
            end
        end
    end
endgenerate
```

This block generation captures and stores the input matrices A and B into internal registers (**a_in_reg** and **b_in_reg**) of 8 bits and arrays at each clock cycle when **valid_in** is asserted. Matrix A is stored row-wise, while matrix B is stored column-wise, ensuring proper synchronization and alignment before feeding data into the systolic array. Reset logic is included to initialize the registers during startup or when needed.

How Matrix Flows into the Systolic Array

```
// ==========================
// Pipeline for matrix_b data into reg1_down
// ==========================
generate;
for (i_in=1;i_in<=N_SIZE-1;i_in=i_in+1)
begin
    always @(posedge clk or negedge rst_n)
    begin
        if (~rst_n)
        begin
            regb    [(((i_in-1)*i_in)/2)+1] <= 0                                ;
        end
        else
        begin
            regb    [(((i_in-1)*i_in)/2)+1] <= b_in_reg [i_in][DATAWIDTH-1:0]   ;
        end
    end
    if(i_in>1)
    begin
        for(i_depth=(((i_in-1)*i_in)/2)+2;i_depth<((((i_in-
1)*i_in)/2)+1)+i_in;i_depth=i_depth+1)
        begin
            always @(posedge clk or negedge rst_n)
            begin
                if (~rst_n)
                begin
                    regb    [i_depth] <= 0                      ;
                end
                else
                begin
                    regb    [i_depth] <= regb   [i_depth-1]     ;
                end
            end
        end
    end
    assign reg1_down    [1] = regb  [1]                         ;
    if(i_in>1)
    begin
        assign reg1_down    [i_in] = regb   [(((i_in-1)*i_in)/2)+1+i_in-1]  ;
    end
end
endgenerate
```



The pipeline logic for handling matrix_b_in values is implemented using a triangular register structure, as depicted in the adjacent figure. This structure ensures that elements from the input array b_in_reg are properly aligned and propagated through the array of processing elements (PEs).

Before we start The algorithm just we have to compute number of registers in terms of N_SIZE:

- N_SIZE = 2 → NUM_OF_REGS = 1
- N_SIZE = 3 → NUM_OF_REGS = 3
- N_SIZE = 4 → NUM_OF_REGS = 6
- N_SIZE = 5 → NUM_OF_REGS = 10

The equation we get is

$$\text{NUM\_OF\_REGS} = \frac{(\text{N\_SIZE} - 1) \times \text{N\_SIZE}}{2}$$

The algorithm can be conceptually divided into three main parts: (1) **Initial placement**, (2) **Vertical pipelining**, and (3) **Final assignment to reg1_down**.

1. In the **first stage**, each element b_in_reg[i] is assigned to a specific starting register in the regb array based on the formula (((i-1)*i)/2)+1.

This creates staggered entry points for the data, forming the base of each vertical pipeline column. For example:

- b_in_reg[1] → regb[1]
- b_in_reg[2] → regb[2]
- b_in_reg[3] → regb[4]
- b_in_reg[4] → regb[7]
- b_in_reg[5] → regb[11]

So b_in_reg[i] = regb[$\frac{(i-1)\times \mathrm{i}}{2} + 1$]

These assignments correspond to the first row of each "ladder" seen in the figure and ensure that each input feeds into its respective pipeline.

2. In the **second stage**, a sequence of registers is vertically stacked above one another for each input. At every positive clock edge, each register shifts its current value down to the next register in the same column. This step is governed by another generate loop and relies on simple indexing progression, where each regb[i] gets its value from regb[i-1]. This operation allows values to flow downward from top to bottom over successive clock cycles, preserving synchronization across the systolic structure.

3. Finally, in the **third stage**, the last value in each vertical register stream is mapped to an index in the reg1_down array. This is done using the same offset formula from the initial placement but with an added offset to reach the correct end-point:
   - For i = 1, reg1_down[1] = regb[1]
   - For i > 1, reg1_down[i] = regb[$\frac{(i-1)\times \mathrm{i}}{2} + i$]

   This step extracts the bottom-most register value from each pipeline and feeds it to the correct column input of the systolic array. The result is a neatly pipelined flow of matrix B elements, synchronized and distributed in a triangular register pattern, optimizing the data feed into the PE array.

A similar pipelining approach is applied to the matrix_a_in elements and stored in reg1_right, forming the row-wise input flow of matrix A.

Instantiation of Processing Elements (PEs)

```
// ==========================
// Instantiate N_SIZE x N_SIZE Processing Elements (PEs)
// Each PE receives:
// - one input from the left (a  = right[k][j])
// - one input from the top  (b  = down[k][j])
// - passes its 'a' value to the right neighbor (a_right)
// - passes its 'b' value to the bottom neighbor (b_down)
// - outputs partial result c_out to be collected later
// 'count' is broadcast to all PEs for timing control
// ==========================
generate;

    // Feed first row and column of systolic array
    for (l=0;l<N_SIZE;l=l+1)
    begin
        assign  down    [0][l] = reg1_down  [l]     ;
        assign  right   [l][0] = reg1_right [l]     ;
    end

    for (k=0;k<N_SIZE;k=k+1)
    begin
        for (j=0;j<N_SIZE;j=j+1)
        begin
            PE #(DATAWIDTH,N_SIZE) PE (
            .clk(clk)                       ,
            .rst_n(rst_n)                   ,
            .count(count)                   ,
            .a(right[k][j])                 ,
            .b(down [k][j])                 ,
            .a_right(right[k][j+1])         ,
            .b_down(down [k+1][j])          ,
            .c_out(c[(j+k*N_SIZE)+1])
            );
        end
    end
endgenerate
```

This Code section instantiates the N_SIZE × N_SIZE systolic array of Processing Elements (PEs), which form the core computational structure for matrix multiplication. Each PE performs a Multiply-Accumulate (MAC) operation using two inputs: one received from the left neighbor (denoted as a) and one from the top (denoted as b). These inputs are passed through internal registers and then forwarded to adjacent PEs — with a propagating to the right (a_right) and b to the bottom (b_down). The partial product output c_out from each PE is indexed and stored in an output array to be collected and packed later. The outermost row and column of the array are initialized using the previously prepared reg1_right and reg1_down registers. The count signal is also globally distributed to all PEs to synchronize their timing and enable pipelined data flow across the array. This structured and fully pipelined architecture ensures continuous throughput of matrix data, ideal for high-performance applications.

Controller Instantiation in Top-Level Module

```
// Instantiate Controller
Controller #(DATAWIDTH,N_SIZE) CU (
.clk(clk)                ,
.rst_n(rst_n)            ,
.valid_in(valid_in)      ,
.valid_out(valid_out)    ,
.sel(sel)                ,
.count_out(count)
);
```

In the systolic_array top module, the Controller module is instantiated to manage the global control signals that orchestrate the operation of the systolic array. This includes generating the count signal, which serves as a timing reference for all Processing Elements (PEs), and producing a selection signal sel used by the output multiplexer. The Controller also asserts the valid_out signal to indicate when the output results are ready. It monitors the valid_in signal to determine when to begin counting and transitions through internal states accordingly. The parameters DATAWIDTH and N_SIZE are passed to ensure consistent configuration with the rest of the array, aligning the control logic with the datapath components.

Packing the Output Results

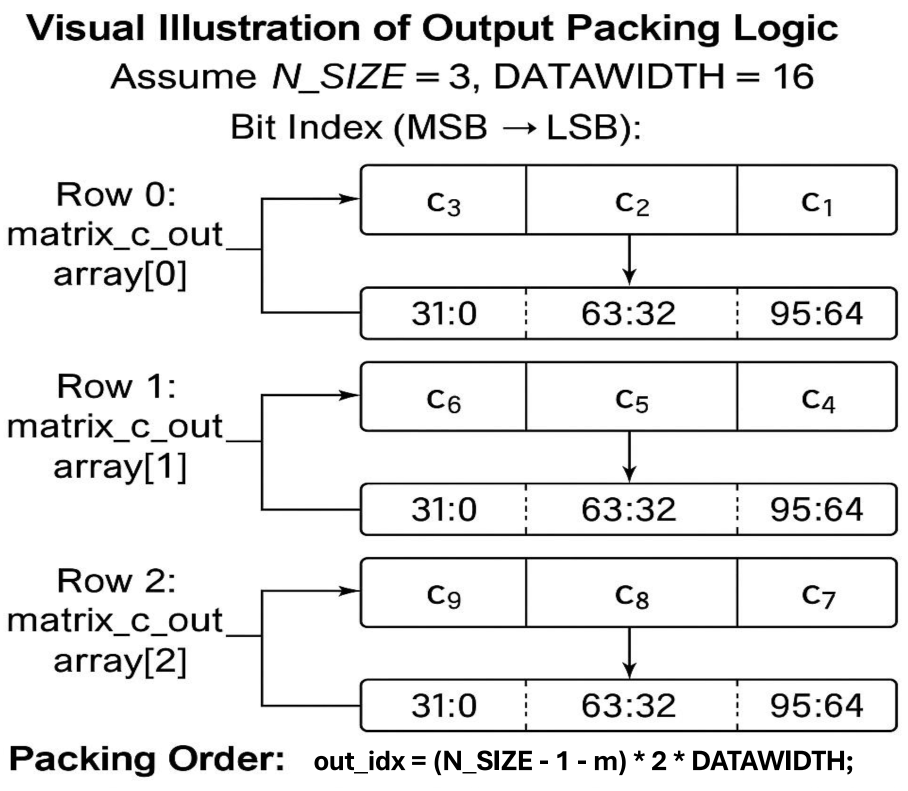
```
// ==========================
// Pack the output results from 'c' into matrix_c_out_array
// ==========================
generate
    for (t = 0; t < N_SIZE; t = t + 1) begin : pack_c_row
        for (m = 0; m < N_SIZE; m = m + 1) begin : pack_bits
            // Compute the correct bit slice in the output bus
            localparam int out_idx = (N_SIZE - 1 - m) * 2 * DATAWIDTH;
            localparam int c_idx   = t * N_SIZE + m + 1;  // since c[1] to c[N]

            assign matrix_c_out_array   [t][out_idx +: 2*DATAWIDTH] = c[c_idx]  ;
        end
    end
endgenerate
```

This block is responsible for organizing and **packing the computed results** from all Processing Elements (PEs) into a structured format suitable for output. The generate block iterates through each row (t) and column (m) of the resulting matrix, pulling the partial products stored in the array c and correctly aligning them into the matrix_c_out_array.

Each output value from the systolic array (c[c_idx]) is **2×DATAWIDTH bits** and must be placed at a specific position inside the corresponding output row bus. The position is calculated using:

- out_idx = (N_SIZE - 1 - m) * 2 * DATAWIDTH: which determines the bit offset where the current value should be inserted.
- c_idx = t * N_SIZE + m + 1: computes the correct index into the c array (note that c is 1-indexed).

This careful packing ensures that the matrix product is flattened correctly row by row into a contiguous bus that can be used for downstream processing or output.


Visual Illustration of Output Packing Logic
Assume $N\_SIZE = 3$, DATAWIDTH = 16
Bit Index (MSB → LSB):
Row 0: matrix_c_out array[0]
c_3
c_2
c_1
31:0
63:32
95:64
Row 1: matrix_c_out array[1]
c_6
c_5
c_4
31:0
63:32
95:64
Row 2: matrix_c_out array[2]
c_9
c_8
c_7
31:0
63:32
95:64
Packing Order:

out_idx = (N_SIZE - 1 - m) * 2 * DATAWIDTH;

c_idx = t * N_SIZE + m + 1;

matrix_c_out_array[t][out_idx +: 2*DATAWIDTH] = c[c_idx];

Generic Multiplexer (mux_out) Module Description

```
module mux_out #(
    parameter DATAWIDTH = 160,              // Width of default each input (e.g.
5 × 2 × 16)
    parameter N_SIZE = 5                    // Number of default rows in matrix C
)(
    input      [DATAWIDTH-1:0] in [N_SIZE] ,   // Array of N inputs, each WIDTH
bits
    input      [$clog2(N_SIZE)-1:0] sel    ,   // Selector
    output reg [DATAWIDTH-1:0] out             // Output
);

    // Combinational multiplexer logic
    always_comb begin
        out = '0;               // default
        if (sel < N_SIZE)
            out = in[sel];
    end
endmodule
```

The mux_out module is a generic, parameterized multiplexer designed to select one row from multiple parallel outputs, specifically used for retrieving a selected row from the result matrix in a systolic array. It takes an array of N_SIZE inputs, where each input is double DATAWIDTH bits wide, and routes one of them to the output based on a selection signal sel. This is implemented using a combinational always block, ensuring zero clock-cycle latency. The selection index is bounded by a $clog2(N_SIZE)-wide input to support any configurable matrix size. A default output value of zero ensures stability when sel is outside the valid range, preventing undefined behavior.

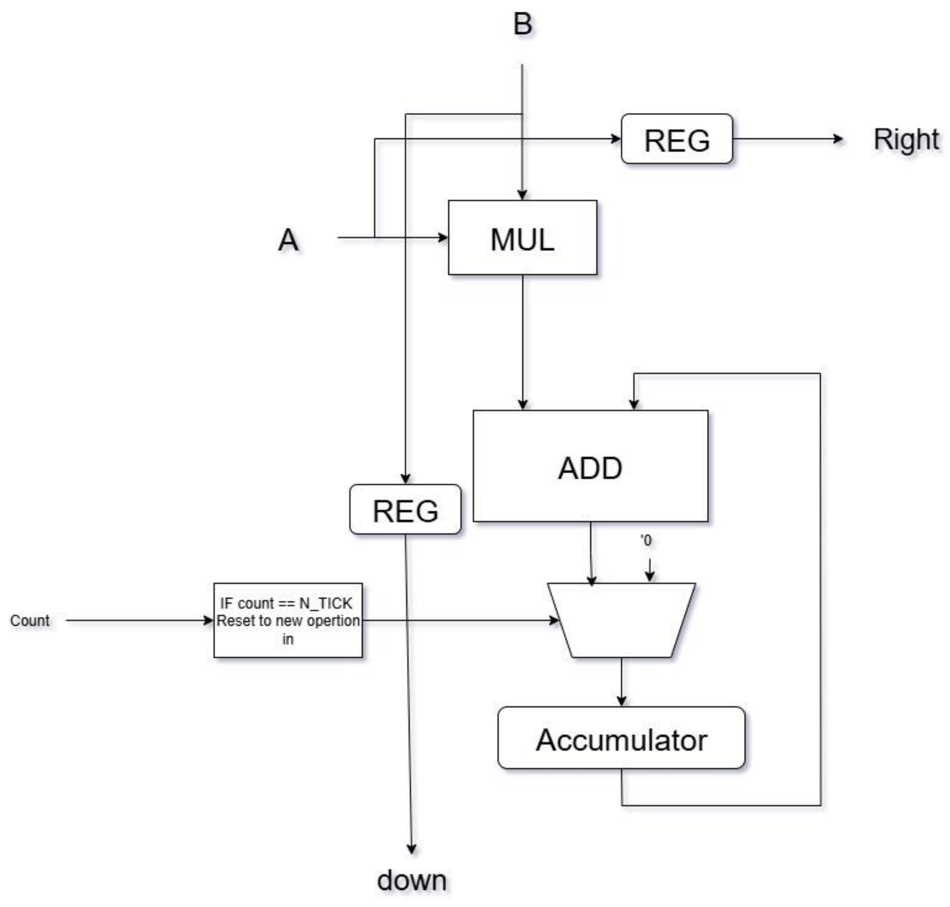
Instantiation of mux_out in Top-Level Module

```
// Output MUX for result row selection
mux_out #(2*DATAWIDTH*N_SIZE,N_SIZE) MUX (
.in(matrix_c_out_array) ,
.sel(sel)               ,
.out(o_mux)
);
```

Within the systolic_array top-level module, the mux_out component is instantiated to serve as the final selection mechanism for the output matrix row. Specifically, the multiplexer receives matrix_c_out_array, a structured array containing all the rows of the computed matrix C, and outputs a single row based on the control signal sel, generated by the Controller module. The parameters 2*DATAWIDTH*N_SIZE and N_SIZE are passed to accommodate the full width of each row and the number of rows, respectively. The result is made available on the signal o_mux, which represents the selected row of the multiplication result. This design choice enhances modularity and scalability, making the output stage adaptable to varying matrix sizes.then reunite them by this module to get matrix_out:

```
module matrix_sep #(parameter DATAWIDTH = 16, N_SIZE = 5) (
input       [2*DATAWIDTH*N_SIZE-1:0]        matrix_c_unsep                ,
output      [2*DATAWIDTH-1:0]               matrix_c_out_sep [N_SIZE-1:0]      );
genvar i ;
generate;
    for(i=0;i<N_SIZE;i++)
    begin
       assign matrix_c_out_sep [i] = matrix_c_unsep[i*2*DATAWIDTH+:2*DATAWIDTH] ;
    end
endgenerate
endmodule
```

Processing Element (PE) Grid



Port Name	Direction	Width	Description
clk	Input	1-bit	Positive-edge clock signal
rst_n	Input	1-bit	Active-low reset signal
count	Input	⌈log₂(3×N_SIZE−2 +1)⌉	Global control tick used to reset the accumulator at the end of computation
a	Input	DATAWIDTH	Input data from the left (element of matrix A)
b	Input	DATAWIDTH	Input data from the top (element of matrix B)
a_right	Output	DATAWIDTH	Propagated value of a to the right PE

b_down	Output	DATAWIDTH	Propagated value of b to the bottom PE
C_out	Output	2 ×DATAWIDTH	Current accumulated result output (partial product)

```
module PE #(parameter DATAWIDTH = 16 , N_SIZE = 5)
(clk,rst_n,count,a,b,a_right,b_down,c_out);

localparam  N_TICKS         = 3 * N_SIZE - 2                    ;   // Total
ticks to process full matrix
localparam  COUNTER_SIZE    = $clog2(N_TICKS+1)                 ;   // Counter
width
input                           clk         ;
input                           rst_n       ;
input       [COUNTER_SIZE-1:0]  count       ;
input       [DATAWIDTH-1:0]     a           ;   // Input A from left
input       [DATAWIDTH-1:0]     b           ;   // Input B from top
output      [DATAWIDTH-1:0]     a_right     ;   // Output A to right neighbor
output      [DATAWIDTH-1:0]     b_down      ;   // Output B to bottom
neighbor
output      [2*DATAWIDTH-1:0]   c_out       ;   // Output result

wire        [2*DATAWIDTH-1:0]   a_mul_b     ;   // a * b
wire        [2*DATAWIDTH-1:0]   add_out     ;   // a * b + acc
reg         [DATAWIDTH-1:0]     a_reg       ;   // Register to hold 'a' for
propagation
reg         [DATAWIDTH-1:0]     b_reg       ;   // Register to hold 'b' for
propagation
reg         [2*DATAWIDTH-1:0]   acc         ;   // Accumulator
assign      a_mul_b     =   a       *   b           ;   // Multiply a and b
assign      add_out     =   a_mul_b +   acc         ;   // Add product to
accumulator
// ==========================
// Accumulator Register
// Reset at N_TICKS to clear old computation
// ==========================
always @(posedge clk or negedge rst_n)
begin
    if (~rst_n)
        acc     <=  0                   ;
    else if (count == N_TICKS)
        acc     <=  0                   ;
    else
        acc     <=  add_out             ;
```

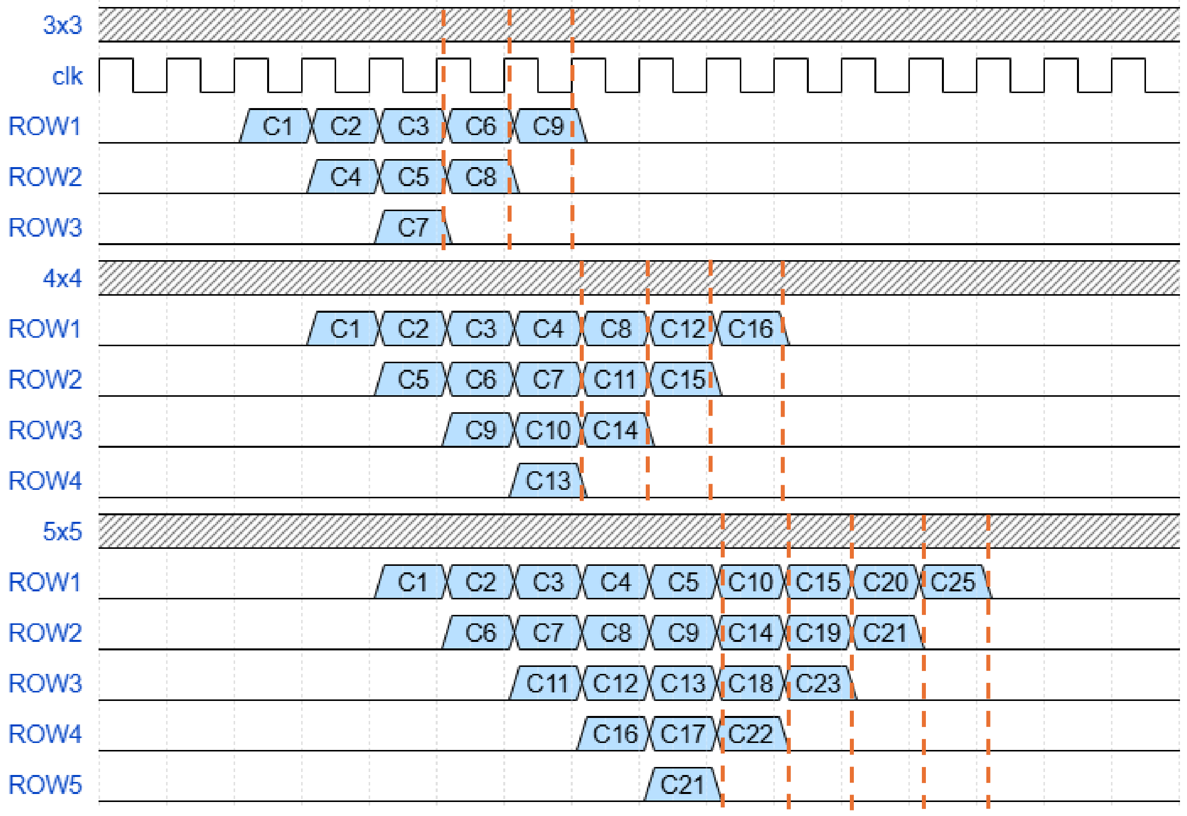
```
end
// ==========================
// Register Inputs for Pipelining
// a and b are stored to propagate to neighbors
// ==========================
always @(posedge clk or negedge rst_n)
begin
    if (~rst_n)
    begin
        a_reg     <=  0                    ;
        b_reg     <=  0                    ;
    end
    else
    begin
        a_reg     <=  a                    ;
        b_reg     <=  b                    ;
    end
end
// ==========================
// Outputs
// Propagate values and return accumulated result
// ==========================
assign      a_right    =   a_reg           ;
assign      b_down     =   b_reg           ;
assign      c_out      =   add_out         ;
endmodule
```

Each Processing Element (PE) in the systolic array performs the fundamental multiply-accumulate (MAC) operation. As shown in the figure, it takes one input A from the left and one input B from the top. These inputs are first registered to allow pipelined dataflow and then multiplied. The multiplication result is added to an internal accumulator which stores the partial sum. A control signal count determines when the accumulator should reset (specifically when count == N_TICKS, indicating the end of the current matrix operation). The result of the addition is fed back into the accumulator on the next clock cycle. Simultaneously, the registered values of A and B are passed to the right and bottom neighboring PEs respectively, enabling data propagation across the array. This design supports full pipelining and contributes to a regular and scalable architecture ideal for matrix multiplication.

Controller FSM & Timing

Port Name	Direction	Width	Description
clk	Input	1-bit	Positive-edge clock signal
rst_n	Input	1-bit	Active-low reset signal
valid_in	Input	⌈log₂(3×N_SIZE–2 +1)⌉	Indicates when input data is valid to start FSM and counting
valid_out	Input	DATAWIDTH	Indicates when the output result row is valid and ready to be sampled
sel	Input	DATAWIDTH	Select signal for the output multiplexer (mux_out) to choose result row
count_out	Output	DATAWIDTH	Global counter used to synchronize all Processing Elements and control timing

Latency and Total Operation Time Calculation



To determine the total number of ticks required to process a full matrix multiplication of size N × N, we use the equation observing from waveform:

$$N\_TICKS = 3 \times N\_SIZE - 2$$

This formula accounts for the pipeline fill, steady processing, and pipeline drain phases. By Waveform :

- For N=3: N_TICKS =7
- For N=4: N_TICKS=10
- For N=5: N_TICKS=13

Latency of one operation is $3 \times N\_SIZE - 2$

Controller Timing and Lookup Table Logic

Counter Size and LUT Depth

To handle all ticks from 0 to N_TICKS, the controller uses a counter with the following bit-width:

COUNTER_SIZE=[log2(N_TICKS+1)]

Accordingly, the size of the lookup tables used to store the sel and valid_out values is:

LUT_SIZE=$2^{counter\_size}$

These tables act like ROM, hardcoded at compile-time using a generate loop.

Sample Output Timing

The output values (e.g., C11,C12,...) are ready at specific clock ticks. For an N×N matrix:

- The first output element C11 becomes valid starting from **tick = N_SIZE**
- The **last output row** becomes valid at **tick = N_TICKS**
- The first output raw (C1 ,C2, C3) for N_SIZE = 3 will be ready in Half way of sampling element so in halfway it sample first row, halfway+1 samples second row and so on.

START_COUT_TICK = N_SIZE, END_TICK = N_TICKS

HALF_COUT_TICK = $\frac{START\_COUT\_TICK \; + \mathrm{END\_TICK}}{2}$

Thus, the **sampling window** spans from:

HALF_COUT_TICK to END_TICK

This value indicates the tick at which output sampling begins, and from there sel is incremented by one at each tick to select each result row in turn.

LUT Generation Logic

The controller defines two lookup tables:

- sel_lut[i]: Indicates the output row to be selected by the output multiplexer (mux_out) at tick i
- valid_lut[i]: A binary flag (1 or 0) indicating whether a valid output is available at tick i

```
// -------- Lookup Table --------
logic   [$clog2(N_SIZE)-1:0]    sel_lut     [LUT_SIZE]   ;
logic                           valid_lut   [LUT_SIZE]   ;

genvar i;
generate
    for (i = 0; i < LUT_SIZE; i = i + 1) begin : gen_sel_valid_table
        if (i >= HALF_COUT_TICK && i <= END_TICK)
        begin
            assign sel_lut[i]   = i - HALF_COUT_TICK    ;
            assign valid_lut[i] = 1'b1                  ;
        end else
        begin
            assign sel_lut[i]   = '0                    ;
            assign valid_lut[i] = 1'b0                  ;
        end
    end
endgenerate

// -------- Output Assignment --------
always_comb
begin
    sel         = sel_lut[count]                        ;
    valid_out   = valid_lut[count]                      ;
end
```

- At each tick from HALF_COUT_TICK to END_TICK, sel_lut selects rows 0 to N_SIZE−1N_SIZE-1N_SIZE−1
- Outside this window, output is not valid

This timing ensures the systolic array performs **fully pipelined, synchronized matrix multiplication**, with each output row being selected and sampled in order at the right moment.

FSM and Counter Logic

This section of the Controller module implements the **finite state machine (FSM)** and a global **counter** to manage the timing and synchronization of operations across the systolic array.

Internal Signals:

- count: A register that tracks the current clock cycle (tick) of the computation.
- state: The current FSM state (IDLE or ACTIVE).
- next_state: Stores the next state based on current conditions and transitions.

State Transition Logic:

The FSM has two states:

- IDLE: The default state where the system waits for a valid input signal to begin computation.
- ACTIVE: The state in which the computation is ongoing and the counter is incrementing.

The state transition behavior is defined as follows:

IDLE → ACTIVE if (valid_in == 1)

ACTIVE → IDLE if (count == END_TICK)

This ensures that the controller starts only when valid_in is asserted, and stops exactly after N_TICKS clock cycles.

Counter Behavior

The count register tracks how many clock cycles have passed during an ACTIVE operation window. It is:

- Reset to 0 when:
    - rst_n is deasserted (asynchronous reset), or
    - A new operation starts (valid_in asserted in IDLE state), or
    - Operation finishes (after reaching END_TICK)
- Incremented only during the ACTIVE state.

This counter is exposed to the top-level module via count_out, and is used internally to index the LUTs and control the overall timing of data propagation in the array.

Simulation & Results

Matrix Multiplication Validation via MATLAB Simulation

To validate the correctness of matrix multiplication logic and to compare with the hardware (SystemVerilog) implementation, I first simulated matrix multiplication using MATLAB.

A simple MATLAB script was written to accept two square matrices from the user and compute their product using the built-in matrix multiplication operator. The inputs and resulting outputs were printed clearly for verification.

```matlab
% Prompt for matrix size
n = input('Enter the size of the square matrices (n): ');

% Initialize matrices
A = zeros(n, n);
B = zeros(n, n);

% Input matrix A
disp('Enter elements for matrix A:');
for i = 1:n
    for j = 1:n
        A(i, j) = input(sprintf('A(%d,%d): ', i, j));
    end
end

% Input matrix B
disp('Enter elements for matrix B:');
for i = 1:n
    for j = 1:n
        B(i, j) = input(sprintf('B(%d,%d): ', i, j));
    end
end

% Matrix multiplication
C = A * B;

% Display result
disp('Matrix A * Matrix B =');
disp(C);
```

Case 1:

A =

[4 34 0 23

6 4 32 65

6 4 3 5

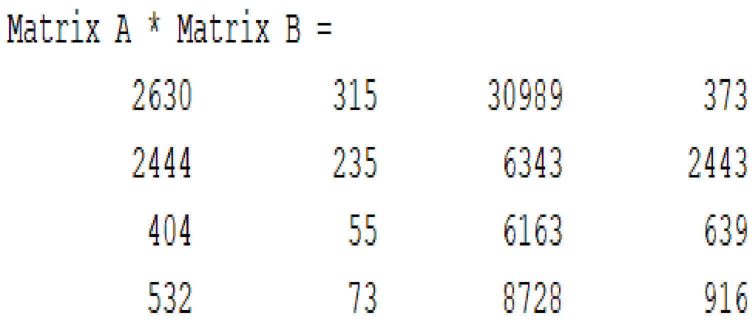
6 7 8 4]

B = [3 2 454 76

54 7 856 0

0 0 0 56

34 3 3 3]



```
Matrix A * Matrix B =
        2630         315       30989         373
        2444         235        6343        2443
         404          55        6163         639
         532          73        8728         916
```

Case 2:

A =

[-3 -32 -4 332

32 4 54 65

43 4 3 3

-3 -3 43 32]

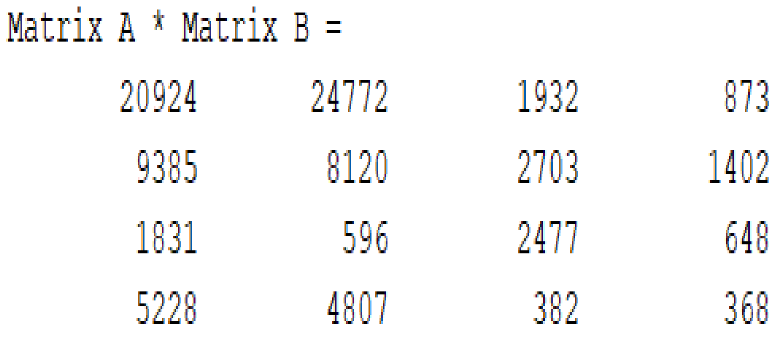
B =

[32 4 56 9

8 7 6 54

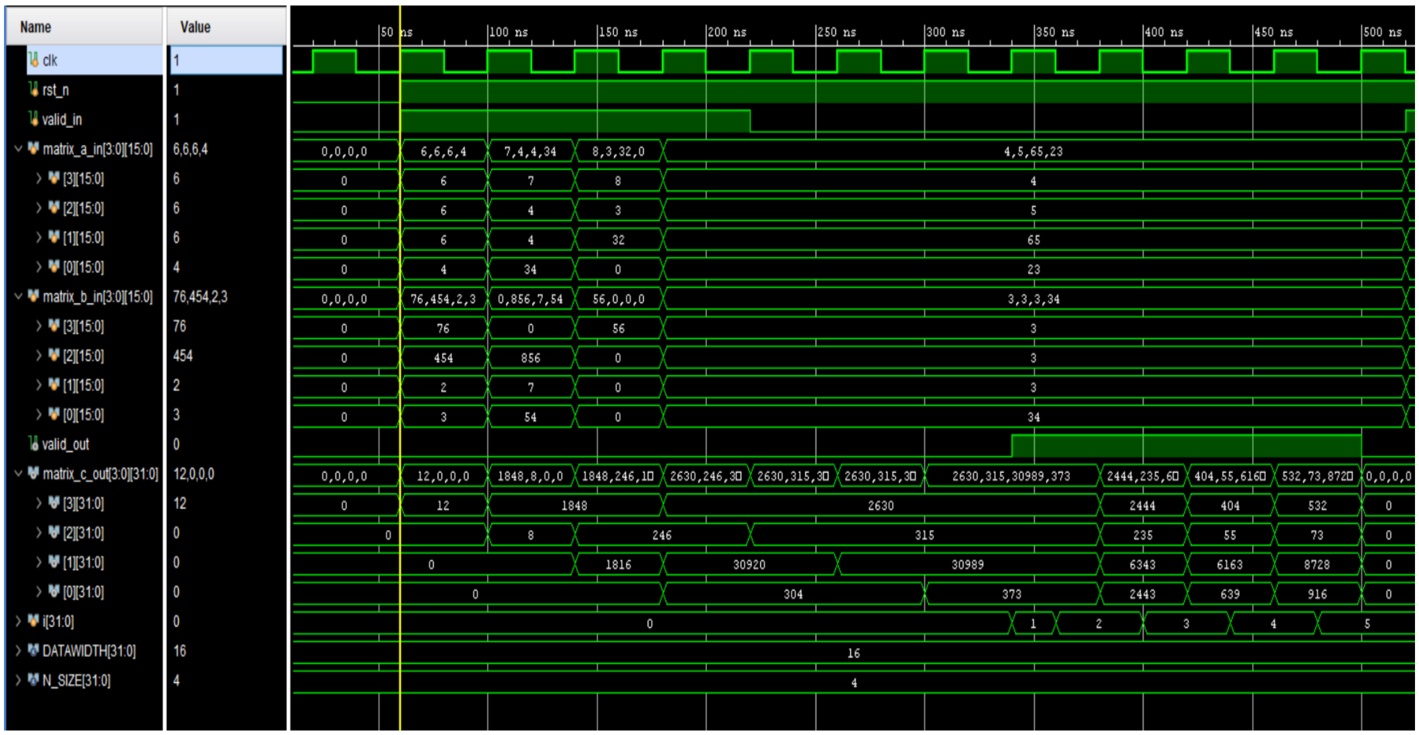
76 56 8 7

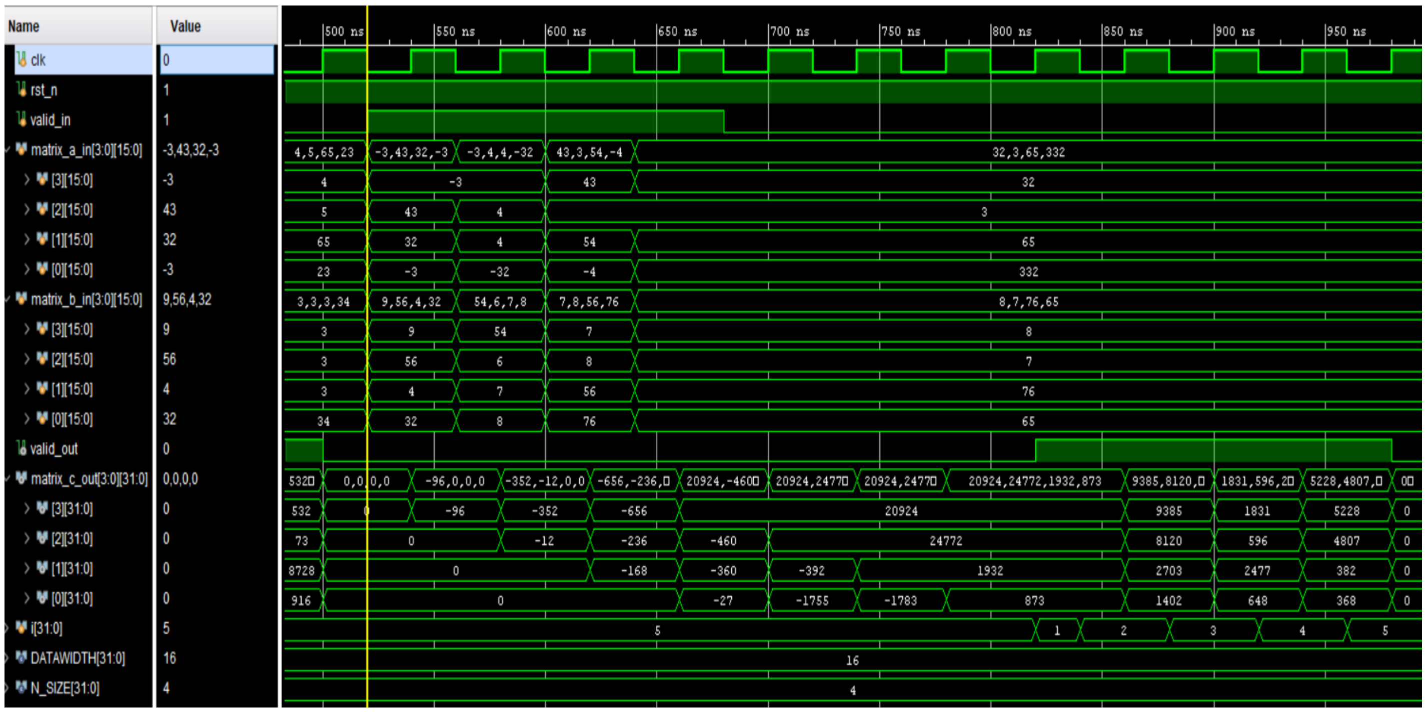
65 76 7 8]



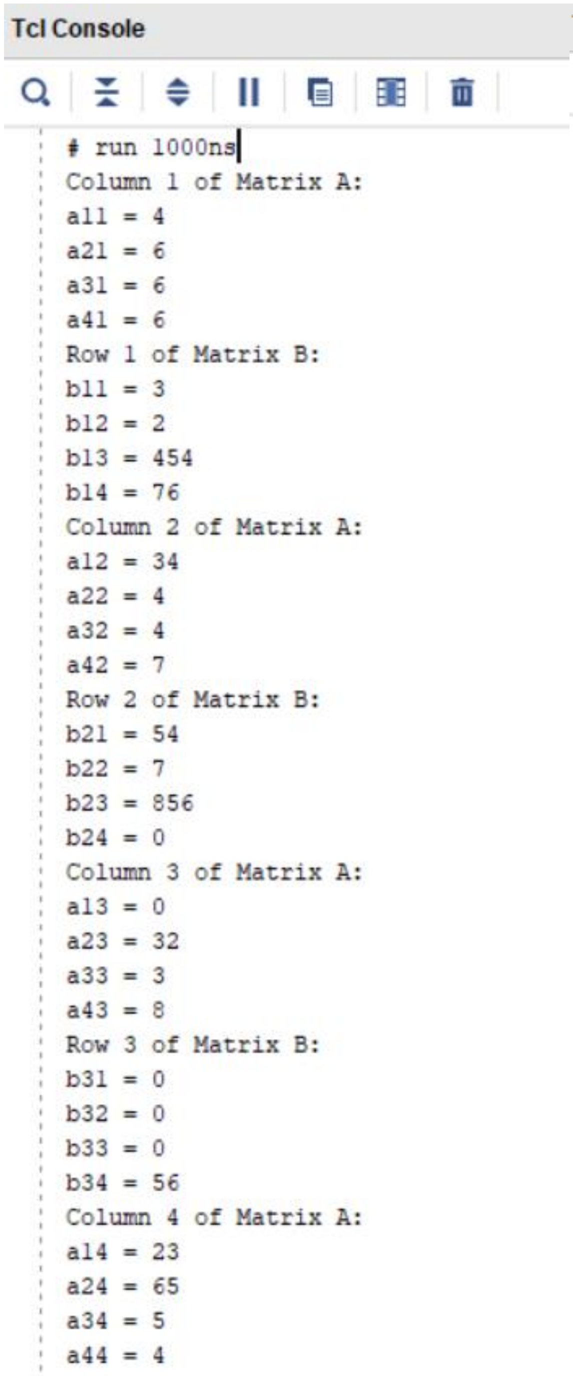
```
Matrix A * Matrix B =
       20924       24772        1932         873
        9385        8120        2703        1402
        1831         596        2477         648
        5228        4807         382         368
```

Waveform

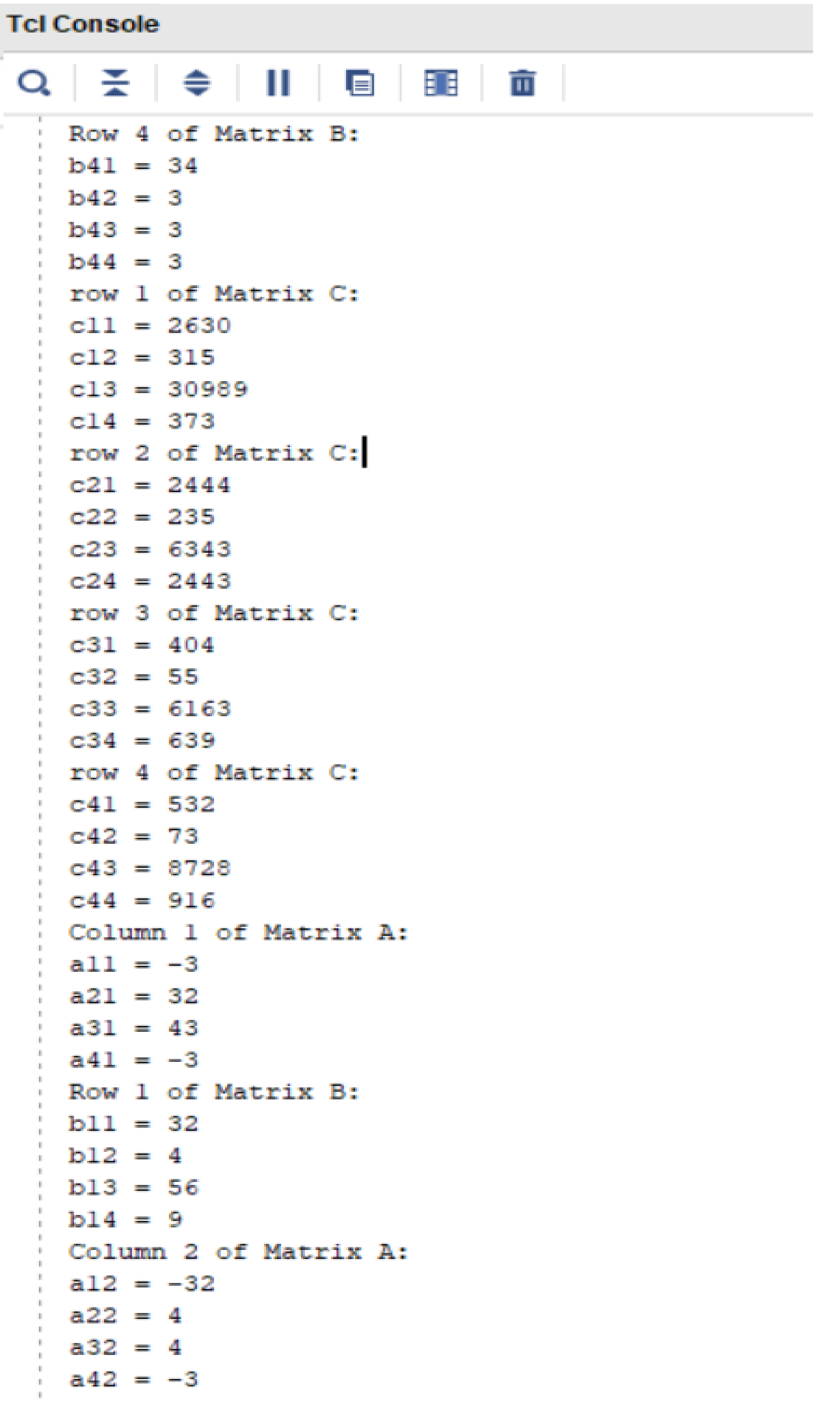

Name
Value
clk 1
rst_n 1
valid_in 1
matrix_a_in[3:0][15:0] 6,6,6,4
[3][15:0] 6
[2][15:0] 6
[1][15:0] 6
[0][15:0] 4
matrix_b_in[3:0][15:0] 76,454,2,3
[3][15:0] 76
[2][15:0] 454
[1][15:0] 2
[0][15:0] 3
valid_out 0
matrix_c_out[3:0][31:0] 12,0,0,0
[3][31:0] 12
[2][31:0] 0
[1][31:0] 0
[0][31:0] 0
i[31:0] 0
DATAWIDTH[31:0] 16
N_SIZE[31:0] 4


Name
Value
clk 0
rst_n 1
valid_in 1
matrix_a_in[3:0][15:0] -3,43,32,-3
[3][15:0] -3
[2][15:0] 43
[1][15:0] 32
[0][15:0] -3
matrix_b_in[3:0][15:0] 9,56,4,32
[3][15:0] 9
[2][15:0] 56
[1][15:0] 4
[0][15:0] 32
valid_out 0
matrix_c_out[3:0][31:0] 0,0,0
[3][31:0] 0
[2][31:0] 0
[1][31:0] 0
[0][31:0] 0
i[31:0] 5
DATAWIDTH[31:0] 16
N_SIZE[31:0] 4

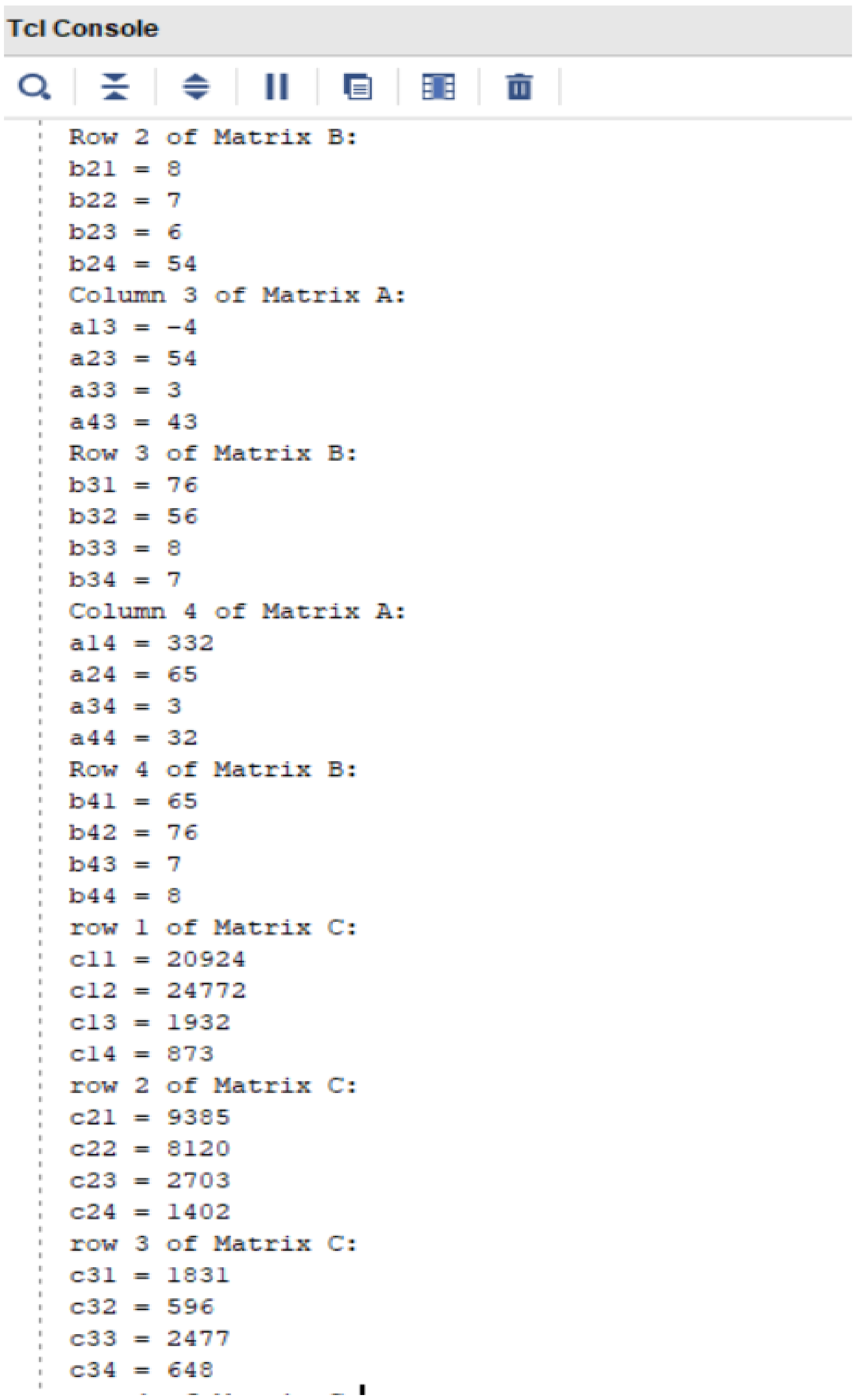
Transcript


Tcl Console

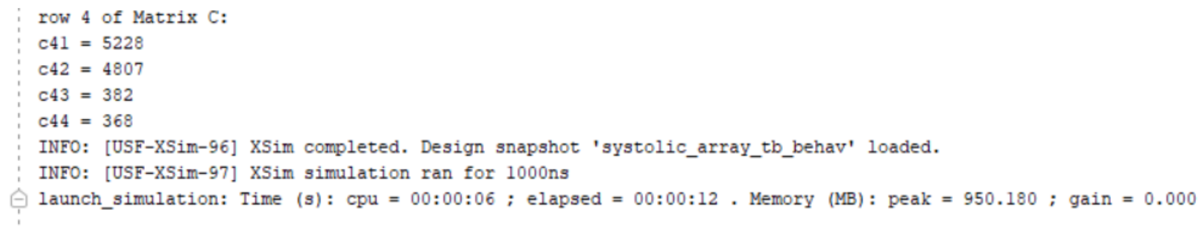
```
# run 1000ns
Column 1 of Matrix A:
a11 = 4
a21 = 6
a31 = 6
a41 = 6
Row 1 of Matrix B:
b11 = 3
b12 = 2
b13 = 454
b14 = 76
Column 2 of Matrix A:
a12 = 34
a22 = 4
a32 = 4
a42 = 7
Row 2 of Matrix B:
b21 = 54
b22 = 7
b23 = 856
b24 = 0
Column 3 of Matrix A:
a13 = 0
a23 = 32
a33 = 3
a43 = 8
Row 3 of Matrix B:
b31 = 0
b32 = 0
b33 = 0
b34 = 56
Column 4 of Matrix A:
a14 = 23
a24 = 65
a34 = 5
a44 = 4
```


Tcl Console

```
Row 4 of Matrix B:
b41 = 34
b42 = 3
b43 = 3
b44 = 3
row 1 of Matrix C:
c11 = 2630
c12 = 315
c13 = 30989
c14 = 373
row 2 of Matrix C:
c21 = 2444
c22 = 235
c23 = 6343
c24 = 2443
row 3 of Matrix C:
c31 = 404
c32 = 55
c33 = 6163
c34 = 639
row 4 of Matrix C:
c41 = 532
c42 = 73
c43 = 8728
c44 = 916
Column 1 of Matrix A:
a11 = -3
a21 = 32
a31 = 43
a41 = -3
Row 1 of Matrix B:
b11 = 32
b12 = 4
b13 = 56
b14 = 9
Column 2 of Matrix A:
a12 = -32
a22 = 4
a32 = 4
a42 = -3
```


Tcl Console

```
Row 2 of Matrix B:
b21 = 8
b22 = 7
b23 = 6
b24 = 54
Column 3 of Matrix A:
a13 = -4
a23 = 54
a33 = 3
a43 = 43
Row 3 of Matrix B:
b31 = 76
b32 = 56
b33 = 8
b34 = 7
Column 4 of Matrix A:
a14 = 332
a24 = 65
a34 = 3
a44 = 32
Row 4 of Matrix B:
b41 = 65
b42 = 76
b43 = 7
b44 = 8
row 1 of Matrix C:
c11 = 20924
c12 = 24772
c13 = 1932
c14 = 873
row 2 of Matrix C:
c21 = 9385
c22 = 8120
c23 = 2703
c24 = 1402
row 3 of Matrix C:
c31 = 1831
c32 = 596
c33 = 2477
c34 = 648
```



```
row 4 of Matrix C:
c41 = 5228
c42 = 4807
c43 = 382
c44 = 368
INFO: [USF-XSim-96] XSim completed. Design snapshot 'systolic_array_tb_behav' loaded.
INFO: [USF-XSim-97] XSim simulation ran for 1000ns
launch_simulation: Time (s): cpu = 00:00:06 ; elapsed = 00:00:12 . Memory (MB): peak = 950.180 ; gain = 0.000
```

References

1. STMicroelectronics: LAB 0 Systolic Array for Applying Matrix Multiplication Specs, Ahmed Abdelsalam
2. COMPUTER ARITHMETIC Algorithms and Hardware Designs SECOND EDITION, Behrooz Parhami