Matrix Multiplication – Systolic Array

Digital Design and Logical Synthesis for Electrical Computer Engineering (36113611)

Matrix Multiplication

matmul

Digital High-Level Design

Version 0.1

Revision Log

Rev	Change	Description	Reason for change	Done By	Date
0.1	Initial document			Oren Schor, Tal Shvartzberg	6.2.24
0.2	Adding block diagrams and tables			Oren Schor, Tal Shvartzberg	6.2.24
0.3	Add descriptions in every block diagram			Oren Schor, Tal Shvartzberg	8.2.24
0.4	Finishing			Oren Schor, Tal Shvartzberg	9.2.24

Contents

LIST OF FIGURES

LIST OF TABLES

1. BLOCKS FUNCTIONAL DESCRIPTIONS

1.1 Top Entity – matmul

The choice to use a systolic array approach in the matmul.v module for matrix multiplication is inspired by its high efficiency in computational parallelism and data flow optimization, as detailed in the 1982 article by Kung on systolic architectures [1]. Systolic arrays allow for simultaneous execution of multiple data processing operations by passing data through a network of processors in a rhythmic, pulse-like manner, akin to the human circulatory system. This architecture is particularly well-suited for matrix operations due to its ability to efficiently handle large volumes of data with minimal memory access delays, offering significant improvements in throughput and scalability for hardware implementations of matrix multiplication. The systolic array design enables the matmul module to perform matrix multiplication with high parallelism, making it an optimal choice for hardware acceleration in applications requiring fast, efficient processing of linear algebra computations.

The module operates within a synchronous clock (clk_i) and asynchronous reset (rst_n_i) inputs, ensuring precise timing and operational control. It initiates matrix multiplication through a start signal (start_operation_i) originating from the control register, delivers flattened representations of matrices A and B (a_flat_i and b_flat_i) from the memory to the matmul calculator module, which producing a flattened result matrix (result_flat_o) that is later stored in the scratchpad memory.

The design of the matrix multiplication project leverages parameterized Verilog code extensively across its various modules to achieve a high degree of flexibility and scalability. This approach allows for the entire design to be adaptable to different matrix sizes and data widths, facilitating code reuse and simplification of design modifications. By employing parameters such as DATA_WIDTH and BUS_WIDTH, the design can dynamically adjust its internal structures, such as memory allocation for operands and processing element configurations, to accommodate varying operand sizes and multiplication complexities.

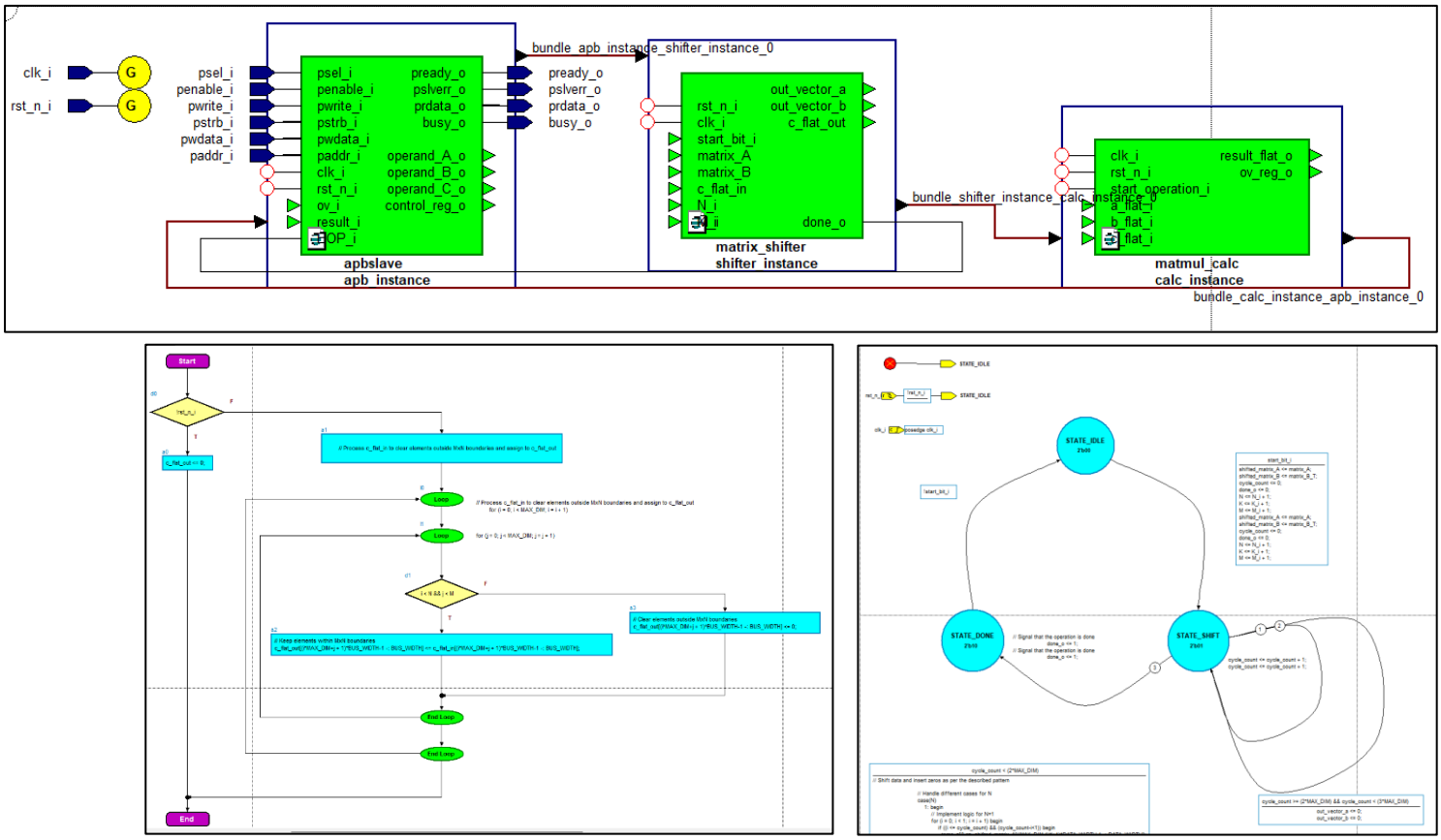


Figure 1 View of matmul

Name	Mode	Type	Bound	Comment
clk_i	input	wire		Clock signal for the design
rst_n_i	input	wire		Reset signal, active low
psel_i	input	wire		APB select
penable_i	input	wire		APB enable
pwrite_i	input	wire		APB write enable
pstrb_i	input	wire	[MAX_DIM-1:0]	APB write strobe (’byte’ select)
pwdata_i	input	wire	[BUS_WIDTH-1:0]	APB write data
paddr_i	input	wire	[ADDR_WIDTH-1:0]	APB address
pready_o	output	wire		APB slave ready
pslverr_o	output	wire		APB slave error
prdata_o	output	wire	[BUS_WIDTH-1:0]	APB read data
busy_o	output	wire		Busy signal, indicating the design cannot be written to

Table 1: matmul interface

1.1.1 APB Slave

The module uses defined states that are crucial for managing the APB communication, particularly focusing on write enable (W_ENABLE), read enable (R_ENABLE), and idle (IDLE) phases, with the BUSY state reflecting the core's activity status during matrix operations.

Key signals integral to the module's functionality include the APB protocol signals (psel_i, penable_i, pwrite_i, paddr_i, pwdata_i) and specific signals for the matrix multiplication process like ov_i for overflow indication, busy_o as an output signal indicating the core's busy status, and data signals (operand_A_o, operand_B_o, operand_C_o) for transferring the matrices. The module acts as a mediator, decoding APB commands into operations on the matmul core, ensuring data is correctly handled between the core and the APB master through these well-defined signals and states.

The state machine within APB Slave is tailored to efficiently manage transitions between operational modes, ensuring data integrity and timely response to APB requests. This involves transitioning from idle to either write or read enable states, based on the APB transaction type, and moving to the busy state when the matrix multiplication is active. This module has a critical role in translating the APB's protocol-driven communication into actionable operations for the matrix multiplication core, ensuring seamless data flow and operational control in compliance with the AMBA™ 4 APB Protocol v2.0 [2].

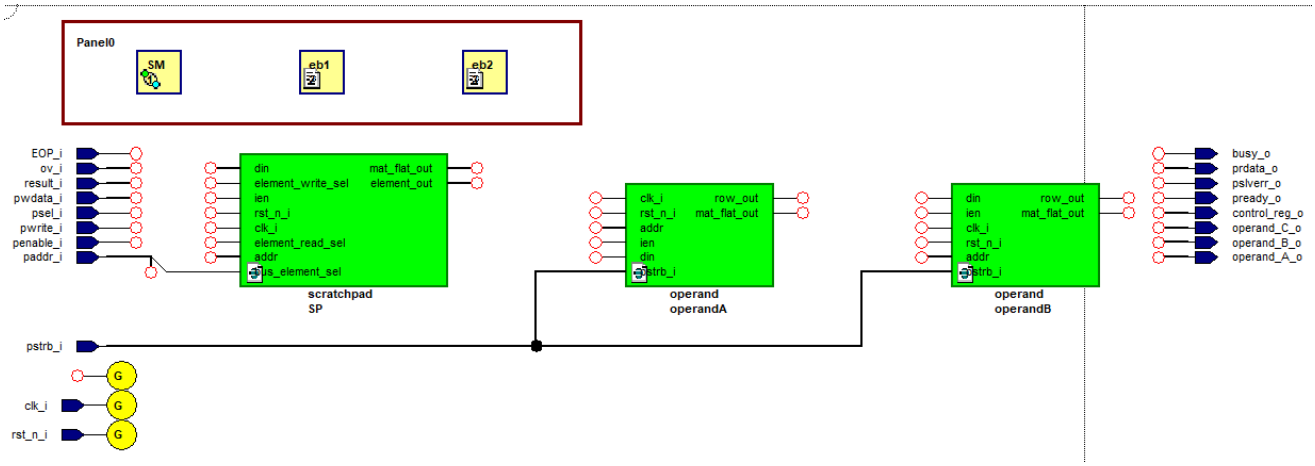


Figure 2 View of APB Slave

Name		Type	Bound	Comment
clk_i	input	wire		Clock signal for the design
rst_n_i	input	wire		Reset signal, active low
psel_i	input	wire		APB select
penable_i	input	wire		APB enable
pwrite_i	input	wire		APB write enable
pstrb_i	input	wire	[MAX_DIM-1:0]	APB write strobe (’byte’ select)
pwdata_i	input	wire	[BUS_WIDTH-1:0]	APB write data
paddr_i	input	wire	[ADDR_WIDTH-1:0]	APB address
ov_i	input	wire	[MAX_DIM*MAX_DIM-1:0]	Overflow vector from calculations
EOP_i	input	wire		End of process signal
result_i	input	wire	[BUS_WIDTH*MAX_DIM*MAX_DIM-1:0]	
pready_o	output	wire		APB slave ready
pslverr_o	output	wire		APB slave error
prdata_o	output	wire	[BUS_WIDTH-1:0]	APB read data
busy_o	output	wire		Busy signal, indicating the design cannot be written to
operand_A_o	output	wire	[BUS_WIDTH*MAX_DIM-1:0]	Matrix A from memory
operand_B_o	output	wire	[BUS_WIDTH*MAX_DIM-1:0]	Matrix B from memory
operand_C_o	output	reg	[BUS_WIDTH*MAX_DIM*MAX_DIM-1:0]	Matrix C from SP
control_reg_o	output	wire	[15:0]	Contains all control bits

Table 2: APB Slave interface

1.1.1.1 Operand Memory

The Operand Memory module is designed as a versatile component within the matrix multiplication project, primarily functioning as a memory interface for storing and retrieving operand A & B matrices.

At its core this module implements a memory structure tailored to store the operands of the matrix multiplication operation. It leverages Verilog's generate and always blocks to dynamically allocate memory for each element of the operand matrices, facilitating parallel read and write operations. This design choice enhances data handling efficiency, enabling rapid access to matrix elements required during the multiplication process.

Key to the module's functionality are signals for clock (clk_i), data input (din), write enable (ien), and address (addr), alongside the partial strobe signal (pstrb_i) which controls the writing to specific sections of the memory. These signals ensure data integrity and synchronization with the system's clock, thereby optimizing the flow of data into and out of the memory structure.

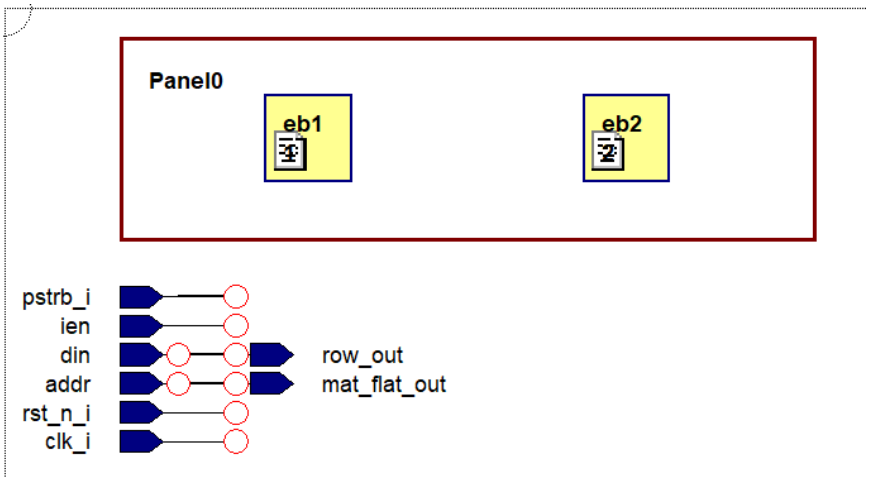


Figure 3 View of Operand Memory

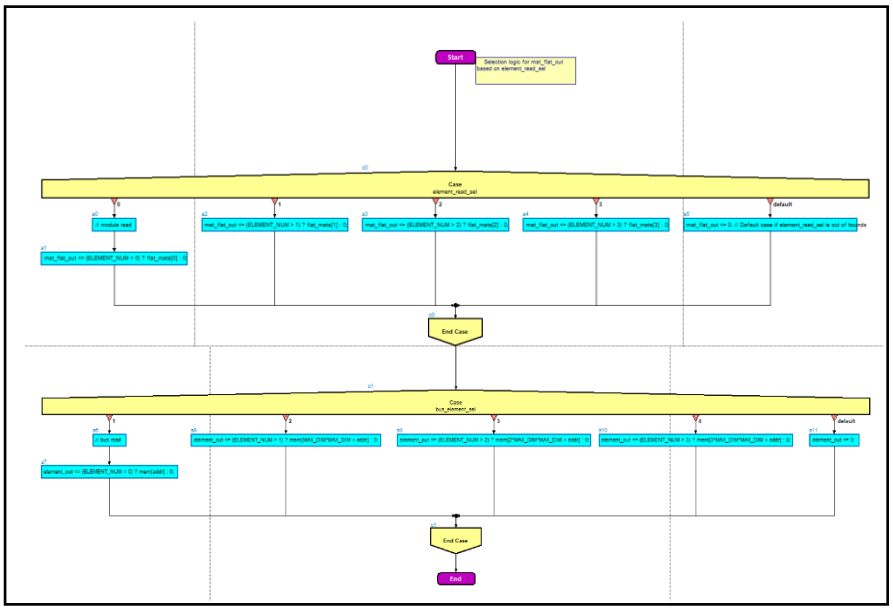
Name	Mode	Type	Bound	Comment
clk_i	input	wire		Clock signal for the design
rst_n_i	input	wire		Reset signal, active low
addr	input	wire	[1:0]	Address from bus
din	input	wire		Data in
ien	input	wire		Write enable
pstrb_i	input	wire	[MAX_DIM-1:0]	Strobe byte select
mat_flat_out	output	wire	[DATA_WIDTH*MAX_DIM*MAX_DIM - 1:0]	Entire matrix output
row_out	output	wire	[DATA_WIDTH*MAX_DIM - 1:0]	Entire row output

Table 3 Operand Memory interface

1.1.1.2 Scratchpad

The Scratchpad module is designed as a specialized memory component within the matrix multiplication project, aimed at providing temporary storage and efficient retrieval of computation results (bias mode operation) or specific matrix elements (bus read operation).

Key signals in the module include a clock input (clk_i), a reset signal (rst_n_i), a write enable signal (write_en), and data input (data_in) for updating memory contents. Additionally, a multiplexer-like selection mechanism, driven by the sel signal, allows for the dynamic selection of data sections within the scratchpad memory, facilitating the parallel handling of multiple data streams or computation stages within the matrix multiplication process.



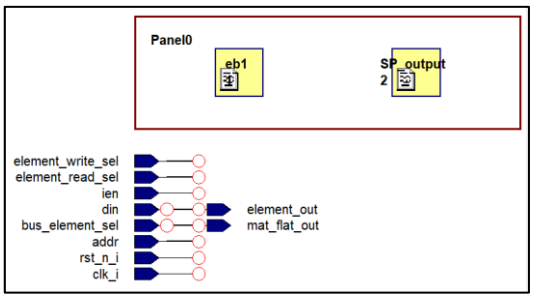


Figure 4 View of Scratchpad

Name	Mode	Type	Bound	Comment
clk_i	input	wire		Clock signal for the design
rst_n_i	input	wire		Reset signal, active low
addr	input	wire	[ADDR_WIDTH-1:0]	Address from bus
bus_element_sel	input	wire	[3:0]	
din	input	wire	[BUS_WIDTH - 1:0]	Data in
ien	input	wire		Write enable
element_read_sel	input	wire	[1:0]	Select which element to read
element_write_sel	input	wire	[1:0]	Select which element to write
mat_flat_out	output	wire	[BUS_WIDTH*MAX_DIM*MAX_DIM - 1:0]	Entire matrix output
element_out	output	wire	[BUS_WIDTH - 1:0]	Single element output

Table 4 Scratchpad interface

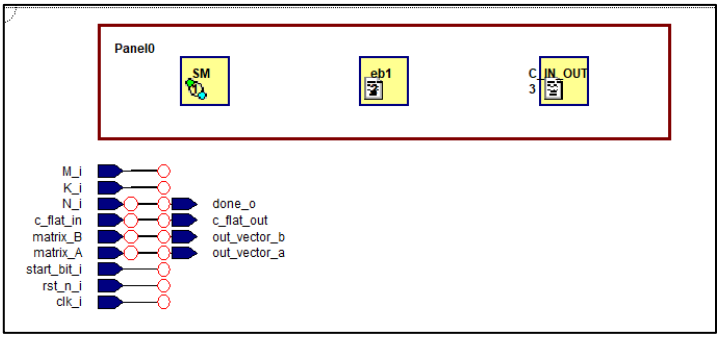
1.1.2 Shifter

The shifter module is intricately designed to synchronously input matrix elements into the systolic array, a key operation in matrix multiplication that ensures the timely and orderly presentation of data to the computational elements. It uses the dimensions of the matrices, specified by the control register, to correctly align and distribute matrix elements in preparation for their processing by the systolic array architecture.

Key to its operation, the module employs a clock signal (clk_i), a reset signal (rst_n_i), and control inputs that dictate the shifting operations. The module processes input matrices represented as flattened arrays (a_flat_in, b_flat_in), applying shifting logic to align matrix elements according to the requirements of the multiplication algorithm.

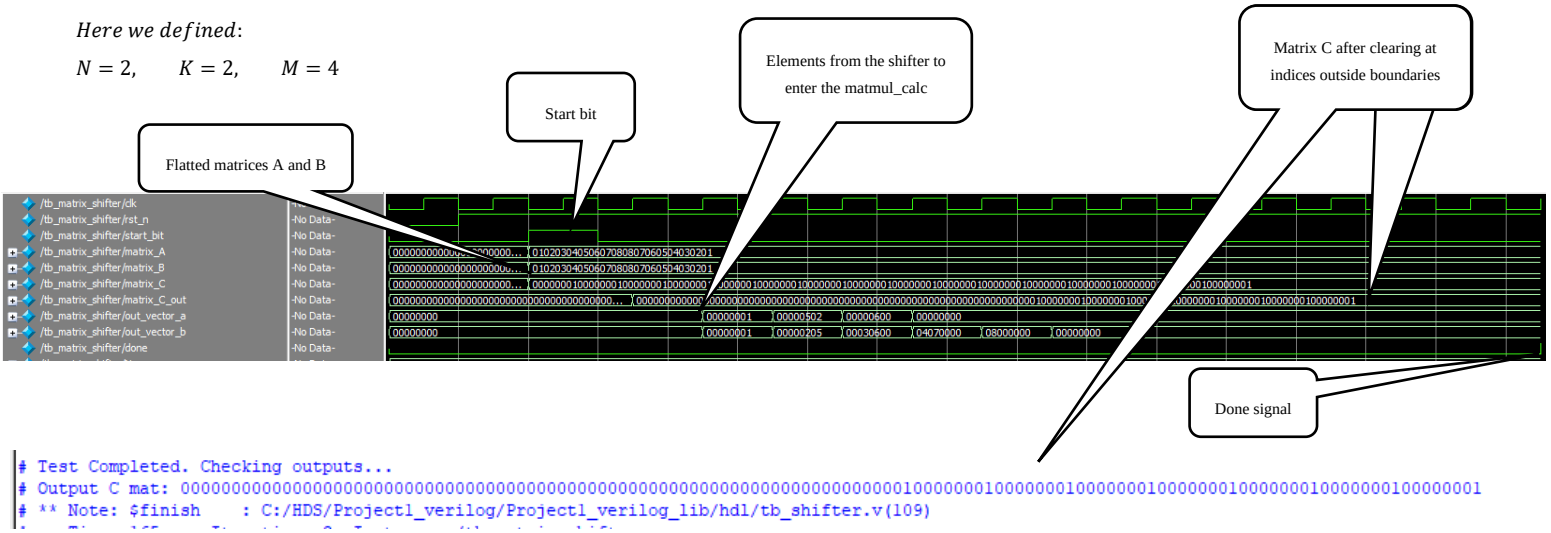
Moreover, shifter module includes logic to output a done signal upon the completion of the systolic array's processing. This signal serves as a synchronization point for the system, indicating that the current matrix multiplication operation has finished, and the results are ready for retrieval or further processing.

```
// Flatten 2D matrices into 1D vectors
// Matrix A
matrix_A = {8'd1, 8'd2, 8'd3, 8'd4,
            8'd5, 8'd6, 8'd7, 8'd8,
            8'd8, 8'd7, 8'd6, 8'd5,
            8'd4, 8'd3, 8'd2, 8'd1};

// Matrix B
matrix_B = {8'd1, 8'd2, 8'd3, 8'd4,
            8'd5, 8'd6, 8'd7, 8'd8,
            8'd8, 8'd7, 8'd6, 8'd5,
            8'd4, 8'd3, 8'd2, 8'd1};


// Matrix C
matrix_C = {32'd1, 32'd1, 32'd1, 32'd1,
            32'd1, 32'd1, 32'd1, 32'd1,
            32'd1, 32'd1, 32'd1, 32'd1,
            32'd1, 32'd1, 32'd1, 32'd1};
```



Here we defined:

$N = 2, \quad K = 2, \quad M = 4$




```
# Test Completed. Checking outputs...
# Output C mat: 00000000000000000000000000000000000000000000000000000000000000000000000000000000000000010000000100000001000000010000000100000001000000010000000100000001
# ** Note: $finish    : C:/HDS/Project1_verilog/Project1_verilog_lib/hdl/tb_shifter.v(109)
```

Figure 5 View of Shifter and test bench results

Name		Type	Bound	Comment
clk_i	input	wire		Clock signal for the design
rst_n_i	input	wire		Reset signal, active low
start_bit_i	input	wire		Start process signal
matrix_A	input	wire	[(MAX_DIM*MAX_DIM*DATA_WIDTH)-1:0]	Entire matrix A from memory
matrix_B	input	wire	[(MAX_DIM*MAX_DIM*DATA_WIDTH)-1:0]	Entire matrix B from memory
c_flat_in	input	wire	[(BUS_WIDTH*MAX_DIM*MAX_DIM)-1:0]	Entire matrix C from memory
N_i	input	wire	[1:0]	Number of rows in Mat A
K_i	input	wire	[1:0]	Number of cols/rows in Mat A/B
M_i	input	wire	[1:0]	Number of cols in Mat B
out_vector_a	output	reg	[(MAX_DIM*DATA_WIDTH)-1:0]	Current Elements A to be proccessed
out_vector_b	output	reg	[(MAX_DIM*DATA_WIDTH)-1:0]	Current Elements B to be proccessed
c_flat_out	output	reg	[(BUS_WIDTH*MAX_DIM*MAX_DIM)-1:0]	Matrix C after clearing unused values
done_o	output	reg		Shifting done flag

Table 5 Shifter interface

1.1.3 Matmul Calculator

The matmul calculator module is engineered to handle the multiplication of matrices in a flattened format, accommodating the parallel processing requirements and optimizing computational efficiency.

At its core, the module operates under the control of a clock signal (clk_i), ensuring that all operations are synchronized with the system's clock. The reset signal (rst_n_i), active low, allows the system to return to a known state, ensuring reliability and predictability in operations. The start_operation_i input initiates the matrix multiplication process, signaling the module to begin processing the input matrices. Internally, the module employs a systolic array architecture to perform the matrix multiplication.

The inputs a_flat_i, b_flat_i represent the matrices to be multiplied in a flattened form, , and c_flat_i is the bias matrix, The result_flat_o output delivers the multiplication result in a similar flattened format (or the multiplication result + matrix C in bias mode), facilitating easy integration with other system components. An overflow indicator (ov_reg_o) provides a critical check on the computation, signaling any arithmetic overflow that may occur during the process.

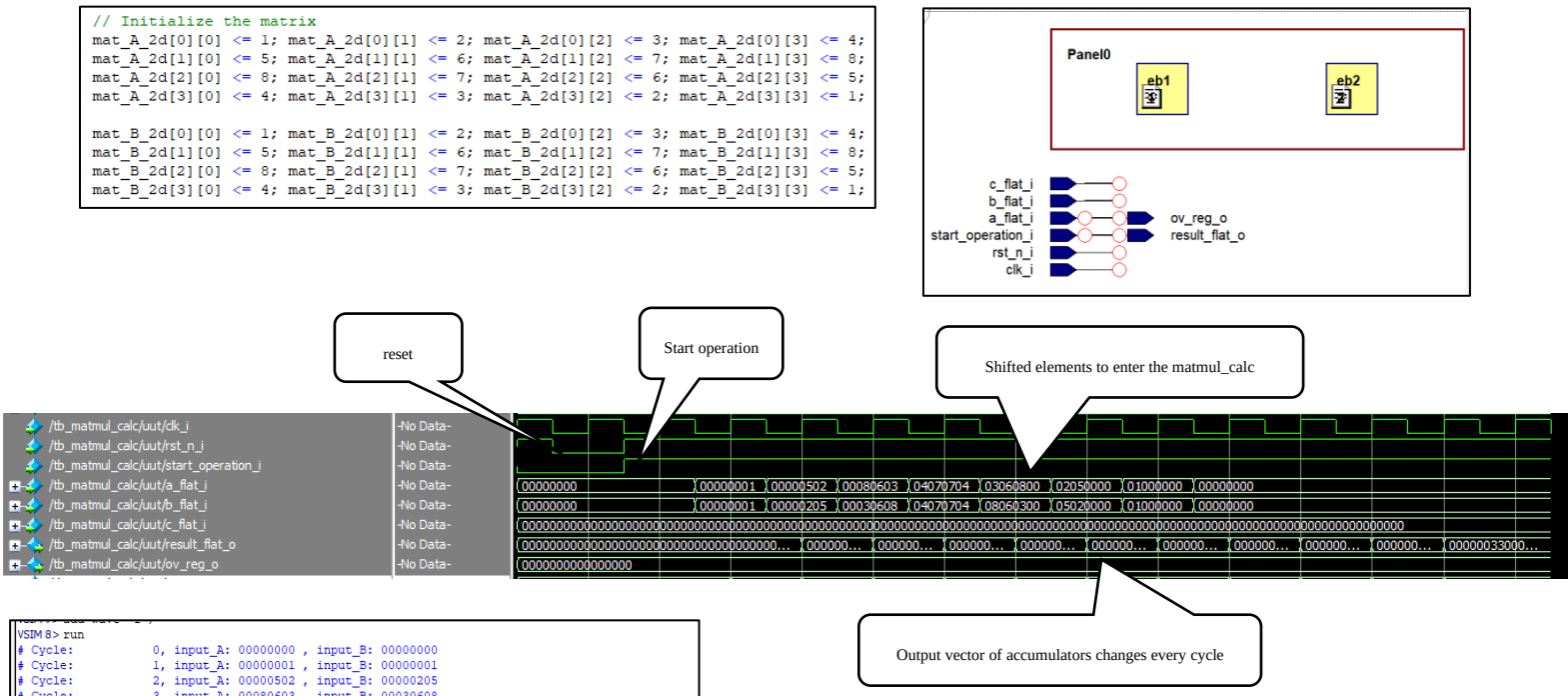


Figure 6 View of Matmul Calculator and test bench results

Name		Type	Bound	Comment
clk_i	input	wire		Clock signal for the design
rst_n_i	input	wire		Reset signal, active low
start_operation_i	input	wire		Start send data to systolic array
a_flat_i	input	wire	[DATA_WIDTH*MAX_DIM-1:0]	Current Elements A to be proccessed
b_flat_i	input	wire	[DATA_WIDTH*MAX_DIM-1:0]	Current Elements B to be proccessed
c_flat_i	input	wire	[BUS_WIDTH*MAX_DIM*MAX_DIM-1:0]	Matrix C after clearing unused values
result_flat_o	output	reg	[BUS_WIDTH*MAX_DIM*MAX_DIM-1:0]	Calculated output from all PE's
ov_reg_o	output	reg	[MAX_DIM*MAX_DIM-1:0]	PE's calculation overflow vector

Table 6 Matmul Calculator interface

1.1.3.1 Processing Element

The PE module is designed to perform core computations involved in matrix multiplication, such as multiplying input operands and accumulating results.

The PEs are linked (data_A_o and data_B_o) in a manner that allows for the sequential processing of matrix elements. This design ensures efficient data flow and minimizes the need for external memory accesses, thereby enhancing computational speed and efficiency.

The module's design includes logic to handle arithmetic operations, data forwarding, and overflow detection. Like other modules, the start_operation_i input triggers the PE to operate.

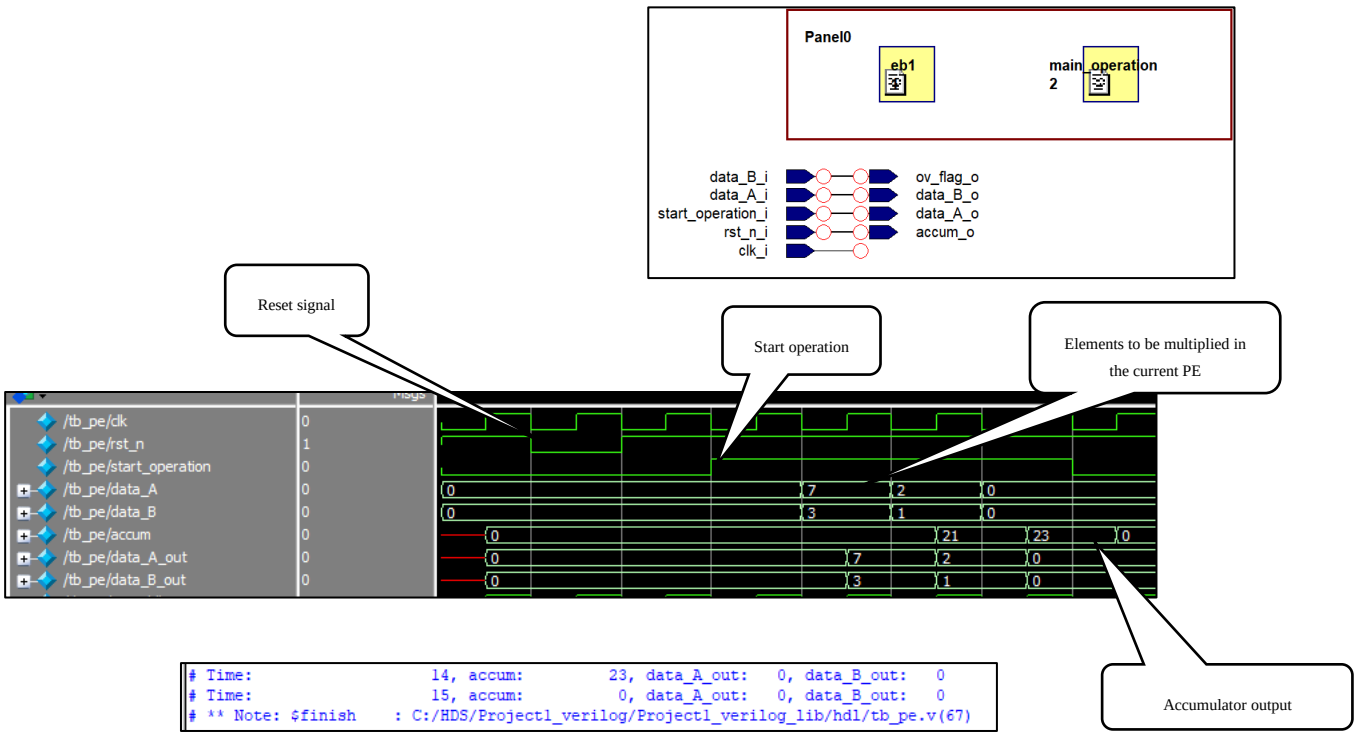
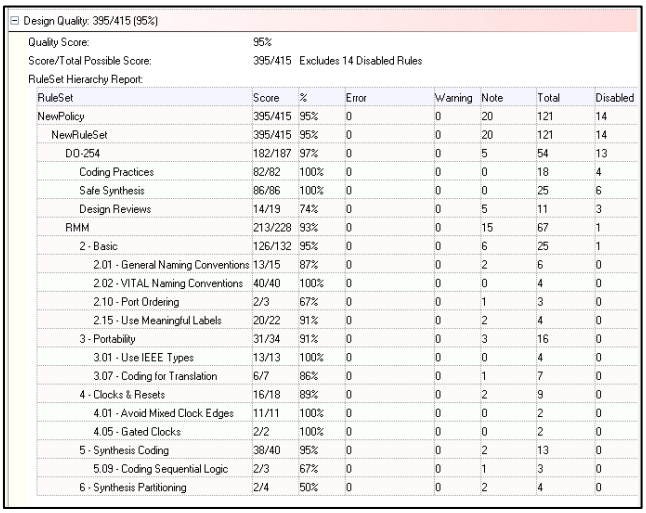


Figure 7 View of Processing Element and test bench results

Name		Type	Bound	Comment
clk_i	input	wire		Clock signal for the design
rst_n_i	input	wire		Reset signal, active low
start_operation_i	input	wire		Start send data to systolic array
data_A_i	input	wire	[DATA_WIDTH-1:0]	Current Element A to be multiply
data_B_i	input	wire	[DATA_WIDTH-1:0]	Current Element B to be multiply
data_A_o	output	reg	[DATA_WIDTH-1:0]	Previous Element A goes to next unit
data_B_o	output	reg	[DATA_WIDTH-1:0]	Previous Element B goes to next unit
accum_o	output	reg	[BUS_WIDTH-1:0]	Acumulated calculation
ov_flag_o	output	reg		Overflow from acumulation or multiplication

Table 7 Processing Element interface

2. RULES AND POLICIES



Design Quality: 395/415 (95%)

Quality Score: 95%

Score/Total Possible Score: 395/415 Excludes 14 Disabled Rules

RuleSet Hierarchy Report:

RuleSet	Score	%	Error	Warning	Note	Total	Disabled
NewPolicy	395/415	95%	0	0	20	121	14
NewRuleSet	395/415	95%	0	0	20	121	14
DO-254	182/187	97%	0	0	5	54	13
Coding Practices	82/82	100%	0	0	0	18	4
Safe Synthesis	86/86	100%	0	0	0	25	6
Design Reviews	14/19	74%	0	0	5	11	3
RMM	213/228	93%	0	0	15	67	1
2 - Basic	126/132	95%	0	0	6	25	1
2.01 - General Naming Conventions	13/15	87%	0	0	2	6	0
2.02 - VITAL Naming Conventions	40/40	100%	0	0	0	4	0
2.10 - Port Ordering	2/3	67%	0	0	1	3	0
2.15 - Use Meaningful Labels	20/22	91%	0	0	2	4	0
3 - Portability	31/34	91%	0	0	3	16	0
3.01 - Use IEEE Types	13/13	100%	0	0	0	4	0
3.07 - Coding for Translation	6/7	86%	0	0	1	7	0
4 - Clocks & Resets	16/18	89%	0	0	2	9	0
4.01 - Avoid Mixed Clock Edges	11/11	100%	0	0	0	2	0
4.05 - Gated Clocks	2/2	100%	0	0	0	2	0
5 - Synthesis Coding	38/40	95%	0	0	2	13	0
5.09 - Coding Sequential Logic	2/3	67%	0	0	1	3	0
6 - Synthesis Partitioning	2/4	50%	0	0	2	4	0

1. Avoid Unused Declarations
- Justification: Some times few bits are irrelevant in some modules and unused.

2. Ensure Consistent FSM State Encoding Style
- Justification: We learned in class that we can define states as parameters.

3. Avoid Mismatching Ranges
- Justification: Integer assignments can create carry. For example: i = i+1 in for loops. But we won't even get to a high value required carry consideration...

4. Avoid Hard-Coded Numeric Values
- Justification: Sometimes we need to use hard coded numeric values, for example, if we have a constant amount of cases, a constant amount of possible N values and so on...

5. Ensure Proper Case Statement Specification
- Justification: We cover all cases, so the default case is not used. Also, we cannot remove the default case since it gives an error of a missing default block.

6. Avoid Asynchronous Reset Release
- Justification: We want to use asynchronous reset.

7. Avoid Undriven & Unused Logic
- Justification: Some times few bits are irrelevant in some modules and unused.

8. Ensure Register Controllability
- Justification: It is easier and more readable to use M K and N as integers, even though they contain small values...

9. Ensure Nesting Limits
- Justification: We think that 3-level nesting is also legitimate and still looking elegant.

10. Avoid Unressetable Register
- We don't nessesarily want to initialize the memory during reset.

11. Ensure Consistent File Header
- Justification: We don't know how to define the file header correctly.

12. Avoid Using Tab
- Justification: We just want to use tabs.

13. Ensure Sufficient Comment Density
- Justification: 50% comments is just too much. We even wrote module description but it still wasn't enough. We wrote as much comments as needed to understand every little piece of the code.

14. Use a separate line for each statement
- Justification: We don't want new lines every time we use for loops (that contain ':' symbol).

3. APPENDIX

3.1 Terminology

LSB - Least Significant Bit

FSM - Finite state machine

3.2 References

[1] H.T. Kung (1982) Why Systolic Architectures?

[2] AMBA™ 4 APB Protocol v2.0