STMicroelectronics

AI Accelerators Hands-on HW Design

Jul.2025

life.augmented

LAB 0

Systolic Array for Applying Matrix Multiplication

By: Youssef Sameh Fawzy

Instructor: Ahmed Abdelsalam

Content:

1- Architecture:

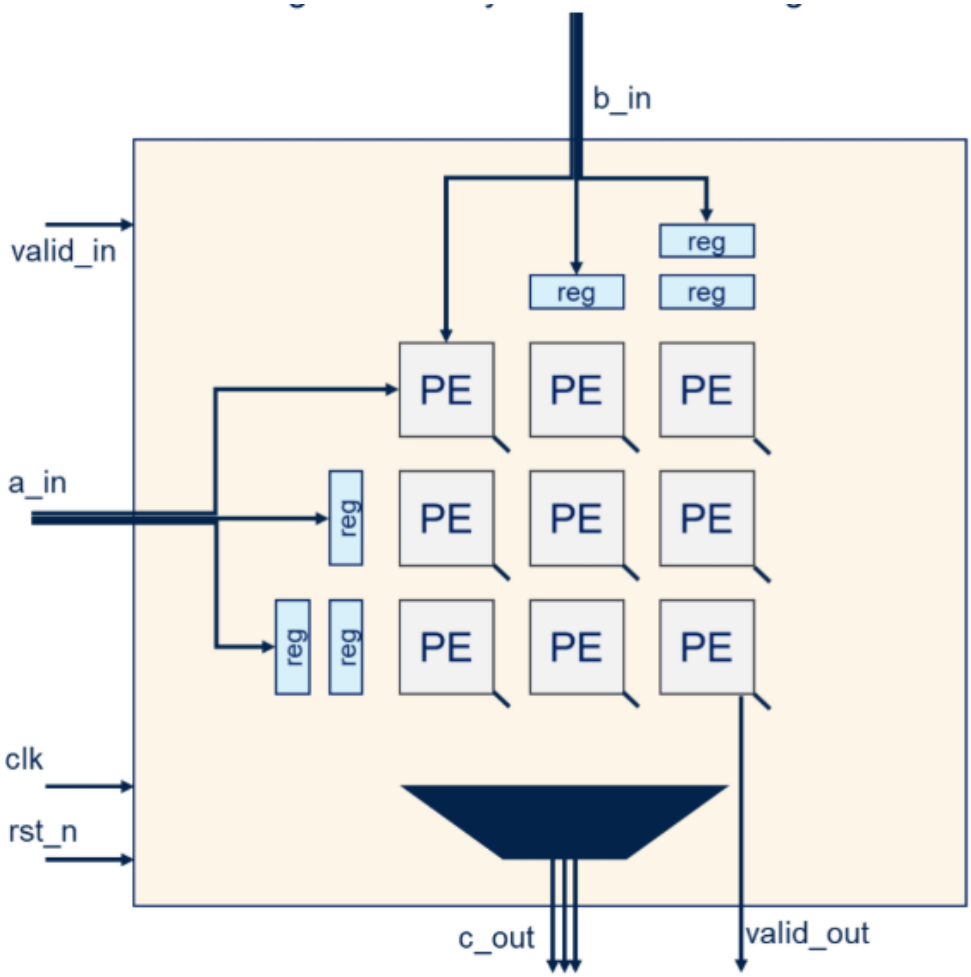
First , We need to understand how our Systolic array works.

Every clock cycle it takes a column from Matrix A , and a row from Matrix B and enter them on a grid of PEs (processing element). Every PE makes it's calculation then pass it's input to the following PE. The A input goes from left to right and that of B from up to down. The input A or B will pass on a register depending on the row or coloumn it goes in. The first part of the input goes directly to the PE but the next one will pass through a register first so it will be delayed a cycle.This will insure that the previous PE has completed its operation and passed the output completely.

The output is taken on Matrix C as rows, when the first row is ready it gets to the output, and when the second row is ready it overwrites the previous row. We have a signal called "valid_out" that indicates when the output is available to take.

There is another signal called "valid_in" that indicates whether we will take the input from matrix A and B or not.

We have an internal counter that will help us indicating when the valid_out will be ready.



This is a visualization of how it works , but with each PE has output connected to the right PE and down PE to pass the value of A and B.

2-RTL Codes:

At first , I did a fixed 3x3 Systolic array to implement on what I have learned.

I noticed blocks being used a lot in the implementation which are the registers and the Processing elements so I did them in a separate modules.

```
module PE#(parameter data_size=8)(
input wire rst_n,clk,
input wire [data_size-1:0] in_a,in_b,
output reg [2*data_size-1:0] out_c,
output reg [data_size-1:0] out_a,out_b
);
always @(posedge clk)begin
if(!rstn_n) begin
out_a <= 0;
out_b <= 0;
out_c <= 0;
end else begin
out_c <= out_c + in_a*in_b;
out_a <=in_a;
out_b <=in_b;
end
end
endmodule
```

The PE module simply took input from the left and from up and multiply them then adds it to out_c which accumulate each cycle and that will be the out of this element of the matrix.

```
module DFF #(parameter data_size = 8)(
    input clk,
    input rst_n,
    input [data_size-1:0] d,
    output reg [data_size-1:0] q
);
    always @(posedge clk) begin
        if (!rst_n) begin
            q <= 0;
        end else begin
            q <= d;
        end
    end
endmodule
```

The register module simply takes d and after a clk cycle it passes it to q, this will delay the input as we expects.

Then we will instantiate them all in the systolic array module.

Systolic3x3:

fixed3x3systolic.sv > systolic3x3

```
1   module systolic3x3#(parameter data_size=8)(
2   input clk,rstn_n,
3   input [data_size*3-1:0] matrix_a_in,
4   input [data_size*3-1:0] matrix_b_in,
5   output reg valid_out,
6   output reg [data_size*3*2-1:0] matrix_c_out
7   );
8       reg [3:0] counter1;
9   always @(posedge clk) begin
10      if (!rstn_n) begin
11          counter1 <= 0;
12      end else begin
13          if (counter1 < 7) counter1 <= counter1 + 1;
14      end
15  end
16
17  always @(posedge clk) begin
18      if (!rstn_n)
19          valid_out <= 0;
20      else begin
21          valid_out <= (counter1 >= 4 && counter1 <= 6);
22      end
23  end
```

First we declare the inputs and we initiate a counter.

As it is a 3x3 array we already know when the output will be ready at cycle 5,6 and 7 so we made valid_out = 1 at that time.

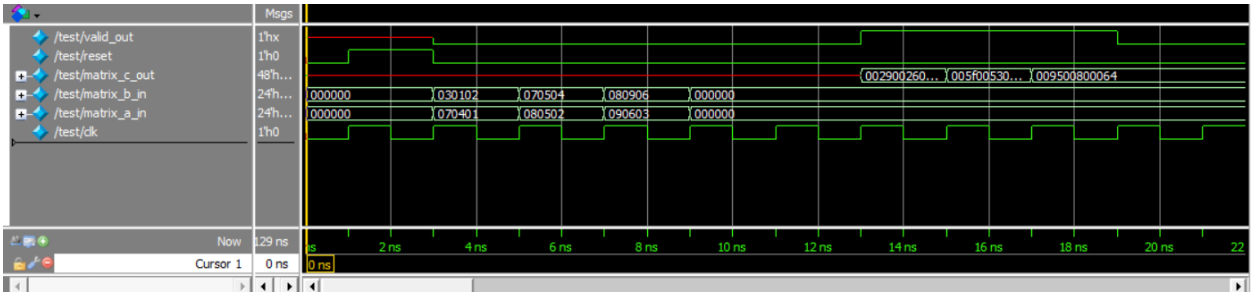
```
//Registers to delay the input
DFF dff1(.clk(clk), .rst_n(rstn_n), .d(matrix_a_in[data_size*2-1:data_size]), .q(a2_reg));
DFF dff2(.clk(clk), .rst_n(rstn_n), .d(matrix_a_in[data_size*3-1:data_size*2]), .q(a3_reg1));
DFF dff3(.clk(clk), .rst_n(rstn_n), .d(a3_reg1), .q(a3_reg2));
DFF dff4(.clk(clk), .rst_n(rstn_n), .d(matrix_b_in[data_size*2-1:data_size]), .q(b2_reg));
DFF dff5(.clk(clk), .rst_n(rstn_n), .d(matrix_b_in[data_size*3-1:data_size*2]), .q(b3_reg1));
DFF dff6(.clk(clk), .rst_n(rstn_n), .d(b3_reg1), .q(b3_reg2));

wire [2*data_size-1:0] c1,c2,c3,c4,c5,c6,c7,c8,c9;
PE PE1 (.clk(clk), .rstn_n(rstn_n), .in_a(matrix_a_in[data_size-1:0]), .in_b(matrix_b_in[data_size-1:0]), .out_a(a1to2), .out_b(b1to4),
 .out_c(c1));
PE PE2 (.clk(clk), .rstn_n(rstn_n), .in_a(a1to2), .in_b(b2_reg), .out_a(a2to3), .out_b(b2to5), .out_c(c2));
PE PE3 (.clk(clk), .rstn_n(rstn_n), .in_a(a2to3), .in_b(b3_reg2), .out_a(), .out_b(b3to6), .out_c(c3));
PE PE4 (.clk(clk), .rstn_n(rstn_n), .in_a(a2_reg), .in_b(b1to4), .out_a(a4to5), .out_b(b4to7), .out_c(c4));
PE PE5 (.clk(clk), .rstn_n(rstn_n), .in_a(a4to5), .in_b(b2to5), .out_a(a5to6), .out_b(b5to8), .out_c(c5));
PE PE6 (.clk(clk), .rstn_n(rstn_n), .in_a(a5to6), .in_b(b3to6), .out_a(), .out_b(b6to9), .out_c(c6));
PE PE7 (.clk(clk), .rstn_n(rstn_n), .in_a(a3_reg2), .in_b(b4to7), .out_a(a7to8), .out_b(), .out_c(c7));
PE PE8 (.clk(clk), .rstn_n(rstn_n), .in_a(a7to8), .in_b(b5to8), .out_a(a8to9), .out_b(), .out_c(c8));
PE PE9 (.clk(clk), .rstn_n(rstn_n), .in_a(a8to9), .in_b(b6to9), .out_a(), .out_b(), .out_c(c9));

   always @(*) begin
          case (counter1)
              5: begin
                   matrix_c_out[2*data_size-1:0] = c1; matrix_c_out[data_size*4-1:2*data_size] = c2; matrix_c_out[data_size*6-1:data_size*4]
              end
              6: begin
                   matrix_c_out[2*data_size-1:0] = c4; matrix_c_out[data_size*4-1:2*data_size] = c5; matrix_c_out[data_size*6-1:data_size*4]
              end
              7: begin
                   matrix_c_out[2*data_size-1:0] = c7; matrix_c_out[data_size*4-1:2*data_size] = c8; matrix_c_out[data_size*6-1:data_size*4]
              end
          endcase
   end
```

As we know it's a 3x3 we will only instantiate the number of DFF and PE we need , we made some wires called c from 1 to 9 to get the output on them till they are ready and we give them to the mat_c_out. At last as we know the output will be ready at cycle 5,6 and 7.

Then we used a tb and directed values we know the expect of them.

```
module test;

reg clk;
reg reset;
reg [23:0] matrix_a_in, matrix_b_in;
reg valid_in;

wire [47:0] matrix_c_out;


systolic3x3 uut (
.clk(clk),
.reset(reset),
.matrix_a_in(matrix_a_in),
.matrix_b_in(matrix_b_in),
.valid_out(valid_out),
.matrix_c_out(matrix_c_out)
);

initial begin
reset = 0;
    matrix_a_in = 24'h000000; // Initialize to zero
    matrix_b_in = 24'h000000; // Initialize to zero
@(posedge clk); reset = 1;
@(posedge clk); reset = 0;
    matrix_a_in = 24'h070401; // Example values for matrix A
    matrix_b_in = 24'h030102; // Example values for matrix B
    @(posedge clk);
       matrix_a_in = 24'h080502; // Example values for matrix A
       matrix_b_in = 24'h070504; // Example values for matrix B
       @(posedge clk);
       matrix_a_in = 24'h090603; // Example values for matrix A
       matrix_b_in = 24'h080906; // Example values for matrix B
       @(posedge clk); // Clear valid input after processing
        matrix_a_in = 24'h000000;
       matrix_b_in = 24'h000000;

       repeat(10) @(posedge clk); // Wait for some cycles to observe outputs


#100;
$stop;
end

```

We ran it and it gave us correct output with correct timing of valid_out.



This is not enough as we ignored valid_in and we need to parametrize the input array size

Parameterized Systolic Array:

We will do the same as we did for the 3x3 but for every dimensions of the matrix.

parameterizedsystolic.sv > systolic_array

```
1   module systolic_array #(
2   parameter DATAWIDTH = 16,
3   parameter N_SIZE = 5)(
4   input clk,rst_n,valid_in,
5   input [N_SIZE*DATAWIDTH-1:0] matrix_a_in,matrix_b_in,
6   output reg valid_out,
7   output reg [N_SIZE*2*DATAWIDTH-1:0] matrix_c_out
8   );
9
10  reg [N_SIZE] clk_counter;
11  always@(posedge clk or negedge rst_n) begin
12      if(!rst_n) begin
13          clk_counter <= 0;
14      end
15      else clk_counter <= clk_counter + 1;
16  end
17
```

First we declared the inputs and outputs as specified then we made a counter the counts the clock cycles as it will be important in next operations.

```
wire [DATAWIDTH-1:0] a_reg[0:N_SIZE-1];
wire [DATAWIDTH-1:0] b_reg[0:N_SIZE-1];
assign a_reg[0]= (valid_in) ? matrix_a_in[DATAWIDTH-1 : 0] : {DATAWIDTH{1'b0}};
assign b_reg[0]= (valid_in) ? matrix_b_in[DATAWIDTH-1 : 0] : {DATAWIDTH{1'b0}};

genvar i, j;
generate
    for (i = 1; i < N_SIZE; i = i + 1) begin : A_PIPE
        wire [DATAWIDTH-1:0] a_input = (valid_in) ? matrix_a_in[(i+1)*DATAWIDTH-1 -: DATAWIDTH] : {DATAWIDTH{1'b0}};
        DFF #(.DATAWIDTH(DATAWIDTH), .NUM_DFF(i)) A(
            clk,
            rst_n,
            a_input,
            a_reg[i]
        );
    end

    for (j = 1; j < N_SIZE; j = j + 1) begin : B_PIPE
        wire [DATAWIDTH-1:0] b_input = (valid_in) ? matrix_b_in[(j+1)*DATAWIDTH-1 -: DATAWIDTH] : {DATAWIDTH{1'b0}};
        DFF #(.DATAWIDTH(DATAWIDTH), .NUM_DFF(j)) B(
            clk,
            rst_n,
            b_input,
            b_reg[j]
        );
    end
endgenerate
```

We now want to parameterize the number of registers before each PE on the first row and coloumn. For a_reg[0] and b_reg[0] it will be assigned directly as the least part of the matrixA and matrixB inputs as we need no regs for them to enter the PE.

Number of registers is a function in N_SIZE, so we made a variable i and for each increament we increase the number of DFF by 1.

If valid_in is 1 all regs will take its part of the input but if not they will take a 0.

```
module DFF #(parameter DATAWIDTH = 16, NUM_DFF = 1) (
    input clk,
    input wire rst_n,
    input wire [DATAWIDTH-1:0] d,
    output wire [DATAWIDTH-1:0] q
);
    reg [DATAWIDTH-1:0] stage_array [0:NUM_DFF-1];
    integer i;

    always @(posedge clk or negedge rst_n) begin
        if (!rst_n) begin
            for (i = 0; i < NUM_DFF; i = i + 1)
                stage_array[i] <= '0;
        end else begin
            stage_array[0] <= d;
            for (i = 1; i < NUM_DFF; i = i + 1)
                stage_array[i] <= stage_array[i - 1];
        end
    end
    assign q = stage_array[NUM_DFF - 1];
endmodule
```

That's the DFF module we used.

It's idea is that we put the input d in an array and we shift it each clock cycle to the left till we reach the value of NUM_DFF we want and the q output will take it after NUM_DFF clk cycle.

```
wire   [2*DATAWIDTH-1:0] c [0:N_SIZE-1][0:N_SIZE-1];
wire   [DATAWIDTH-1:0] a_bet_pe     [0:N_SIZE-1][0:N_SIZE-1];
wire   [DATAWIDTH-1:0] b_bet_pe      [0:N_SIZE-1][0:N_SIZE-1];


generate
    for (i = 0; i < N_SIZE; i = i + 1) begin
    for (j = 0; j < N_SIZE; j = j + 1) begin
 PE #(.data_size(DATAWIDTH)) pe(clk,rst_n,(j == 0) ? a_reg[i] : a_bet_pe[i][j-1] ,(i == 0) ? b_reg[j] : b_bet_pe[i-1][j]
    , c[i][j], a_bet_pe[i][j],b_bet_pe[i][j]);
        end
    end
endgenerate
```

We first declared the internal wires between every pe and each other and the output of it is assigned to a wire C. a_bet_pe and b_bet_pe are 2-D to represent the internal wires between PE.

If the pe is in the first coloumn we take a_reg[i] directly but if not we take from the previous a_bet_pe and the same for B if it's first row we take from b_reg and if not we take from the previous(upove) b_between_pe.

```
module PE#(parameter data_size=16)(
input wire clk,rst_n,
input wire [data_size-1:0] in_a,in_b,
output reg [2*data_size-1:0] out_c,
output reg [data_size-1:0] out_a,out_b
);
always @(posedge clk or negedge rst_n)begin
if(!rst_n) begin
out_a <= 0;
out_b <= 0;
out_c <= 0;
end else begin
out_c <= out_c + in_a*in_b;
out_a <=in_a;
out_b <=in_b;
end
end
endmodule
```

This is the PE module. We take the 2 inputs a and b and multiply it then we accumulate it to the value of out_c we have already. Then we pass the 2 inputs to the following PE.

```
integer out_col;
always @(*) begin
    if (clk_counter >= 2*N_SIZE - 1 && clk_counter <= 3*N_SIZE - 2) begin
        valid_out = 1'b1;
        for (out_col = 0; out_col < N_SIZE; out_col = out_col + 1) begin
            matrix_c_out[(out_col+1)*2*DATAWIDTH-1 -: 2*DATAWIDTH] =
                c[clk_counter - (2*N_SIZE - 1)][out_col];
        end
    end else begin
        matrix_c_out = 'b0;
        valid_out = 1'b0;
    end
end
```

Finally, We get the output, it's always expected that the output rows will be ready in every clk from the range between 2*N_SIZE-1 to 3*N_SIZE -2. At clk_counter = 2*N_SIZE -1 we will get row 1 till the final row will be ready at 3*N_SIZE-2.

The matrix_c_out will take the rows 1 by 1 from the C wires which is the output of every PE.

We make the rows fixed as it is always = clk_counter – (2*N_SIZE-1) and iterate through the coloumn.

Matrix_C_out will take all c from the row and put them from MSB to LSB.

We make valid_out = 0 and reset the matrix_c_out after all the rows are got on the output.

3 – Simulation:

- **I made a testbench and tried 5x5 multiplication with 16 bit width:**

$$A = \begin{bmatrix} 1 & 2 & 3 & 4 & 5 \\ 6 & 7 & 8 & 9 & 10 \\ 11 & 12 & 13 & 14 & 15 \\ 16 & 17 & 18 & 19 & 20 \\ 21 & 22 & 23 & 24 & 25 \end{bmatrix}$$

$$B = \begin{bmatrix} 26 & 27 & 28 & 29 & 30 \\ 31 & 32 & 33 & 34 & 35 \\ 36 & 37 & 38 & 39 & 40 \\ 41 & 42 & 43 & 44 & 45 \\ 46 & 47 & 48 & 49 & 50 \end{bmatrix}$$

It decimal result:

$$C = \begin{bmatrix} 590 & 605 & 620 & 635 & 650 \\ 1490 & 1530 & 1570 & 1610 & 1650 \\ 2390 & 2455 & 2520 & 2585 & 2650 \\ 3290 & 3380 & 3470 & 3560 & 3650 \\ 4190 & 4305 & 4420 & 4535 & 4650 \end{bmatrix}$$

What we expect on the waveform:

$$\begin{bmatrix} 0x24E & 0x25D & 0x26C & 0x27B & 0x28A \\ 0x5D2 & 0x5FA & 0x622 & 0x64A & 0x672 \\ 0x956 & 0x997 & 0x9D8 & 0xA19 & 0xA5A \\ \mathbf{0xCD2} & 0xD34 & 0xD8E & 0xDE8 & 0xE42 \\ 0x105E & 0x10D1 & 0x1144 & 0x11B7 & 0x122A \end{bmatrix}$$

Tb Code:

C: > Users > Sameh Fawzy > Desktop > youssefsameh_01288215640 > simu > systolic_array_tb.sv

```
module systolic_array_tb;
    parameter DATAWIDTH = 16;
    parameter N_SIZE = 5;

    reg clk;
    reg rst_n;
    reg valid_in;
    reg [N_SIZE*DATAWIDTH-1:0] matrix_a_in;
    reg [N_SIZE*DATAWIDTH-1:0] matrix_b_in;
    wire valid_out;
    wire [N_SIZE*2*DATAWIDTH-1:0] matrix_c_out;

    // DUT instantiation
    systolic_array #(
        .DATAWIDTH(DATAWIDTH),
        .N_SIZE(N_SIZE)
    ) dut (
        .clk(clk),
        .rst_n(rst_n),
        .valid_in(valid_in),
        .matrix_a_in(matrix_a_in),
        .matrix_b_in(matrix_b_in),
        .valid_out(valid_out),
        .matrix_c_out(matrix_c_out)
    );

    // Clock generation
    initial begin
        clk = 0;
        forever #1 clk = ~clk;
    end

```

- **First we generate the clock.**

```
    initial begin
        rst_n = 0;
        valid_in = 0;
        @(negedge clk);
        rst_n = 1;
$display("Input Matrix A =");
$display(" Row 1 : 1  2  3  4  5");
$display(" Row 2 : 6  7  8  9 10");
$display(" Row 3 :11 12 13 14 15");
$display(" Row 4 :16 17 18 19 20");
$display(" Row 5 :21 22 23 24 25");
$display("Input Matrix B =");
$display(" Row 1 : 26 27 28 29 30");
$display(" Row 2 : 31 32 33 34 35");
$display(" Row 3 : 36 37 38 39 40");
$display(" Row 4 : 41 42 43 44 45");
$display(" Row 5 : 46 47 48 49 50");
        // Apply input rows (Matrix A and B as rows/columns)
        valid_in = 1;

        matrix_a_in = 80'h00150010000b00060001;
        matrix_b_in = 80'h001e001d001c001b001a;
        @(negedge clk);
        matrix_a_in = 80'h00160011000c00070002;
        matrix_b_in = 80'h0023002200210020001f;
        @(negedge clk);


        matrix_a_in = 80'h00170012000d00080003;
        matrix_b_in = 80'h0028002700260025 0024;
        @(negedge clk);


        matrix_a_in = 80'h00180013000e00090004;
        matrix_b_in = 80'h002d002c002b002a0029;
        @(negedge clk);


        matrix_a_in = 80'h00190014000f000a0005;
        matrix_b_in = 80'h0032003100300002f002e;
        @(negedge clk);
```

We enter matrix A and B as coloumns and rows.

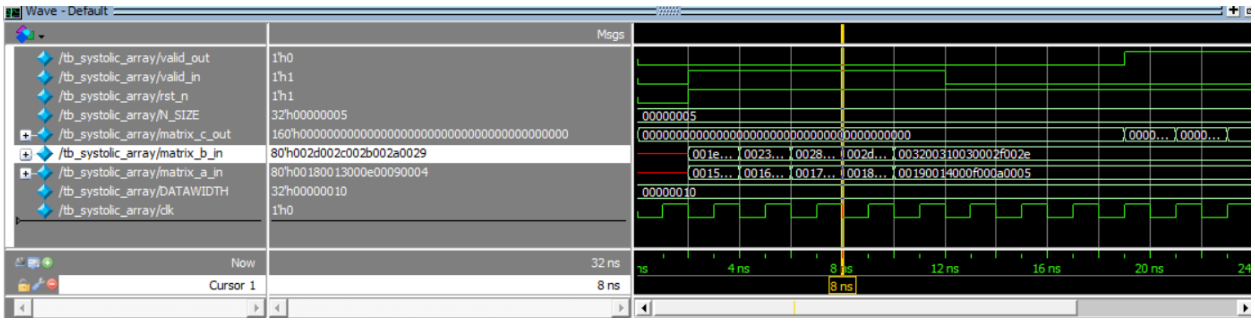
Note: That we enter every row or coloumn from left to right and from down to up so the LSB in mata or b is the first element.

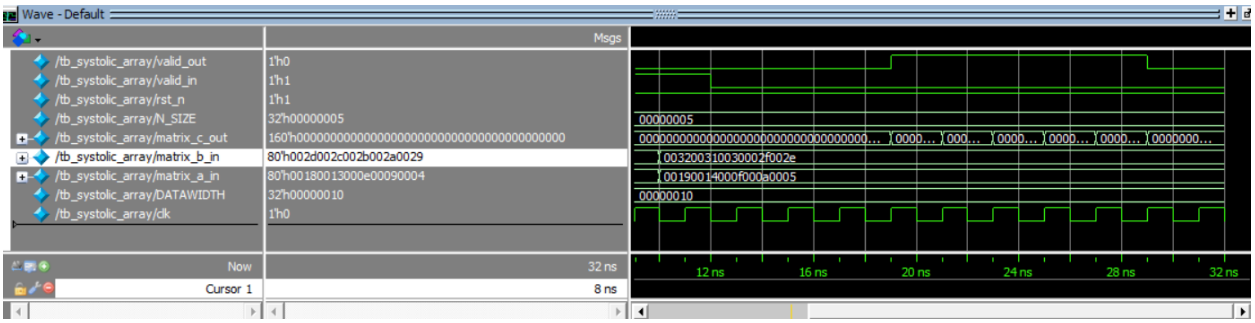
```
        valid_in = 0;

        repeat(4) @(negedge clk);
        $display("output matrix C : ");
        $display("");
$write("Row:1 ");
for (int j = 0; j < N_SIZE; j++) begin
    $write("%0d ", matrix_c_out[j*DATAWIDTH*2 +: DATAWIDTH*2]);
end
$display("");

@(negedge clk);
$write("Row:3 ");
for (int j = 0; j < N_SIZE; j++) begin
    $write("%0d ", matrix_c_out[j*DATAWIDTH*2 +: DATAWIDTH*2]);
end
$display("");
@(negedge clk);
$write("Row:3 ");
for (int j = 0; j < N_SIZE; j++) begin
    $write("%0d ", matrix_c_out[j*DATAWIDTH*2 +: DATAWIDTH*2]);
end
$display("");
@(negedge clk);
$write("Row:4 ");
for (int j = 0; j < N_SIZE; j++) begin
    $write("%0d ", matrix_c_out[j*DATAWIDTH*2 +: DATAWIDTH*2]);
end
$display("");
@(negedge clk);
$write("Row:5 ");
for (int j = 0; j < N_SIZE; j++) begin
    $write("%0d ", matrix_c_out[j*DATAWIDTH*2 +: DATAWIDTH*2]);
end
$display("");
        $display("=== End of Output ===");
        $stop;
    end
endmodule
```

We then close the valid_in and check the output every cycle and print it.

Note: the code is provided inside the zip file and its simulation log

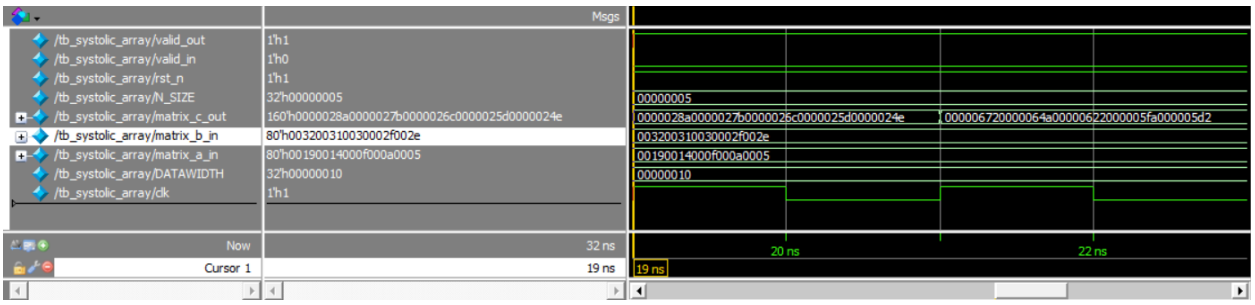
Wave Form:



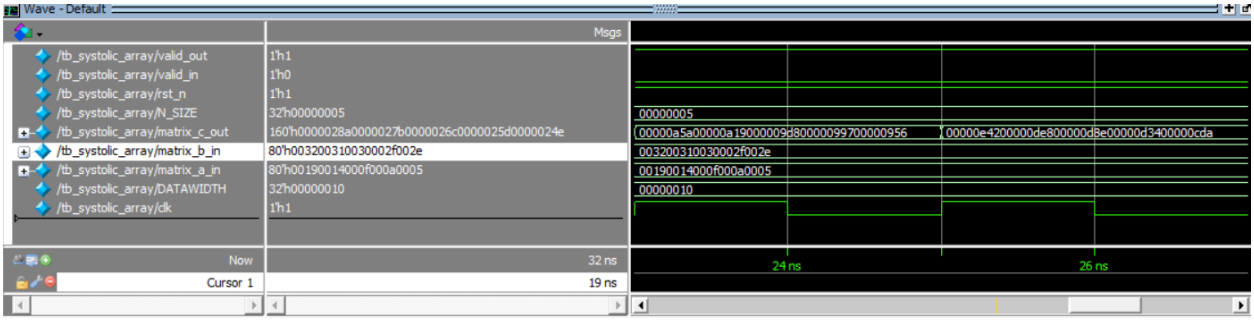


Outputs are:

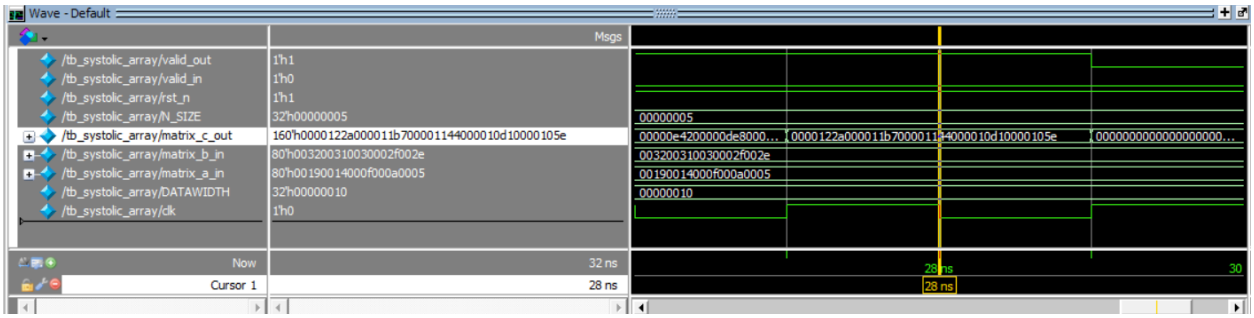
First 2 rows



Second 2 rows



Last row



All are the values we expect.

Output in the transcript:

```
# Input Matrix A =
#  Row 1 : 1  2  3  4  5
#  Row 2 : 6  7  8  9 10
#  Row 3 :11 12 13 14 15
#  Row 4 :16 17 18 19 20
#  Row 5 :21 22 23 24 25
# Input Matrix B =
#  Row 1 : 26 27 28 29 30
#  Row 2 : 31 32 33 34 35
#  Row 3 : 36 37 38 39 40
#  Row 4 : 41 42 43 44 45
#  Row 5 : 46 47 48 49 50
# output matrix C :
#
# Row:1 590 605 620 635 650
# Row:3 1490 1530 1570 1610 1650
# Row:3 2390 2455 2520 2585 2650
# Row:4 3290 3380 3470 3560 3650
# Row:5 4190 4305 4420 4535 4650
```

4- What is missing?:

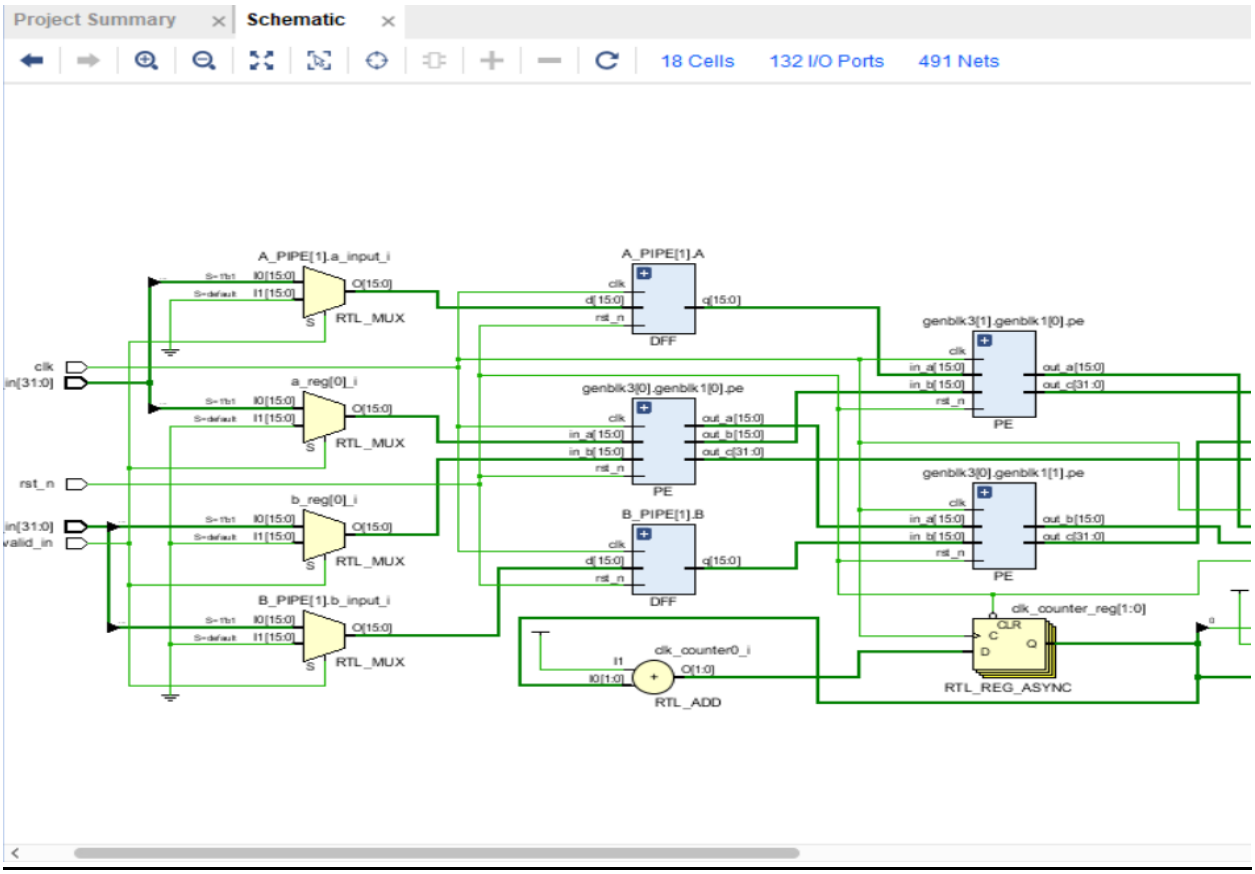
I assumed valid_in is like in the gif sent to us in the email and that it will take value 1 for N_SIZE clk cycles then it will be 0 and wait for out. However, I contacted engineer Ahmed and he said that it is intended that valid_in will be custom and randomly drived and that if it's 0 we will not take the input if it's 1 we will take it. This is not implemented in my code as I made it systematic to that the valid in will always be opened in the first cycles and will close at N_size cycle.

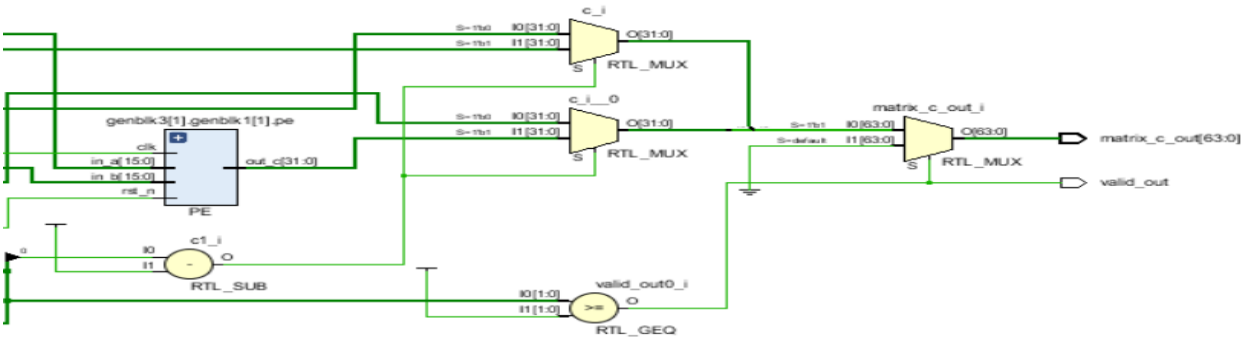
We don't handle the case when for example we enter 2 inputs then valid_in = 0 then we open it again , it will give wrong output.

To solve this problem I tried to stop the counter when valid_in is low but it made a bigger problem with the output , if we find a way that when valid_in = 0 the whole design will stop working then return from where it stopped , it will be solved.

5- Vivado Elaboration Schematic:

Before I generated the schematic I made N_SIZE = 2 so we can see it easily





For a 2x2 matrix multiplication we expect 4 PE blocks and 2 DFF and that's what we are seeing.