MATRIX MULTIPLIER ACCELERATOR

PS-2

Team 10 :Naman Goyal (230002046)

:Sarvadnyee Ghogare(230002065)

Project Task:

[FOR FINAL SUBMISSION]
Designing a Matrix Multiplication Accelerator in Verilog HDL:

Description:
- Design an efficient matrix multiplication accelerator for binary matrices with 4-bit values as inputs, using only **gate-level description** in Verilog HDL. The accelerator should be capable of processing two input matrices of **maximum size 3x3**, with a total **maximum of 18 inputs**, **each being 4 bits**.

Specifications:
- The accelerator should take inputs in a serial manner over a maximum of 18 clock cycles and produce outputs in a serialised fashion, providing one output at a time.
- The primary objective is to design a matrix multiplication accelerator that minimises the number of clock cycles required to compute the results efficiently.
- Additionally, the solution should be scalable and adaptable to different matrix sizes while ensuring accurate computation of the product.

TIMELINE OF PROJECT WORK:

[JUNE]

JUNE: 1ST WEEK

- ❑ Understanding basics of Verilog language and its application , why it is used ,other hardware descriptive languages, what is VLSI ,what are simulation and synthesis tools . Also ,learned about data types , variables and logic operations used in Verilog.

JUNE: 2ND WEEK

- ❑ Covered basics of digital electronics mainly focused on theory part , what is digital electronics , difference in digital and analog circuits ,Understanding usage of basic components of combinational circuits like gates,decoders , MUX, adders , etc .

JUNE: 3RD WEEK

- ❑ Practiced and worked on impelementing Verilog codes for Components of Combinational circuits learned till now . Learned how to write a testbench and how to use all procedural assignment statements in main code & testbench , learned usage of sequential statements such as if else , case , repeat , forever etc .Each time implementing codes &testbenches with new ideas to cover up basics from root and to build strong concepts .

JUNE: 4TH WEEK

- ❑ Started wIth formulating the main code for matrix multiplier of sizes of 3X3 only and also tried to match up with other specifications of project , with the help of instantiating adders and multiplier and also prepared testbench for it. Also started to work on ppt for mid eval submission .

JULY

2ND JULY

- ✓ MID-EVAL SUBMISSION

JULY: 1ST WEEK

- ❑ Covering theory of sequential circuits of digital electronics ,what are they ,difference between combinational and sequential circuits .Understanding usage of basic components of sequential circuits like flip flop , registers and its types, latches ,counters,etc.

JULY: 2ND WEEK

- ❑ Practiced impelementing for Components of Sequential circuits learned until now. Designing of testbench for each and learned how to produce clock cycles and reset inputs ..Also continued the theory part of sequential circuits.

JULY: 3RD WEEK

- ❑ Formulating the main code for matrix multiplier for different sizes and also completing other specifications of the project . Instantiated both sequential and combinational components , and also prepared a testbench . Finalizing the Matrix Multiplier code and also done with completion of PPT to demonstrate everything regarding this project.

27TH JULY

- ✓ FINAL SUBMISSION OF THE PROJECT

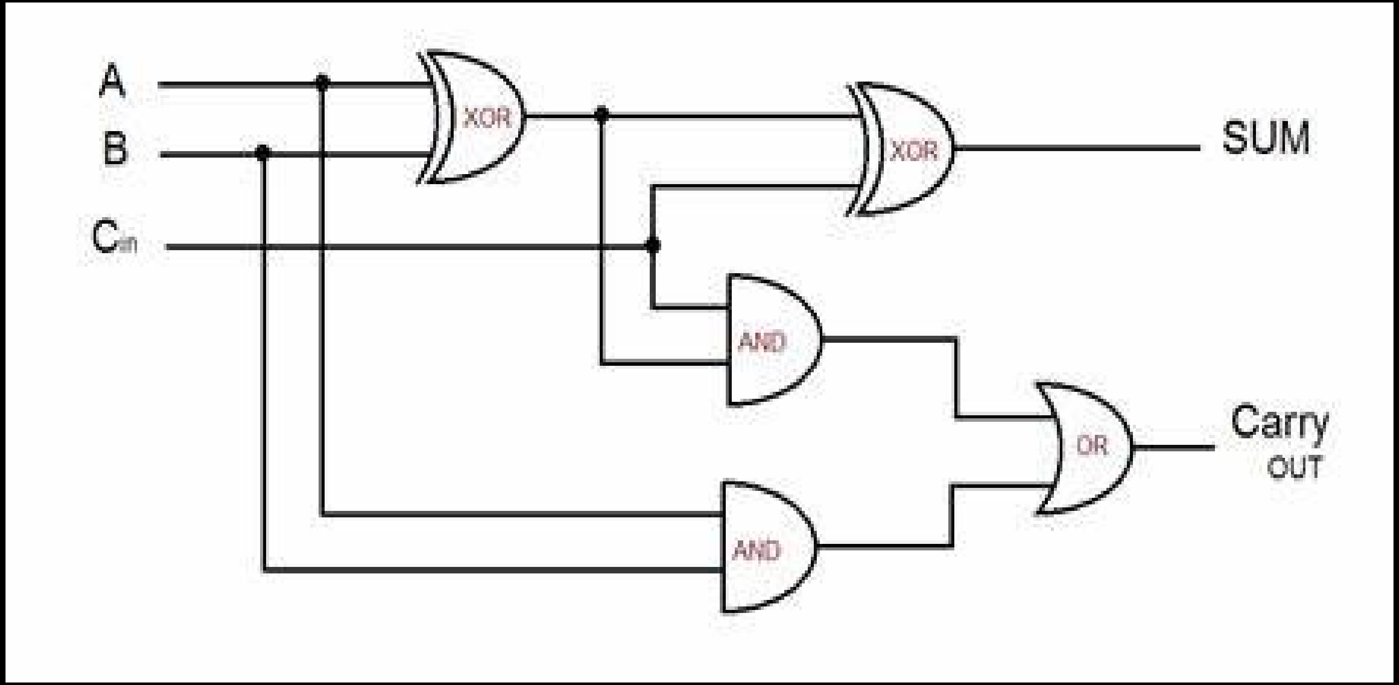
MID EVAL SUBMISSION

Underlying Principle: [FOR PROJECT TILL MID-EVAL]

- This project implements a matrix multiplier of size 3x3 and 3x3 (FOR MID-EVAL SUBMISSION) in Verilog using gate-level description with 18 elements in total and each 3x3 containing 9 elements each of size 4 BITS.
- We have used 9 clock cycles , one clock cycle for each output element at at time.
- It includes modules of following for generating matrix multiplier :
- Basic gates (AND , OR , XOR, NOT).(Predefined)
- Full adder.
- 4-bit Adder.
- 8-bit Adder.
- 4-bit Multiplier.
- EXTRA-Examples of Test bench of 4 BIT ADDER & 4 BIT MULTIPLIER.
- Matrix Multiplier producing one output at a time over 9 clock cycles.
- Use of reset and clock inputs will be required.
- Testbench (for simulation).
- NOTE- ALL THESE CODES ARE ALSO THERE IN XILINX SHEET ATTACHED IN GITHUB ITSELF.

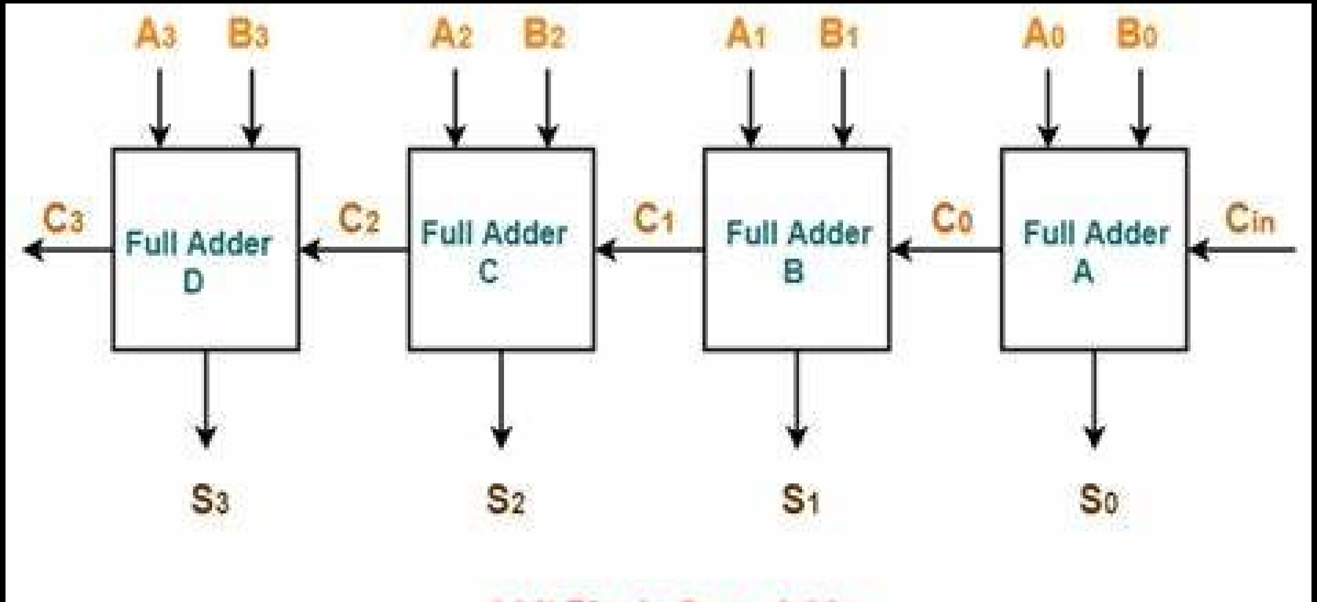
FULL ADDER

```
module Full_Adder(Sum,Cout,A,B,Cin);
input A,B,Cin;
output Sum,Cout;
   wire S1, C1, C2;
   xor (S1, A, B);
   xor (Sum, S1, Cin);
   and (C1, A, B);
   and (C2, S1, Cin);
   or  (Cout, C1, C2);
endmodule
```



4 BIT ADDER

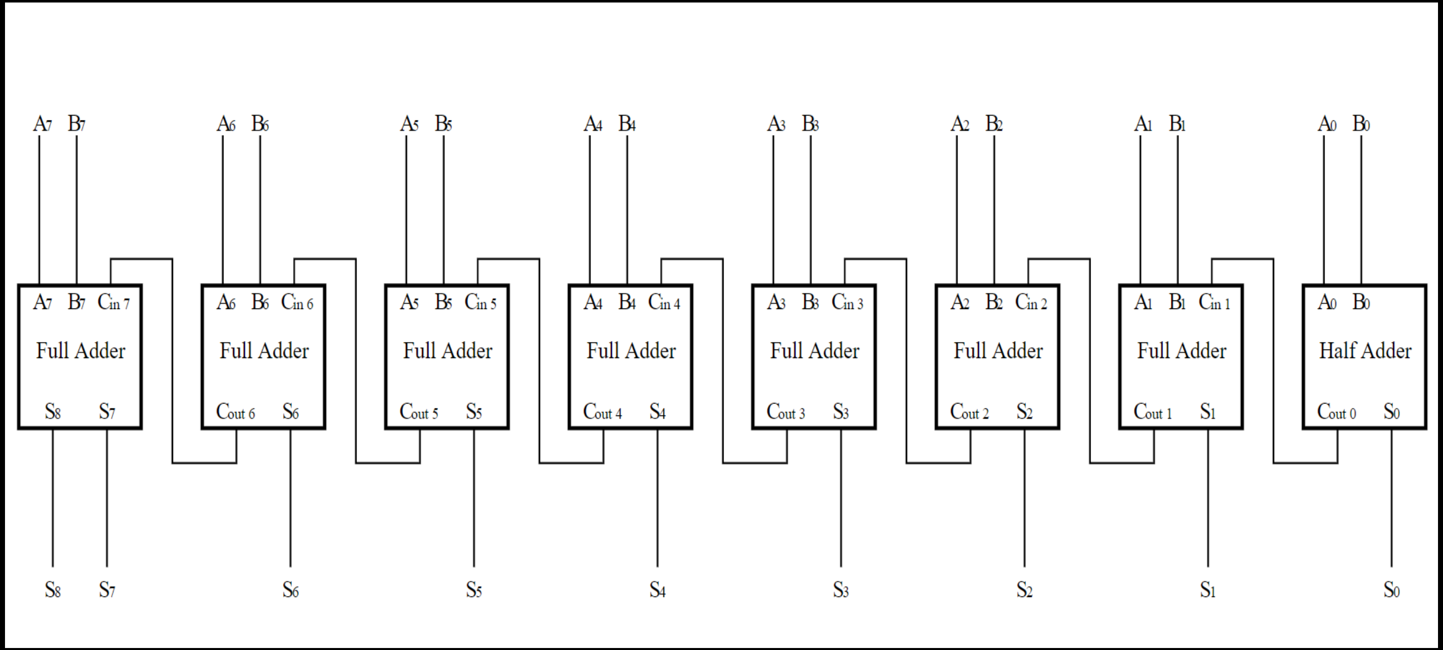
```
module adder_4bit(S, Cout,  A, B,  Cin);
output [3:0] S;
output Cout;
input[3:0] A,B;
input Cin;
wire C1, C2, C3;
//using instantiation

Full_Adder  FA0(S[0], C1, A[0], B[0], Cin);
Full_Adder FA1(S[1], C2, A[1], B[1], C1);
Full_Adder FA2(S[2], C3, A[2], B[2], C2);
Full_Adder FA3(S[3], Cout, A[3], B[3], C3);
endmodule
```



4-bit Ripple Carry Adder

8-BIT ADDER

```
module adder_8bit(S,Cout1,A,B,Cin);
input [7:0]A ,B;
input Cin;
output [7:0]S;
output Cout1;
 wire Cout2;
 adder_4bit
A0(S[3:0],Cout2,A[3:0],B[3:0],Cin);
 adder_4bit
A1(S[7:4],Cout1,A[7:4],B[7:4],Cout2);
endmodule
```


A7 B7 A6 B6 A5 B5 A4 B4 A3 B3 A2 B2 A1 B1 A0 B0
A7 B7 Cin 7 Full Adder S8 S7
A6 B6 Cin 6 Full Adder Cout 6 S6
A5 B5 Cin 5 Full Adder Cout 5 S5
A4 B4 Cin 4 Full Adder Cout 4 S4
A3 B3 Cin 3 Full Adder Cout 3 S3
A2 B2 Cin 2 Full Adder Cout 2 S2
A1 B1 Cin 1 Full Adder Cout 1 S1
A0 B0 Half Adder Cout 0 S0
S8 S7 S6 S5 S4 S3 S2 S1 S0

4 BIT-MULTIPLIER CODE

```
//USING 4 BIT ADDER AND FULL ADDER (AS GATE LEVEL DESCRIPTION)
module multiplier_4bit(a,b,outp);
   input [3:0] a;
   input [3:0] b;
   output [7:0]outp;
   wire [3:0] p0, p1, p2, p3,p4,p5,p6;
   wire [7:0] sum1, sum2, sum3;
   wire  cout1, cout2,cout3;
   wire S1,S2,S3,S4,S5,S6,COUT1,COUT2,COUT3,COUT4,COUT5;

   assign p0 = a & {4{b[0]}};
   assign p1 = a & {4{b[1]}};
   assign p2 = a & {4{b[2]}};
   assign p3 = a & {4{b[3]}};
   //HERE WE ARE USING CONCANTATION OPERATOR
   assign p4[3:0] ={ p1[2], p1[1], p1[0] ,1'b0};
   assign p5[3:0] ={ p2[1], p2[0],2'b00};
   assign p6[3:0] ={p3[0] ,3'b000};

adder_4bit adder1 (.A(p0), .B(p4), .Cin(1'b0), .S(sum1[3:0]), .Cout(cout1));
//this is another way we can instantiate
Full_Adder FA1(S1,COUT1,cout1,p1[3],0);
assign sum1[7:4] = {2'b0,COUT1,S1};

adder_4bit adder2 (.A(sum1[3:0]), .B(p5), .Cin(1'b0), .S(sum2[3:0]), .Cout(cout2));
Full_Adder FA2(S2,COUT2,cout2,sum1[4],p2[2]);
Full_Adder FA3(S3,COUT3,COUT2,sum1[5],p2[3]);
assign sum2[7:4] = {1'b0, COUT3,S3,S2};

adder_4bit adder3 (.A(sum2[3:0]), .B(p6), .Cin(1'b0), .S(sum3[3:0]), .Cout(cout3));
Full_Adder FA4(S4,COUT4,cout3,sum2[4],p3[1]);
Full_Adder FA5(S5,COUT5,COUT4,sum2[5],p3[2]);
Full_Adder FA6(S6,COUT6,COUT5,sum2[6],p3[3]);
assign sum3[7:4] = {COUT6,S6,S5,S4};
assign outp = sum3;

endmodule
```

Examples of TESTBENCH:

FOR 4-BIT ADDER

```
module TESTBENCH;
reg[3:0] A,B;
reg Cin;
wire [3:0]S;
wire Cout;
adder_4bit dut1(S, Cout,  A, B,  Cin);
initial
begin
A=4'b1111;B=4'b1010;Cin=1’b0;
#5
A=4'b1001;B=4'b1110 ;Cin=1’b1;
#5
$finish;
end
endmodule
```

FOR 4-BIT MULTIPLIIER

```
module TESTBENCH;
reg[3:0] A,B;
wire [7:0]S;
multiplier_4bit_2inp dut2(A,B,S);
initial
begin
A=4'b1111;B=4'b1010;
#5 //DELAY TIME
A=4'b1001;B=4'b1110;
#5
$finish;
end
endmodule
```

MATRIX MULTPLIER (3X3 EACH) CODE:-

[FOR MID-EVAL]

```
Module   MatrixMultiplier(clk,reset,a11,a12,a13,a21,a22,a23,a31,a32,a33,b11,b12,b13,
b21,b22,b23,b31,b32,b33,c12,c11,c13,c21,c22,c23,c31,c32,c33);

   input clk, reset;
   input [3:0] a11, a12, a13, a21, a22, a23, a31, a32, a33;
   input [3:0] b11,b12,b13,b21,b22,b23,b31,b32,b33;
   output reg [9:0] c11,c12,c13,c21,c22, c23, c31,  c32,  c33;
   reg [3:0] cycle_count;
   wire [7:0]MUL[0:26];
   wire [7:0]SUM[0:17];
   wire [0:26]Cout;
   wire [0:8]sum;
```

```
multiplier_4bit dut0(a11,b11,MUL[0]);
multiplier_4bit dut1(a12,b21,MUL[1]);
multiplier_4bit dut2(a13,b31,MUL[2]);
adder_8bit dut3(SUM[0],Cout[0],MUL[0],MUL[1],0);
adder_8bit dut4(SUM[1],Cout[1],SUM[0],MUL[2],0);
Full_Adder dut5(sum[0],Cout[2],Cout[1],Cout[0],0);

multiplier_4bit dut6(a11,b12,MUL[3]);
multiplier_4bit dut7(a12,b22,MUL[4]);
multiplier_4bit dut8(a13,b32,MUL[5]);
adder_8bit dut9(SUM[2],Cout[3],MUL[3],MUL[4],0);
adder_8bit dut10(SUM[3],Cout[4],SUM[2],MUL[5],0);
Full_Adder dut11(sum[1],Cout[5],Cout[4],Cout[3],0);

multiplier_4bit dut12(a11,b13,MUL[6]);
multiplier_4bit dut13(a12,b23,MUL[7]);
multiplier_4bit dut14(a13,b33,MUL[8]);
adder_8bit dut15(SUM[4],Cout[6],MUL[6],MUL[7],0);
adder_8bit dut16(SUM[5],Cout[7],SUM[4],MUL[8],0);
Full_Adder dut17(sum[2],Cout[8],Cout[7],Cout[6],0);
```

```
multiplier_4bit dut18(a21,b11,MUL[9]);
multiplier_4bit dut19(a22,b21,MUL[10]);
multiplier_4bit dut20(a23,b31,MUL[11]);
adder_8bit dut21(SUM[6],Cout[9],MUL[9],MUL[10],0);
adder_8bit dut22(SUM[7],Cout[10],SUM[6],MUL[11],0);
Full_Adder dut23(sum[3],Cout[11],Cout[10],Cout[9],0);

multiplier_4bit dut24(a21,b12,MUL[12]);
multiplier_4bit dut25(a22,b22,MUL[13]);
multiplier_4bit dut26(a23,b32,MUL[14]);
adder_8bit dut27(SUM[8],Cout[12],MUL[12],MUL[13],0);
adder_8bit dut28(SUM[9],Cout[13],SUM[8],MUL[14],0);
Full_Adder dut29(sum[4],Cout[14],Cout[13],Cout[12],0);

multiplier_4bit dut30(a21,b13,MUL[15]);
multiplier_4bit dut31(a22,b23,MUL[16]);
multiplier_4bit dut32(a23,b33,MUL[17]);
adder_8bit dut33(SUM[10],Cout[15],MUL[15],MUL[16],0);
adder_8bit dut34(SUM[11],Cout[16],SUM[10],MUL[17],0);
Full_Adder dut35(sum[5],Cout[17],Cout[16],Cout[15],0);
```

```
multiplier_4bit dut36(a31,b11,MUL[18]);
multiplier_4bit dut37(a32,b21,MUL[19]);
multiplier_4bit dut38(a33,b31,MUL[20]);
adder_8bit dut39(SUM[12],Cout[18],MUL[18],MUL[19],0);
adder_8bit dut40(SUM[13],Cout[19],SUM[12],MUL[20],0);
Full_Adder dut41(sum[6],Cout[20],Cout[19],Cout[18],0);

multiplier_4bit dut42(a31,b12,MUL[21]);
multiplier_4bit dut43(a32,b22,MUL[22]);
multiplier_4bit dut44(a33,b32,MUL[23]);
adder_8bit dut45(SUM[14],Cout[21],MUL[21],MUL[22],0);
adder_8bit dut46(SUM[15],Cout[22],SUM[14],MUL[23],0);
Full_Adder dut47(sum[7],Cout[23],Cout[22],Cout[21],0);

multiplier_4bit dut48(a31,b13,MUL[24]);
multiplier_4bit dut49(a32,b23,MUL[25]);
multiplier_4bit dut50(a33,b33,MUL[26]);
adder_8bit dut51(SUM[16],Cout[24],MUL[24],MUL[25],0);
adder_8bit dut52(SUM[17],Cout[25],SUM[16],MUL[26],0);
Full_Adder dut53(sum[8],Cout[26],Cout[25],Cout[24],0);
```

```
 always @(posedge clk or negedge reset)
    begin
       if (!reset)
       begin
           cycle_count = 0;
       end

       else
       begin
           case (cycle_count)
               0:c11={Cout[2],sum[0],SUM[1]};
               1:c12={Cout[5],sum[1],SUM[3]};
               2:c13={Cout[8],sum[2],SUM[5]};
               3:c21={Cout[11],sum[3],SUM[7]};
               4:c22={Cout[14],sum[4],SUM[9]};
               5:c23={Cout[17],sum[5],SUM[11]};
               6:c31={Cout[20],sum[6],SUM[13]};
               7:c32={Cout[23],sum[7],SUM[15]};
               8:c33={Cout[26],sum[8],SUM[17]};
           endcase

           if (cycle_count < 9)
               cycle_count = cycle_count + 1;
       end
   end
endmodule
```

TESTBENCH

```
module Testbench;
    reg clk, reset;
    reg [3:0] a11, a12, a13, a21, a22, a23, a31, a32, a33;
    reg [3:0] b11,b12,b13,b21,b22,b23,b31,b32,b33;
    wire [9:0] c11,c12,c13,c21,c22, c23, c31,  c32,  c33;

MatrixMultiplier
DUT(clk,reset,a11,a12,a13,a21,a22,a23,a31,a32,a33,b11,b12,b
13,b21,b22,b23,b31,b32,b33,c12,c11,c13,c21,c22,c23,c31,c32,
c33);

initial
begin
clk=1'b1;
 forever
 #5 clk=~clk;
end
```

```
initial
begin
reset =1'b0;
# 5 reset =1;
end
initial
begin
$monitor($time,"a11=%b,a12=%b,a13=%b,a21=%b,a22=%b,a23=%b,a31=%b,a32=%b,a33=%b,b11=%b,b12=%b,
b13=%b,b21=%b,b22=%b,b23=%b,b31=%b,b32=%b,b33=%b,c11=%b,c12=%b,c13=%b,c21=%b,c22=%b,c23=%b,
c31=%b,c32=%b,c33=%b",a11,a12,a13,a21,a22,a23,a31,a32,a33,b11,b12,b13,b21,b22,b23,b31,b32,b33,c11,c12,
c13,c21,c22,c23,c31,c32,c33);

#10
a11=4'b1010;a12=4'b1110;a13=4'b1011;a21=4'b0010;a22=4'b1010;a23=4'b0011;a31=4'b1011;a32=4'b1111;a33=4'b0000;
b11=4'b1110;b12=4'b1011;b13=4'b1001;b21=4'b0110;b22=4'b1110;b23=4'b1100;b31=4'b1000;b32=4'b0010;
b33=4'b1111;
end

initial
begin
#100
$finish;
end
endmodule
```

NOTE-THESE MONITOR VALUES WILL BE SEEN IN TCL CONSOLE IN XILINX VERILOF FILE.

EXPLANATION: [FOR PROJECT TILL MID-EVAL]

- We have used gate level description to generate a Matrix Multiplier of sizes 3X3 AND 3X3 with each element of size 4-BIT . Hence total 18 inputs are required and as per specification the outputs should be in a serialised fashion, providing one output at a time.
- Here, we have used 9 Clock cycles for 9 total outputs.
- We have used Full adder , 4 bit adder , 8 bit adder and 4 bit multiplier modules .
- Full adder- By instantiating predefined logic gates.
- 4 bit adder - By instantiating modules of full adders.
- 8 bit adder - By instantiating modules of 4 bit adders .
- 4 bit multiplier - By instantiating Full adders and 8 bit adders , as multiplication of two 4 bit numbers will result in a 8 bit number so to add them according to multiplication of matrices we required 8 bit adders .
- The final size of each element in output matrix will be of 10 bits as addition of three 8 bit numbers resulted from three individual multiplications result in a max of 10 bit number .
- Just for sake of interest we have added testbenches for 4 bit adder and 4 bit multiplier in above slides .
- In Matrix Multiplicator Module- we required three 4 bit multipliers, two 8 bit adders and a single full adder because 4 bit multiplier to multiply elements and then using 8 bit adders to add them simultaneously according to principle of matrix multiplication , so the result will be of 10 bits , so to solve for 9th and 10th bit we have used one full adder by giving 10th bit the name as Cout and 9th bit as Sum of full adder .i.e we have individually solved for these two bits for the result to match the actual value of 10 bit binary number .

- After this , on positive clock edge or negative reset edge "always" block will be executed and hence we can specify each output at each positive clock edge . And reset input which is used is just for implementing first IF statement so that cycle_count may get a zero value at first , that is why in testbench you see after certain time delay the value of reset is kept high all time to execute ELSE statement and to proceed further increments of cycle_count until finish.
- So, this always block will be triggered at each positive clock edge again and again and skipping the IF statement and executing ELSE statement each time until cycle_count reaches its final value cycle_count<9 starting from 0 (hence in total 9 clock cycles are used) .
- The time delays in test bench are also set accordingly to execute code smoothly .
- The first positive edge is there on t=0 where reset is low hence implementing IF statement therefore giving cycle_count value of zero. After that at random t=5 delay reset is set high all time so to not execute IF statement anymore.
- And at t=10; when second positive edge triggers the inputs of matrices are stored are given as inputs till finish and hence the values come accordingly at a delay of 10 each time one by one and hence at t=10 first element outputs appears .
- Similarly at t=20 : second element output appears and this goes till when at t=90 , 9th element output appears.
- Finally, at t=100 , the clock cycles and other inputs are terminated .
- Hence , execution of code is done successfully.

FINAL SUBMISSION

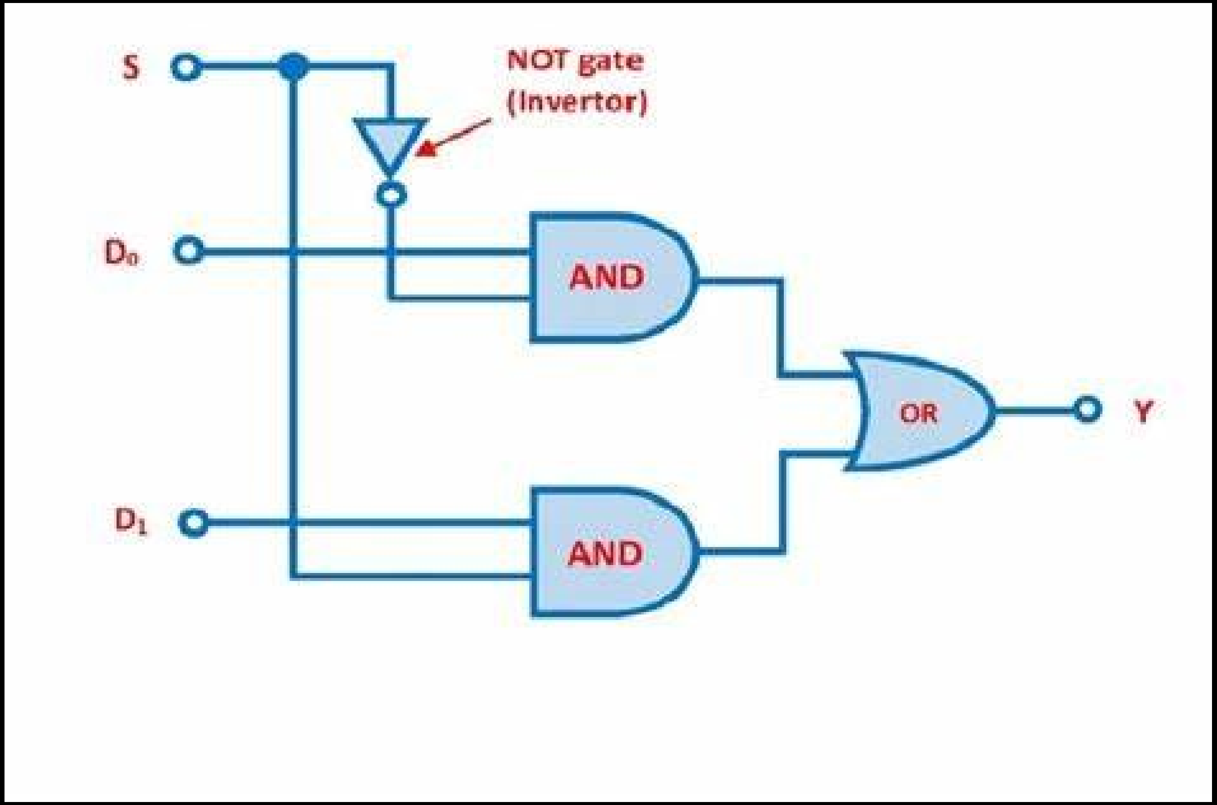
PRINCIPLE:

- ➢ This project implements Matrix Multiplier of two different or same sized matrices of form **mxn and nxp** such that m,n,p belongs to integer from 1 to 3 .That is there are 27 different possibilities of matrix multiplication where minimum size can be 1x1 and 1x1 and maximum can be 3x3 and 3x3 . Using VERILOG gate-level description with maximum of 18 elements in total and each element of **size 4 BITS**.
- ➢ We have used 18 clock cycles for inputs , one clock cycle for each input element getting stored into a Register .
- ➢ And, after that we have set next 9 clock cycles demonstrating one output at a time at each positive clock edge.
- ➢ It includes modules of following for generating matrix multiplier :

- Basic gates (AND , OR , XOR, NOT).(Predefined)
- Full adder.
- 4-bit Adder.
- 8-bit Adder.
- 4-bit Multiplier.
- Multiplexer 2 to 1.
- Multiplexer 2 to 1 [4-BIT].
- Multiplexer 9 to 1 [Each Data Line Of Size 4-BITS].
- PIPO (Parallel IN Parallel OUT) Register.
- Using concepts of FSM [FINITE STATE MACHINE]
- Matrix Multiplier producing one output at a time over 9 clock cycles.
- Use of reset and clock inputs will be required.
- Testbench (for simulation).

Mux 2:1

```
module mux2to1(a,b,sel,y);
 input a,b,sel;
 output y;
   wire not_sel, and_a, and_b;

   not (not_sel, sel);
   and (and_a, a, not_sel);
   and (and_b, b, sel);
   or (y, and_a, and_b);
endmodule
```



Mux 2:1 (FOR 4-BITS OF INPUTS)

```
module mux2to1_4bit(a,b,sel,y);
input [3:0] a,b;
input sel;
output [3:0] y;

   mux2to1 mux0(a[0], b[0], sel, y[0]);
   mux2to1 mux1(a[1], b[1], sel, y[1]);
   mux2to1 mux2(a[2], b[2], sel, y[2]);
   mux2to1 mux3(a[3], b[3], sel, y[3]);

endmodule
```

MUX 9:1 (FOR A MATRIX INPUTS)

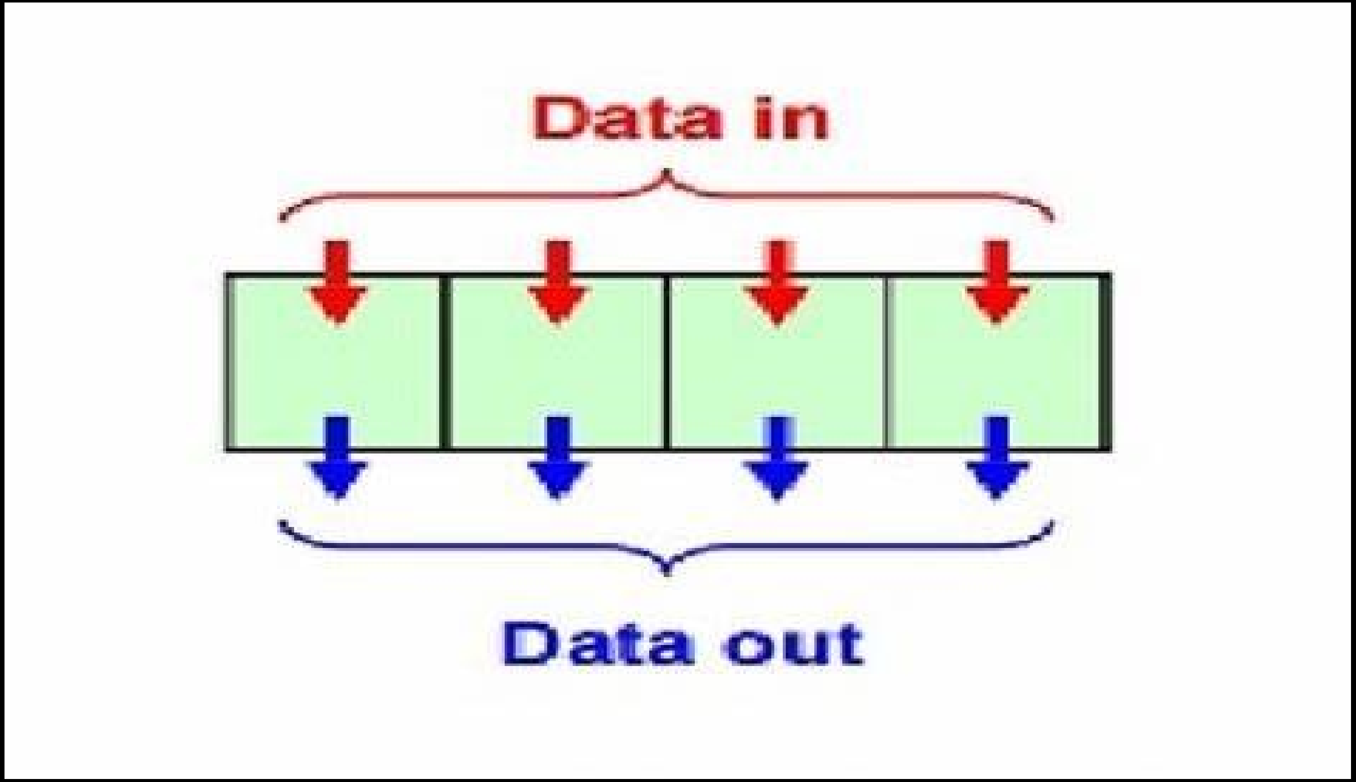
```
module mux9to1_a(input [4:0] sel, input [35:0] d, output [3:0] y);
   wire [3:0] mux0_out, mux1_out, mux2_out, mux3_out;
   wire [3:0] mux4_out, mux5_out, mux6_out, mux7_out;
   wire [3:0] mux8_out, mux9_out, mux10_out, mux11_out;

   mux2to1_4bit mux0_4bit(d[3:0], d[7:4], sel[0], mux0_out);
   mux2to1_4bit mux1_4bit(d[11:8], d[15:12], sel[0], mux1_out);
   mux2to1_4bit mux2_4bit(d[19:16], d[23:20], sel[0], mux2_out);
   mux2to1_4bit mux3_4bit(d[27:24], d[31:28], sel[0], mux3_out);
   mux2to1_4bit mux4_4bit(mux0_out, mux1_out, sel[1], mux4_out);
   mux2to1_4bit mux5_4bit(mux2_out, mux3_out, sel[1], mux5_out);
   mux2to1_4bit mux6_4bit(mux4_out, mux5_out, sel[2], mux6_out);
   mux2to1_4bit mux7_4bit( mux6_out, d[35:32],sel[3], mux7_out);
   mux2to1_4bit mux8_4bit(mux7_out, 4'bxxxx, sel[4], y);
endmodule
```

MUX 9:1 (FOR B MATRIX INPUTS)

```
module mux9to1_b(input [4:0] sel, input [35:0] d, output [3:0]y);
   wire [3:0] mux0_out, mux1_out, mux2_out, mux3_out;
   wire [3:0] mux4_out, mux5_out, mux6_out, mux7_out;
   wire [3:0] mux8_out, mux9_out, mux10_out, mux11_out;

   mux2to1_4bit mux0_4bit(d[3:0], d[7:4], sel[0], mux0_out);
   mux2to1_4bit mux1_4bit(d[11:8], d[15:12], sel[0], mux1_out);
   mux2to1_4bit mux2_4bit(d[19:16], d[23:20], sel[0], mux2_out);
   mux2to1_4bit mux3_4bit(d[27:24], d[31:28], sel[0], mux3_out);
   mux2to1_4bit mux4_4bit(mux0_out, mux1_out, sel[1], mux4_out);
   mux2to1_4bit mux5_4bit(mux2_out, mux3_out, sel[1], mux5_out);
   mux2to1_4bit mux6_4bit(mux4_out, mux5_out, sel[2], mux6_out);
   mux2to1_4bit mux7_4bit( mux6_out, d[35:32],sel[3], mux7_out);
   mux2to1_4bit mux8_4bit(mux7_out, 4'bxxxx, sel[4], y);
endmodule
```

PIPO REGISTER:

```
module PIPO_REGISTER(clk,inp,outp);
input clk;
input [3:0] inp;
output reg[3:0] outp;

always @(*)
  outp <=inp;

 endmodule
```



MATRIX MULTIPLIER: [size -(MxN NxP)] CODE

```
module MatrixMultiplier (done,clk,rst,m,n,p,A00, A01, A02, A10, A11, A12,A20, A21, A22, B00, B01, B02,B10, B11,
B12,B20, B21, B22, C00, C01, C02, C10, C11, C12, C20, C21, C22);
    input clk,rst;
    input [1:0]m,n,p;
    input [3:0] A00, A01, A02, A10, A11, A12,A20, A21, A22;
    input [3:0] B00, B01, B02,B10, B11, B12,B20, B21, B22;
    output reg [9:0] C00, C01, C02, C10, C11, C12, C20, C21, C22;
    output reg done;

    reg [2:0] state;
    integer  i, j,k;
    reg [3:0] A[0:2][0:2];
    reg [3:0] B[0:2][0:2];
    reg [9:0] C[0:2][0:2];
    wire [7:0] mul_result;
    wire [7:0] add_result;
wire add_9th;
reg partial_add_9th;
reg [7:0] partial_sum;
reg  [4:0] clock_count_a;
reg  [4:0] clock_count_b;
wire Sum_9th,Cout_10th;
reg Cout_val_10th;

wire [35:0] a_inputs  ={A22,A21,A20,A12,A11,A10,A02,A01,A00};
wire [35:0] b_inputs ={B22, B21, B20, B12, B11, B10,B02,B01,B00 };

parameter S0 = 3'b000, S1 = 3'b001, S2 = 3'b010, S3 = 3'b011,S4=3'b100;
wire[3:0] mux_a_output;
wire[3:0] mux_b_output;
wire[3:0] mux_a_out;
wire[3:0] mux_b_out;

multiplier_4bit mult(A[i][k],B[k][j],mul_result);
adder_8bit add(add_result , add_9th ,partial_sum, mul_result ,0);
Full_Adder bit_9_10(Sum_9th,Cout_10th,partial_add_9th,add_9th,0);
```

```
mux9to1_a a_input(.sel(clock_count_a[4:0]),.d(a_inputs),.y(mux_a_out));
mux9to1_b b_input(.sel(clock_count_b[4:0]),.d(b_inputs),.y(mux_b_out));

PIPO_REGISTER store_val_a(clk,mux_a_out,mux_a_output);
PIPO_REGISTER store_val_b(clk,mux_b_out,mux_b_output);


always @(posedge clk or negedge rst or k)

begin
    if (!rst)
        begin
          clock_count_a <= 0;
          state <= S0;


          done<=0;
           i<=0;
           j<=0;
        end
```

```
else
   begin
      if (state==S0)
           begin
                if (clock_count_a < 9)
                    begin
                           A[i][j]<=mux_a_output;
                           clock_count_a<= clock_count_a + 1;

                           if (j<2)
                               j<=j+1;

                           else if(i<2)
                             begin
                              i<=i+1;
                              j<=0;
                             end
                     end

                else
                   begin
                       state <= S1;
                       i<= 0;
                       j <= 0;
                       clock_count_b <= 0;
                   end
             end
```

```
else if (state==S1)
        begin
              if (clock_count_b  < 9)
                 begin
                    B[i][j]<=mux_b_output;
                    clock_count_b<= clock_count_b + 1;

                    if (j<2)
                      j<=j+1;

                    else if(i<2)
                       begin
                        i<=i+1;
                        j<=0;
                        //k<=0;
                       end
                  end
               else
                 begin
                 state = S2;
                 i <= 0;
                 j <= 0;
                 k <= 1;
                 end
         end
```

```
else if (state==S2)
begin
        partial_sum <= 0;
        partial_add_9th<=0;
        Cout_val_10th<=0;
        state <= S3;
        k <= 0;
     end

  else if(state==S3)
   begin
         if (k < n)
           begin
             partial_sum <= add_result;
             partial_add_9th<=Sum_9th;
             Cout_val_10th<=Cout_10th;
             k <= k + 1;
           end
         else
           begin
              C[i][j] <={Cout_val_10th,partial_add_9th, partial_sum};
              state <= S4;
          end
      end
```

```
else if(state== S4)
                begin
                       if (j < p-1)
                          begin
                            j <= j + 1;
                            state <= S2;
                            k<=1;
                          end
                       else if (i < m-1)
                          begin
                             j <= 0;
                             i <= i + 1;
                            state <= S2;
                              k<=1;
                         end

                       else
                         begin
                           done <= 1;
                        end
                    end
        end
    end
    always @(posedge clk) begin
     //ouputs getting stored into 10 bit PIPO registers
        C00 <= C[0][0];
        C01 <= C[0][1];
        C02 <= C[0][2];
        C10 <= C[1][0];
        C11 <= C[1][1];
        C12 <= C[1][2];
        C20 <= C[2][0];
        C21 <= C[2][1];
        C22 <= C[2][2];
     end
  endmodule
```

TESTBENCH-1 : [size 3x3 and 3x3]

```
module testing;

   reg [1:0]m,n,p;
   reg clk;
   reg rst;
   reg [3:0] A00, A01, A02;
   reg [3:0] A10, A11, A12;
   reg [3:0] A20, A21, A22;
   reg [3:0] B00, B01, B02;
   reg [3:0] B10, B11, B12;
   reg [3:0] B20, B21, B22;
   wire [9:0] C00, C01, C02;
   wire [9:0] C10, C11, C12;
   wire [9:0] C20, C21, C22;
   wire done;
 MatrixMultiplier uut (done,clk,rst,m,n,p,A00, A01, A02, A10, A11, A12,A20, A21, A22, B00, B01, B02,B10, B11, B12,B20, B21,
B22, C00, C01, C02, C10, C11, C12, C20, C21, C22);
```

```
initial
      begin
      clk=1'b0; rst=1'b0;
      m=2'b11;n=2'b11;p=2'b11;  // hence m=3,n=3,p=3 for 3x3 and 3x3
      end
initial begin
      #10  rst=1'b1;
       end
always #5 clk = ~clk;
initial   begin
                #5  A00<=4'b0001;  #10 A01<= 4'b0010; #10 A02<=4'b0011;
                #10 A10<=4'b0100; #10 A11<=4'b0101;  #10 A12<=4'b0110;
                #10 A20<=4'b0111; #10 A21<=4'b1000; #10 A22<=4'b1001;
                #10 B00<=4'b0001; #10 B01<=4'b0010; #10 B02<=4'b0011;
                #10 B10<=4'b0100;  #10 B11<=4'b0101;  #10 B12<=4'b0110;
                #10 B20<=4'b0111; #10 B21<=4'b1000;   #10 B22<=4'b1001;
                   end
initial  begin
       #215    $monitor("TIME :",$time,": C00=%d, C01=%d, C02=%d,C10=%d, C11=%d, C12=%d,C20=%d, C21=%d, C22=%d ",C00, C01, C02, C10,
C11, C12, C20, C21, C22);
       end

initial begin
      #300   $finish;
       end
endmodule
```

TESTBENCH-2: [size 2x3 and 3x3]

```
module testing;

   reg [1:0]m,n,p;
   reg clk;
   reg rst;
   reg [3:0] A00, A01, A02;
   reg [3:0] A10, A11, A12;
   reg [3:0] A20, A21, A22;
   reg [3:0] B00, B01, B02;
   reg [3:0] B10, B11, B12;
   reg [3:0] B20, B21, B22;
   wire [9:0] C00, C01, C02;
   wire [9:0] C10, C11, C12;
   wire [9:0] C20, C21, C22;
   wire done;
 MatrixMultiplier uut (done,clk,rst,m,n,p,A00, A01, A02, A10, A11, A12,A20, A21, A22, B00, B01, B02,B10, B11, B12,B20, B21,
B22, C00, C01, C02, C10, C11, C12, C20, C21, C22);
```

```
initial
     begin
     clk=1'b0; rst=1'b0;
     m=2’b10;n=2'b11;p=2'b11; // hence m=2,n=3,p=3 for 2x3 and 3x3
     end
initial begin
     #10  rst=1'b1;
      end
always #5 clk = ~clk;
initial   begin
             #5  A00<=4'b0001;  #10 A01<= 4'b0010; #10 A02<=4'b0011;
             #10 A10<=4'b0100; #10 A11<=4'b0101;  #10 A12<=4'b0110;
            #10 B00<=4'b0001; #10 B01<=4'b0010; #10 B02<=4'b0011;
             #10 B10<=4'b0100;  #10 B11<=4'b0101;  #10 B12<=4'b0110;
             #10 B20<=4'b0111; #10 B21<=4'b1000;   #10 B22<=4'b1001;
               end
initial  begin
      #215    $monitor("TIME :",$time,": C00=%d, C01=%d, C02=%d,C10=%d, C11=%d, C12=%d,C20=%d, C21=%d, C22=%d ",C00, C01, C02, C10,
C11, C12, C20, C21, C22);
      end

initial begin
     #300   $finish;
      end
endmodule
```

EXPLANATION:

- We have used gate level description to generate a Matrix Multiplier of different sizes matrices of form mXn and nXp with each element of size 4-BIT . Hence total of max 18 inputs are required and as per specification the inputs should be stored in a serialised fashion at each clock edge hence using 18 clock cycles and also providing one output at a time so using another 9 clock cycles .
- Here, we have used 9 Clock cycles for 9 total outputs and 18 clock cycles for 18 inputs .
- If there are not 18 inputs ,that is if order of matrix is not 3x3 then also the total number of cycles used in inputs and outputs remain 18 and 9 respectively ,according to the code written in our matrix multiplier .
- We have used Full adder , 4 bit adder , 8 bit adder and 4 bit multiplier , 2 to 1 mux, 4 bit 2to1 mux , 9to1 mux having each input of size 4 bit, PIPO Register modules .
- Full adder- By instantiating predefined logic gates.
- 4 bit adder - By instantiating modules of full adders.
- 8 bit adder - By instantiating modules of 4 bit adders .
- 4 bit multiplier - By instantiating Full adders and 8 bit adders , as multiplication of two 4 bit numbers will result in a 8 bit number so to add them according to multiplication of matrices we required 8 bit adders .
- 2to1 MUX- By instantiating predefined logic gates.
- 4 bit 2to1 mux-By instantiating 2to1 MUXes.
- 9to1 mux- mux-By instantiating 4 bit 2to1 mux.

- The final size of each element in output matrix will be of 10 bits as addition of three 8 bit numbers resulted from three individual multiplications result in a max of 10 bit number .
- We have used two multiplexers 9to1 of each input size of 4 bits as to assign values to A[i][j] and B[i][j] at each clock_count_a and clock_count_b when they are incrementing by one at each posedge of clock and both these clock_count of a and b act as a select line for 9 to 1 mux so according to this select line value we are able to store A and B matrix inputs in a register at each clock edge and then assigning this mux_output_a to A[i][j] and mux_output_b to B[i][j] as per the values of i and j incrementing also by 1 at each clock edge . So that means this is like storing and assigning inputs side by side at each clock edge , hence, that is why we have used 4 bit PIPO register which is executing its always block at change of each input or clock .
- Now, starting with first positive clock edge , the if(!rest) block will be executed hence making clock_count_a =0 and assigning state to S0 with i and j equal to zero and with this the first element from matrix A is stored in register and when the next clock edge strikes this stored value is getting assigned to A[0][0] and this way the process continues till clock_count_a reaches value 8 and hence all 9 inputs of A are stored first and then assigned to A[i][j] simultaneously and now the state is shifted to S1 with now assigning value clock_count_b equal to zero , and after tranition to S2 state at nextclock edge the whole process of assigning inputs one by one in a serialized fashion is similarly done with matrix elements of matrix B . So in this way 18 clock cycles are used for storing 18 elements.
- Note- First clock cycle is getting used for executing if(!rst) block for setting up the values of state , i , j and clock_count_a. So we can say at next 18 clock cycles the inputs are assigned one by one at each positive edge.

- That means states S0 and S1 has significance of storing and assigning one by one inputs according to project tasks specified.
- Now , after this work of inputs , at next clock edge ,state is shifted to S2 and from here the task of matrix multiplication begins , where A[i][k] and B[k][j] elements are placed into instantiated multiplier module .So, here in this state we define i and j to zero and k is also assigned value zero as we can see always block is getting triggered by change in value of k , so to avoid excess clock cycles usage we have also made always block triggered by k value hence as we define k=0 in state S2 it helps to change state to S3 state with i, j, k equal to zero hence avoiding excess clock cycles .
- For S3 state now with i and j equal to zero means A[0][k] and b[k][0] signifying that for multiplying and adding elements of row 0 of matrix A and elements of column 0 of matrix B as per matrix multiplication principle with value of k getting incremented each time according to statements we can see in code , and also as value of k changes always block is getting executed again and again and each time saving extra clock cycles and also the state remains S3 only until all the calculation for the first element C[0][0] is completed. The values of A[0][k] and B[k][0] is passing through the multiplier module at each k=1,2,3 and then is getting added with the help of 8 bit adder .
- Firstly the multiplication of 2 elements occur which is then stored to add_result and this add_result is assigned to partial_sum and now k is incremented by 1 so that next two elements multiplication occur which is added to previously assigned partial_sum and the sum of these two 8 bits number is equal to add_result and now again this value is assigned to partial_sum and k is again incremented by 1 and the multiplication of next 2 elements is again added to previously defined partial_sum finally resulting add_result (which is now sum of three multiplication results) and this add_result is finally assigned to partial_sum and hence we get final output of our first element C[0][0] which is assigned value of partial_sum and at the end state is updated to S4.

- Now, this is a point where always block is getting triggered by next positive clock edge , when once we have got a value of particular C[i][ij element .At this point, state is shifted from S3 to S4 with help of this clock edge and not by changing value of k.
- Also for 9 th and 10 th bit , as we know addition of three 8 bits number may result in a maximum of 10 bit number , so for that we have solved it separately for 9 th and 10 th bit by instantiating one Full adder whose Sum is assigned to 9th bit and whose Cout is assigned to 10 th bit of a 10 bit C[i][j] value ,so hence we have used concatenation operator for the purpose so that output elements value could match the actual result which are observed by normal matrix multiplication.
- S4 is just meant to increase and change value of i and j in order to proceed multiplication and addition for different rows and columns of A and B and coming up with ouput elements of ouput matrix C, when i or j is incremented according to statements there in code , the state is updated to S2 as by changing value of k and the same process continues for that particular i row of matrix A and j column of matrix B with value k initialized to zero value again. In state S2.
- This process continues and we get each element of matrix C in form of C[i][j] for particular i and j , and these values of C[i][j] are assigned to C0,C01,C02,C10,C11,..etc at each posedge of clock and this way we are able to extract each output element of C at each clock edge .As one run after clock edge ends until we get one value of output element of matrix C , as that is why we have assigned different values of k in each state so that always block may get triggered and may not waste our clock edges for getting executed .

TEAM WORK:

- **Naman Goyal (230002046) :**
- Executing the main module -Matrix Multiplier Module , Module of Multiplier ,PIPO Register and 9 bit MUX .
- Designed the main module for final submission and contributed for the same in mid eval submission.
- Executed TESTBENCHES.
- Handling of explanation part in ppt.
- Attached files to Github through my account.

- **Sarvadnyee Ghogare (230002065):**
- Executed all other Modules required in implementation of Matrix Multiplier such as of adders , mux etc.
- Contributed in designing main module of matrix multiplier of size 3x3 which was for mid eval submission .
- Executed TESTBENCHES of those modules which are required in main module ,to check whether there code implementation is correct or not.
- Presentation and handling of Project in a mannered way in form of ppt.

BIBLIOGRAPHY

For completion of this project . We have used following sources:

- https://youtu.be/NCrlyaXMAn8?si=EW0wYZ2JKMJ0FE4V
- https://youtu.be/33PAoJGm2Fo?si=59G74QvJg13ZA2mF
- https://www.auhd.edu.ye/upfiles/elibrary/Azal2020-01-22-12-16-48-75016.pdf (MANO DIGITAL DESIGN BOOK)
- CHAT GPT
- https://support.xilinx.com/s/question/0D52E00006hpkV6SAI/on-the-fly-syntax-checking-and-code-folding?language=en_US
- XILINX VERILOG SOFTWARE.