```
//================Constants.v=================

`define BIT_WIDTH 16
`define ROWS 4
`define COLS 4
```

```verilog
`timescale 1ns / 1ps


//This systolic array uses output stationary method to calculate matrices
// while inputs and weights are fed in a "wavefront" manner, outputs remain in the systolic array
// once calculations finish, one needs to peak into each PE element to pull data
// The other common types are input stationary and weight stationary. sometimes more efficient, but this was easier to implement

//==================TOP MODULE===================
`include "Constants.v"
module Top_Module(
input reset, clk, start,
output reg [15:0] a0x,a1x,a2x,a3x, b0x,b1x,b2x,b3x,
output reg [`BIT_WIDTH*2-1:0] c_data0,c_data1,c_data2,c_data3,
output reg execution_finished,
output array_ready_output
);

//index wires TO ROM
reg [`BIT_WIDTH-1:0] a_index0, a_index1, a_index2,a_index3;
reg [`BIT_WIDTH-1:0] b_index0, b_index1, b_index2,b_index3;
reg [`BIT_WIDTH-1:0] c_index;
//output wires from ROM
wire [`BIT_WIDTH-1:0] a_data0,a_data1,a_data2,a_data3;
wire [`BIT_WIDTH-1:0] b_data0,b_data1,b_data2,b_data3;

reg [`BIT_WIDTH*2-1:0] c_data0,c_data1,c_data2,c_data3;
reg [`BIT_WIDTH*2-1:0] c_temp0,c_temp1,c_temp2,c_temp3;

//output from systollic array
wire [`BIT_WIDTH * 2 -1:0] c_out [0:`ROWS * `COLS - 1];
//controlls to systollic array
reg [2:0] YZctrl;

// ROM instantiation
ROM_A rom_a_inst(
.clk(clk),
.index0(a_index0), .index1(a_index1), .index2(a_index2), .index3(a_index3),
.data0(a_data0), .data1(a_data1), .data2(a_data2), .data3(a_data3)
);

ROM_B rom_B_inst(
.clk(clk),
.index0(b_index0),.index1(b_index1),.index2(b_index2),.index3(b_index3),
.data0(b_data0),.data1(b_data1),.data2(b_data2),.data3(b_data3)
);

RAM_C RAM_C_inst(
.clk(clk),
.index(c_index),
.data0(c_data0),.data1(c_data1),.data2(c_data2),.data3(c_data3)
);

SimpleSystolicArray systolic_inst(
.clk(clk), .YZctrl(YZctrl),
.a0x(a0x), .a1x(a1x), .a2x(a2x), .a3x(a3x),
.bx0(b0x), .bx1(b1x), .bx2(b2x), .bx3(b3x),
.rdyGive(array_ready_output),
.c00(c_out[0]), .c01(c_out[1]), .c02(c_out[2]), .c03(c_out[3]),
.c10(c_out[4]), .c11(c_out[5]), .c12(c_out[6]), .c13(c_out[7]),
.c20(c_out[8]), .c21(c_out[9]), .c22(c_out[10]), .c23(c_out[11]),
.c30(c_out[12]), .c31(c_out[13]), .c32(c_out[14]), .c33(c_out[15])
);

//state machine declaration
//there are missing states so that we may build upon as needed
parameter IDLE = 3'b000, FEED =3'b001, COMPUTE = 3'b010, PULL = 3'b011, DONE = 3'b100;
reg [2:0] STATE, NEXT_STATE;

//counter instantiation These are to keep track of ROM feeding cycles
reg signed [`ROWS + `COLS - 1 :0] cycle_counter; //Tells logic what you are feeding now
reg signed [`ROWS + `COLS - 1 :0] next_cycle_counter; //Tells ROM what you are fetching next

always @(posedge clk or posedge reset) begin
    if(reset) STATE <= IDLE;
    else STATE <= NEXT_STATE;
end

always @(*) begin
    NEXT_STATE <= STATE;
    case (STATE)
        IDLE:if(start) NEXT_STATE <= FEED;
        FEED: NEXT_STATE <= (cycle_counter <= `ROWS+`COLS - 1) ? FEED : COMPUTE;
        COMPUTE:begin
                if (array_ready_output) begin
                next_cycle_counter <= 0;
                cycle_counter <= -1;
                NEXT_STATE <= PULL;
                YZctrl <= 2'b10;
                end else NEXT_STATE <= COMPUTE;
            end
        PULL: NEXT_STATE <= (cycle_counter < `ROWS-1)?  PULL : DONE;
        DONE: begin end  //DO nothing you are done :)
        default: NEXT_STATE <= STATE;
    endcase
end

always @(posedge clk or posedge reset) begin
    if(reset) begin
        next_cycle_counter <=0;
        cycle_counter <= -1; //this is so that this counter lags behind next_cycle_counter
        YZctrl <= 2'b00; //this resets array
        //I wanted to reset the a0x... registers here, but its a little easier to debug when it shows X
        //signifying that we are in idle state and have yet started to feed from ROM
    end
    else begin
        case(STATE)
            IDLE: begin
                // prefetch first set of data while waiting
                a_index0 <= next_cycle_counter;
                a_index1 <= next_cycle_counter-1;
                a_index2 <= next_cycle_counter-2;
                a_index3 <= next_cycle_counter-3;

                b_index0 <= next_cycle_counter;
                b_index1 <= next_cycle_counter-1;
                b_index2 <= next_cycle_counter-2;
                b_index3 <= next_cycle_counter-3;
            end
            FEED: begin
                //==========GENERATE ROM ADDRESS===============
                //Rom expects a Row/column index and thats what this is
                //the ROM_A will map A[0][aindex0] ;  A[1][aindex1]; M[2][aindex2]; M[3][aindex3]
                //This value is read in this clock cycle into a_data0 to a_data3
                //like wise for b

                //this is responsible for calculating the adresses so that wavefront can be formed
                a_index0 <= next_cycle_counter;
                a_index1 <= next_cycle_counter-1;
                a_index2 <= next_cycle_counter-2;
                a_index3 <= next_cycle_counter-3;

                b_index0 <= next_cycle_counter;
                b_index1 <= next_cycle_counter-1;
                b_index2 <= next_cycle_counter-2;
                b_index3 <= next_cycle_counter-3;

                //============FEED TO ARRAY LOGIC==================
                //This block takes the value from ROM output and puts it into the aXx registers
                //This is also responsible for adding zeroes into the cascading for delaying feed
                //These registers are currently set as outputs for debuggin, but can be connected to systollic array.
                if(cycle_counter >=0) begin
                    YZctrl <= 2'b01;
                    a0x <= (cycle_counter - 0 >= 0 && cycle_counter <= `COLS - 1) ? a_data0 : 16'b0;
                    a1x <= (cycle_counter - 1 >= 0 && cycle_counter - 1 <= `COLS-1) ? a_data1 : 16'b0;
                    a2x <= (cycle_counter - 2 >= 0 && cycle_counter - 2 <= `COLS-1) ? a_data2 : 16'b0;
                    a3x <= (cycle_counter - 3 >= 0 && cycle_counter - 3 <= `COLS-1) ? a_data3 : 16'b0;

                    b0x <= (cycle_counter - 0 >= 0 && cycle_counter - 0 <= `ROWS - 1) ? b_data0 : 16'b0;
                    b1x <= (cycle_counter - 1 >= 0 && cycle_counter - 1 <= `ROWS-1) ? b_data1 : 16'b0;
                    b2x <= (cycle_counter - 2 >= 0 && cycle_counter - 2 <= `ROWS-1) ? b_data2 : 16'b0;
                    b3x <= (cycle_counter - 3 >= 0 && cycle_counter - 3 <= `ROWS-1) ? b_data3 : 16'b0;
                end
                //============UPDATE COUNTER============
                //These counters keeps the feeding pipelined and in check
                //Next_clock_cycle leads cycle_count by 1
                //this is so that in each cycle, next_Cycle is queeing up the next adresses and ROM values
                //while cycle_counter is writing the current ROM values to the a0x... registers
                next_cycle_counter <= next_cycle_counter + 1;
                cycle_counter <= cycle_counter + 1;

                //This repeats this state until 2n-1 rows have been read (matrix has been read and inserted into array)
                //need to calculate how many clock cycles until array itself has completed its calculation
                //i was intendeding to feed this into the Compute state to signify that reading is done, but computation is still ongoing
            end
            COMPUTE: begin
                //This is where feeding from ROM has completed, the left and top elements have recieved all data
                //THis should just be sending 0 to array and wait for last element to calculate
                //Last element calculates at
                //I calculated that this is how many cycles from it takes to complete the co67mputation, but i dont know tho

            end
            PULL: begin
                c_temp0 <= c_out[next_cycle_counter * `ROWS];
                c_temp1 <= c_out[next_cycle_counter * `ROWS + 1];
                c_temp2 <= c_out[next_cycle_counter * `ROWS + 2];
                c_temp3 <= c_out[next_cycle_counter * `ROWS + 3];

                if(cycle_counter >= 0) begin
                    c_data0 <= c_temp0;
                    c_data1 <= c_temp1;
                    c_data2 <= c_temp2;
                    c_data3 <= c_temp3;
                    c_index <= cycle_counter;
                end
                next_cycle_counter <= next_cycle_counter + 1;
                cycle_counter <= cycle_counter + 1;
            end
            DONE: begin
                execution_finished <= 1;
            //RElax you are finished
            end
        endcase
    end
end
endmodule
```

```
//========================SYSTOLIC ARRAY=============================
module mathStandin(
    input [15:0] A, B,
    input [31:0] C,
    output [31:0] Cp
    );

    assign Cp = C + A*B;

endmodule

module addMultMatrix (
    input [15:0]
        a00, a01, a02, a03,
        a10, a11, a12, a13,
        a20, a21, a22, a23,
        a30, a31, a32, a33,

    input [15:0]
        b00, b01, b02, b03,
        b10, b11, b12, b13,
        b20, b21, b22, b23,
        b30, b31, b32, b33,

    input [31:0]
        c00, c01, c02, c03,
        c10, c11, c12, c13,
        c20, c21, c22, c23,
        c30, c31, c32, c33,

    output [31:0]
        cp00, cp01, cp02, cp03,
        cp10, cp11, cp12, cp13,
        cp20, cp21, cp22, cp23,
        cp30, cp31, cp32, cp33
    );

    mathStandin m00 (a00, b00, c00, cp00);
    mathStandin m01 (a01, b01, c01, cp01);
    mathStandin m02 (a02, b02, c02, cp02);
    mathStandin m03 (a03, b03, c03, cp03);
    mathStandin m10 (a10, b10, c10, cp10);
    mathStandin m11 (a11, b11, c11, cp11);
    mathStandin m12 (a12, b12, c12, cp12);
    mathStandin m13 (a13, b13, c13, cp13);
    mathStandin m20 (a20, b20, c20, cp20);
    mathStandin m21 (a21, b21, c21, cp21);
    mathStandin m22 (a22, b22, c22, cp22);
    mathStandin m23 (a23, b23, c23, cp23);
    mathStandin m30 (a30, b30, c30, cp30);
    mathStandin m31 (a31, b31, c31, cp31);
    mathStandin m32 (a32, b32, c32, cp32);
    mathStandin m33 (a33, b33, c33, cp33);
endmodule

module SimpleSystolicArray(
    input clk,
    input [2:0] YZctrl,
    input [15:0]
        a0x, a1x, a2x, a3x,
    input [15:0]
        bx0, bx1, bx2, bx3,
    output reg rdyGive, // this is to tell the controls that the C matrix is ready to take.
    output reg [31:0]
        c00, c01, c02, c03,
        c10, c11, c12, c13,
        c20, c21, c22, c23,
        c30, c31, c32, c33
    );

    parameter
        reset     = 2'b00, // must be run first when first ran,
        run       = 2'b01, //
        outputC   = 2'b10, // Controls want an output for C.
        pause     = 2'b11, // pauses the systolic array. all check and calculation values are kept in place.
        ding      = 4'd8, // NOT CORRECT VALUE for sure.
        running   = 2'b01,
        waiting   = 2'b10,
        clearC    = 2'b11;

        // ding means that the Systolic array is and will store C matrix until controller wants to take the matrix.

    reg [3:0] clk_count;
    reg [15:0]
        a01, a02, a03,
        a11, a12, a13,
        a21, a22, a23,
        a31, a32, a33,

        b10, b11, b12, b13,
        b20, b21, b22, b23,
        b30, b31, b32, b33;

    reg [31:0]
        ci00, ci01, ci02, ci03,
        ci10, ci11, ci12, ci13,
        ci20, ci21, ci22, ci23,
        ci30, ci31, ci32, ci33;

    wire [31:0]
        cp00, cp01, cp02, cp03,
        cp10, cp11, cp12, cp13,
        cp20, cp21, cp22, cp23,
        cp30, cp31, cp32, cp33;

    reg [1:0] sysArrState;
            // determines the state that the systolic array is in.
            // only applicable if run is current global state (ie YZctrl)
    reg matrixWasInitialized;

    addMultMatrix CTicPlusOne (
        a0x, a01, a02, a03,
        a1x, a11, a12, a13,
        a2x, a21, a22, a23,
        a3x, a31, a32, a33,

        bx0, bx1, bx2, bx3,
        b10, b11, b12, b13,
        b20, b21, b22, b23,
        b30, b31, b32, b33,

        ci00, ci01, ci02, ci03,
        ci10, ci11, ci12, ci13,
        ci20, ci21, ci22, ci23,
        ci30, ci31, ci32, ci33,

        cp00, cp01, cp02, cp03,
        cp10, cp11, cp12, cp13,
        cp20, cp21, cp22, cp23,
        cp30, cp31, cp32, cp33
    );

    always @ (posedge clk) begin

        case (YZctrl)
            // this should initialize all values to zero
            // reset should be run before running anything through the systolic array.
            // counters and anything that should have an initial value should be put here.
            // including C matrix.

            reset : begin
                rdyGive = 0;
               { a01, a02, a03,
                 a11, a12, a13,
                 a21, a22, a23,
                 a31, a32, a33 } <= 192'b0;

               { b10, b11, b12, b13,
                 b20, b21, b22, b23,
                 b30, b31, b32, b33 } <= 192'b0;

                { ci00, ci01, ci02, ci03,
                  ci10, ci11, ci12, ci13,
                  ci20, ci21, ci22, ci23,
                  ci30, ci31, ci32, ci33 } <= 512'b0;

                { c00, c01, c02, c03,
                  c10, c11, c12, c13,
                  c20, c21, c22, c23,
                  c30, c31, c32, c33 } <= 512'b0;

                 clk_count <= 4'b0;

                 // this is here because when reset is set back to 0, and the controls indicate
                 // to start computing C matrix, then I want it to immediately start running.
                 sysArrState <= running;
                 matrixWasInitialized = 1;

            end
            // this is where a lot of the work will go into.
            // this will be the step that takes inputs from A and B and computes a cycle of C matrix values
            // this module will also have a clock counter so we know when C matrix is ready.
            run : begin
                if (!matrixWasInitialized)
                    sysArrState = waiting;
                case (sysArrState)
                    clearC : begin
                       rdyGive = 0;
                       { a01, a02, a03,
                         a11, a12, a13,
                         a21, a22, a23,
                         a31, a32, a33 } <= 192'b0;

                       { b10, b11, b12, b13,
                         b20, b21, b22, b23,
                         b30, b31, b32, b33 } <= 192'b0;

                        { ci00, ci01, ci02, ci03,
                          ci10, ci11, ci12, ci13,
                          ci20, ci21, ci22, ci23,
                          ci30, ci31, ci32, ci33 } <= 512'b0;

                        { c00, c01, c02, c03,
                          c10, c11, c12, c13,
                          c20, c21, c22, c23,
                          c30, c31, c32, c33 } <= 512'b0;

                          clk_count <= 4'b0;
                    end
                    running : begin
                        rdyGive = 0;

                        if (clk_count == ding) begin
                            sysArrState = waiting;
                        end
                        else begin
                            // do the mathy stuff here
                            //as you wish ;)

                            // setting the registers of C values to C_plus_ab
                             {ci00, ci01, ci02, ci03,
                              ci10, ci11, ci12, ci13,
                              ci20, ci21, ci22, ci23,
                              ci30, ci31, ci32, ci33}
                             <=
                             {cp00, cp01, cp02, cp03,
                              cp10, cp11, cp12, cp13,
                              cp20, cp21, cp22, cp23,
                              cp30, cp31, cp32, cp33};

                           // delaying b([x+1]K) <-  b(xK)
                             {b10, b11, b12, b13}
                             <=
                             {bx0, bx1, bx2, bx3};

                             {b20, b21, b22, b23}
                             <=
                             {b10, b11, b12, b13};

                             {b30, b31, b32, b33}
                             <=
                             {b20, b21, b22, b23};

                           // delaying a([x+1]K) <-  a(xK)
                             {a01, a11, a21, a31}
                             <=
                             {a0x, a1x, a2x, a3x};

                             {a02, a12, a22, a32}
                             <=
                             {a01, a11, a21, a31};

                             {a03, a13, a23, a33}
                             <=
                             {a02, a12, a22, a32};

                            clk_count = clk_count + 1;
                        end
                    end
                    waiting : begin
                        // do nothing here. ensure that the C matrix is protected and do not take any a or b inputs.
                        // this is the only place that C matrix is
                        rdyGive = 1;
                    end
                    endcase
            end
            outputC : begin
               // if (rdyGive && matrixWasInitialized) begin
                    { c00, c01, c02, c03,
                      c10, c11, c12, c13,
                      c20, c21, c22, c23,
                      c30, c31, c32, c33 }
                     <=
                    { cp00, cp01, cp02, cp03,
                      cp10, cp11, cp12, cp13,
                      cp20, cp21, cp22, cp23,
                      cp30, cp31, cp32, cp33 };

                    // this is here so that when the YZctrls switch back to running after taking the C matrix,
                    // we the systolic array will ititize values.
                    sysArrState <= clearC;
              /*  end
                else begin
                    { c00, c01, c02, c03,
                      c10, c11, c12, c13,
                      c20, c21, c22, c23,
                      c30, c31, c32, c33 } <= 512'd23;
                end
                rdyGive = 0; */
            end
            pause : begin
                // do nothing here.
                // no values are changed.
                // the systolic array has been paused until the ZYctrls say to begin again.
            end
        endcase
    end
endmodule
```

```
//==========================ROM B=====================

`timescale 1ns / 1ps
`include "Constants.v"

module ROM_A(
input clk,
input [`ROWS-1:0] index0,index1,index2,index3,
output reg [`BIT_WIDTH-1:0] data0,data1,data2,data3
);

reg [`BIT_WIDTH-1:0] matrix [`ROWS-1:0][`COLS-1:0];
initial begin
//ROW 0             ROW 1               ROW 2               ROW 3
matrix[0][0] = 4'd1;   matrix[1][0] = 4'd5;   matrix[2][0] = 4'd9;   matrix[3][0] = 4'd13;
matrix[0][1] = 4'd2;   matrix[1][1] = 4'd6;   matrix[2][1] = 4'd10;   matrix[3][1] = 4'd14;
matrix[0][2] = 4'd3;   matrix[1][2] = 4'd7;   matrix[2][2] = 4'd11;   matrix[3][2] = 4'd15;
matrix[0][3] = 4'd4;   matrix[1][3] = 4'd8;   matrix[2][3] = 4'd12;   matrix[3][3] = 4'd6;
end

//recives memory adress and simply returns data

always @(posedge clk) begin
    data0 = matrix[0][index0];
    data1 = matrix[1][index1];
    data2 = matrix[2][index2];
    data3 = matrix[3][index3];
    //blocking because communication from top 2 cycles to recieve addresses, latch data, and send back to top module
    //blocking reduces top module to send adressses and recieve data in 1 cycle.
    //keep an eye out for future scheduling issues
end
endmodule

//==========================ROM B=====================

`timescale 1ns / 1ps
`include "Constants.v"

module ROM_B(
input clk,
input [`ROWS-1:0] index0,index1,index2,index3,
output reg [`BIT_WIDTH-1:0] data0,data1,data2,data3
);

reg [`BIT_WIDTH-1:0] matrix [`ROWS-1:0][`COLS-1:0];
initial begin
//ROW 0             ROW 1               ROW 2               ROW 3
matrix[0][0] = 4'd1;   matrix[1][0] = 4'd5;   matrix[2][0] = 4'd9;   matrix[3][0] = 4'd13;
matrix[0][1] = 4'd2;   matrix[1][1] = 4'd6;   matrix[2][1] = 4'd10;   matrix[3][1] = 4'd14;
matrix[0][2] = 4'd3;   matrix[1][2] = 4'd7;   matrix[2][2] = 4'd11;   matrix[3][2] = 4'd15;
matrix[0][3] = 4'd4;   matrix[1][3] = 4'd8;   matrix[2][3] = 4'd12;   matrix[3][3] = 4'd6;
end

 //recives memory adress and simply returns data

always @(posedge clk) begin
    data0 = matrix[index0][0]; //[t][0] //t as in cycle(t)
    data1 = matrix[index1][1]; //[t-1][1]
    data2 = matrix[index2][2]; //[t-2][2]
    data3 = matrix[index3][3]; //[t-3][3]
    //blocking because communication from top 2 cycles to recieve addresses, latch data, and send back to top module
    //blocking reduces top module to send adressses and recieve data in 1 cycle.
    //keep an eye out for future scheduling issues
end
endmodule
```

```verilog
//=============================RAM C===================

`timescale 1ns / 1ps
`include "Constants.v"

module RAM_C(
input clk,
input [`ROWS-1:0] index,
input [`BIT_WIDTH-1:0] data0,data1,data2,data3
);

reg [`BIT_WIDTH-1:0] matrix [`ROWS-1:0][`COLS-1:0];

always @(posedge clk) begin
//Writes one 4 words into an indexed ROW
     matrix[index][0] = data0;
     matrix[index][1] = data1;
     matrix[index][2] = data2;
     matrix[index][3] = data3;
    //blocking because communication from top 2 cycles to recieve addresses, latch data, and send back to top module
    //blocking reduces top module to send adressses and recieve data in 1 cycle.
    //keep an eye out for future scheduling issues
end
endmodule
```

```verilog
//=============TESTBENCH=================

`timescale 1ns / 1ps

module testbench_tb();
reg clk,reset,start;
wire [15:0] A_Out0,A_Out1,A_Out2,A_Out3;
wire [15:0] B_Out0,B_Out1,B_Out2,B_Out3;
wire [31:0] C_OUT0,C_OUT1,C_OUT2,C_OUT3;
wire ready_output,done;

Top_Module uutA(
.reset(reset),.clk(clk),.start(start),.execution_finished(done),
.a0x(A_Out0),.a1x(A_Out1),.a2x(A_Out2),.a3x(A_Out3),
.b0x(B_Out0),.b1x(B_Out1),.b2x(B_Out2),.b3x(B_Out3),
.c_data0(C_OUT0),.c_data1(C_OUT1),.c_data2(C_OUT2),.c_data3(C_OUT3),
.array_ready_output(ready_output)
);

always #10 clk = ~clk;

initial begin
clk = 0; reset=0; #10

reset = 1; #20;
reset = 0; #20;
start = 1; #20;
#20; // wait 1 cycle for things to settle

@(posedge done);
$finish;
#700; //everything is running now
$stop;
end

endmodule
```