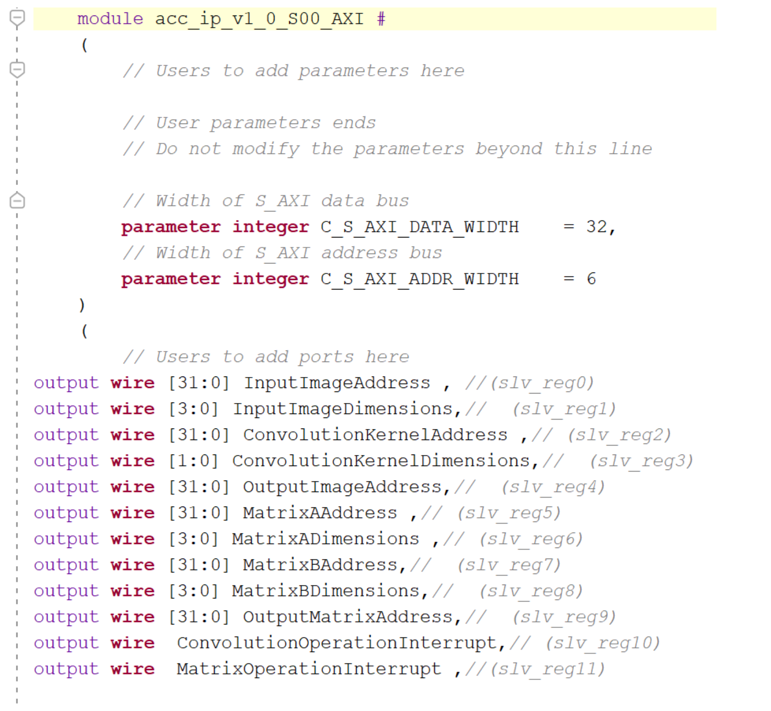
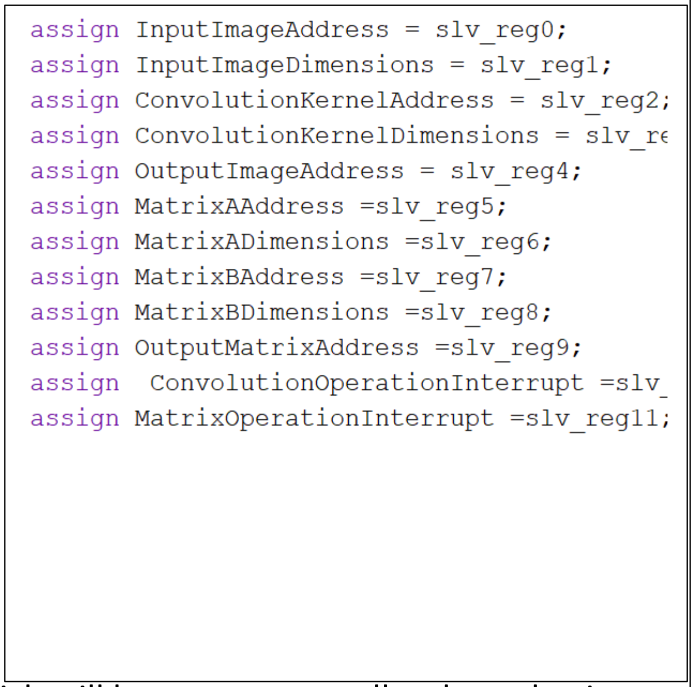
Explaining the Flow of SOC Code: From Input to Execution



```
module acc_ip_v1_0_S00_AXI #
    (
        // Users to add parameters here

        // User parameters ends
        // Do not modify the parameters beyond this line

        // Width of S_AXI data bus
        parameter integer C_S_AXI_DATA_WIDTH    = 32,
        // Width of S_AXI address bus
        parameter integer C_S_AXI_ADDR_WIDTH    = 6
    )
    (
        // Users to add ports here
output wire [31:0] InputImageAddress , //(slv_reg0)
output wire [3:0] InputImageDimensions,//  (slv_reg1)
output wire [31:0] ConvolutionKernelAddress ,// (slv_reg2)
output wire [1:0] ConvolutionKernelDimensions,//  (slv_reg3)
output wire [31:0] OutputImageAddress,//  (slv_reg4)
output wire [31:0] MatrixAAddress ,// (slv_reg5)
output wire [3:0] MatrixADimensions ,// (slv_reg6)
output wire [31:0] MatrixBAddress,//  (slv_reg7)
output wire [3:0] MatrixBDimensions,//  (slv_reg8)
output wire [31:0] OutputMatrixAddress,//  (slv_reg9)
output wire  ConvolutionOperationInterrupt,// (slv_reg10)
output wire  MatrixOperationInterrupt ,//(slv_reg11)
```



```
assign InputImageAddress = slv_reg0;
assign InputImageDimensions = slv_reg1;
assign ConvolutionKernelAddress = slv_reg2;
assign ConvolutionKernelDimensions = slv_re
assign OutputImageAddress = slv_reg4;
assign MatrixAAddress =slv_reg5;
assign MatrixADimensions =slv_reg6;
assign MatrixBAddress =slv_reg7;
assign MatrixBDimensions =slv_reg8;
assign OutputMatrixAddress =slv_reg9;
assign  ConvolutionOperationInterrupt =slv_
assign MatrixOperationInterrupt =slv_reg11;
```

At the start, CPU will fill data in slave registers , which will be sent to controller through wires.

CONTROLLER MUDULE EXPLANATION:

The module takes several inputs, including the clock (`M_AXI_ACLK`), reset (`M_AXI_ARESETN`), addresses for input data, kernel data, and output data, as well as dimensions for the input data, kernel data, and matrices.

The controller uses a finite state machine (FSM) with five states: `IDLE`, `READ_DATA`, `COMPUTE` , `WRITE_DATA`, and `DONE`.

In the `IDLE` state, the controller waits for an interrupt signal (`ConvolutionOperationInterrupt` or `MatrixOperationInterrupt`) to initiate a new operation. If a convolution interrupt is received, the operation type is set to convolution (`operation_type = 1'b1`). If a matrix interrupt is received, the operation type is set to matrix multiplication (`operation_type = 1'b0`).

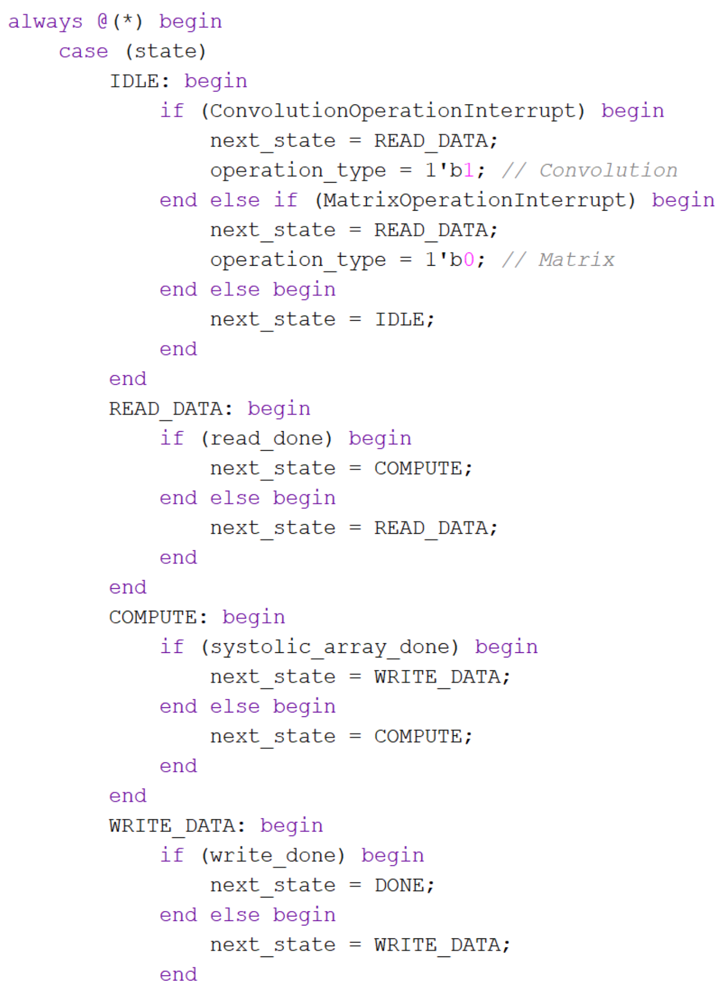
Upon receiving an interrupt, the controller transitions to the `READ_DATA` state, where it initiates an AXI transaction (`INIT_AXI_TXN = 1'b1`) to fetch the necessary input data and kernel/weight data from memory.

The addresses and dimensions for the input data and kernel/weight data are assigned based on the operation type (convolution or matrix multiplication).

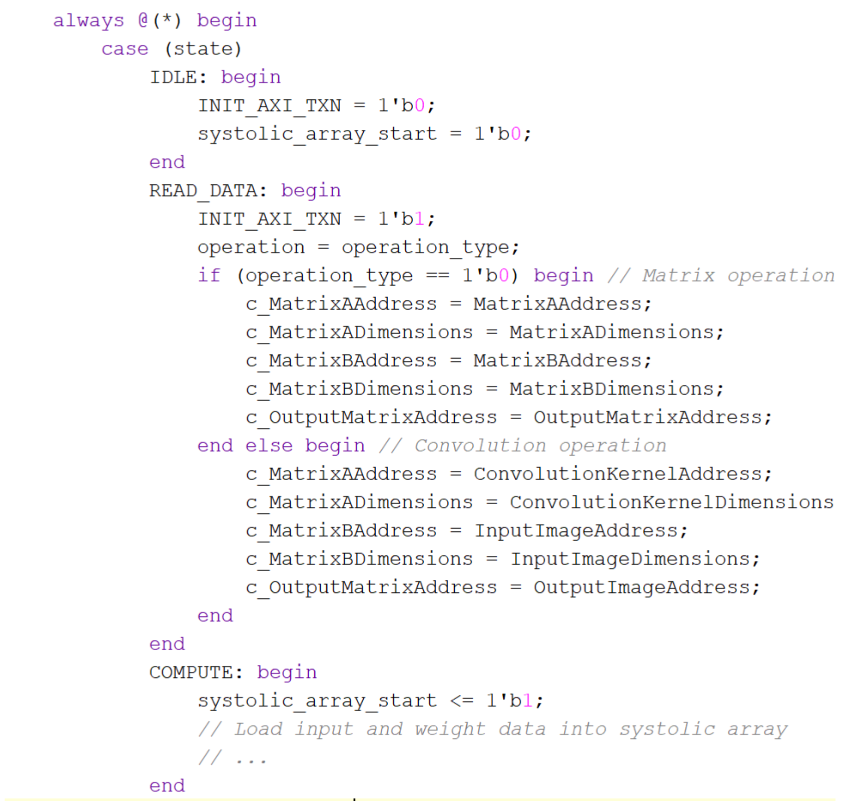
Once the data is read from memory (`read_done` signal is asserted), the controller transitions to the `COMPUTE` state, where it starts the systolic array computation (`systolic_array_start = 1'b1`).

In the `COMPUTE` state, the controller waits for the systolic array to complete the computation (`systolic_array_done` signal is asserted).
After the computation is done, the controller transitions to the `WRITE_DATA` state, where it likely initiates another AXI transaction to store the output data in memory (this part is not fully implemented in the provided code).

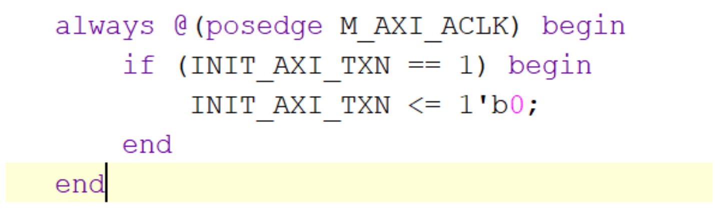
Once the output data is written (`write_done` signal is asserted), the controller transitions to the `DONE` state, indicating the completion of the operation.



```
always @(*) begin
    case (state)
        IDLE: begin
            if (ConvolutionOperationInterrupt) begin
                next_state = READ_DATA;
                operation_type = 1'b1; // Convolution
            end else if (MatrixOperationInterrupt) begin
                next_state = READ_DATA;
                operation_type = 1'b0; // Matrix
            end else begin
                next_state = IDLE;
            end
        end
        READ_DATA: begin
            if (read_done) begin
                next_state = COMPUTE;
            end else begin
                next_state = READ_DATA;
            end
        end
        COMPUTE: begin
            if (systolic_array_done) begin
                next_state = WRITE_DATA;
            end else begin
                next_state = COMPUTE;
            end
        end
        WRITE_DATA: begin
            if (write_done) begin
                next_state = DONE;
            end else begin
                next_state = WRITE_DATA;
            end
```



```
always @(*) begin
    case (state)
        IDLE: begin
            INIT_AXI_TXN = 1'b0;
            systolic_array_start = 1'b0;
        end
        READ_DATA: begin
            INIT_AXI_TXN = 1'b1;
            operation = operation_type;
            if (operation_type == 1'b0) begin // Matrix operation
                c_MatrixAAddress = MatrixAAddress;
                c_MatrixADimensions = MatrixADimensions;
                c_MatrixBAddress = MatrixBAddress;
                c_MatrixBDimensions = MatrixBDimensions;
                c_OutputMatrixAddress = OutputMatrixAddress;
            end else begin // Convolution operation
                c_MatrixAAddress = ConvolutionKernelAddress;
                c_MatrixADimensions = ConvolutionKernelDimensions
                c_MatrixBAddress = InputImageAddress;
                c_MatrixBDimensions = InputImageDimensions;
                c_OutputMatrixAddress = OutputImageAddress;
            end
        end
        COMPUTE: begin
            systolic_array_start <= 1'b1;
            // Load input and weight data into systolic array
            // ...
        end
```



```
always @(posedge M_AXI_ACLK) begin
    if (INIT_AXI_TXN == 1) begin
        INIT_AXI_TXN <= 1'b0;
    end
end
```

It is responsible for resetting the `INIT_AXI_TXN` signal to 0 (1'b0) on the next positive clock edge after it has been set to 1.

In the context of the provided controller code, the `INIT_AXI_TXN` signal is set to 1 in the `READ_DATA` state to initiate an AXI transaction for fetching input data and kernel/weight data from memory. By resetting it to 0 on the next clock cycle, the controller ensures that the AXI transaction is initiated only for a single cycle, preventing it from being continuously initiated.

EXPLANATION OF FSM IN AXI_MASTER

The FSM in this code has the following states:

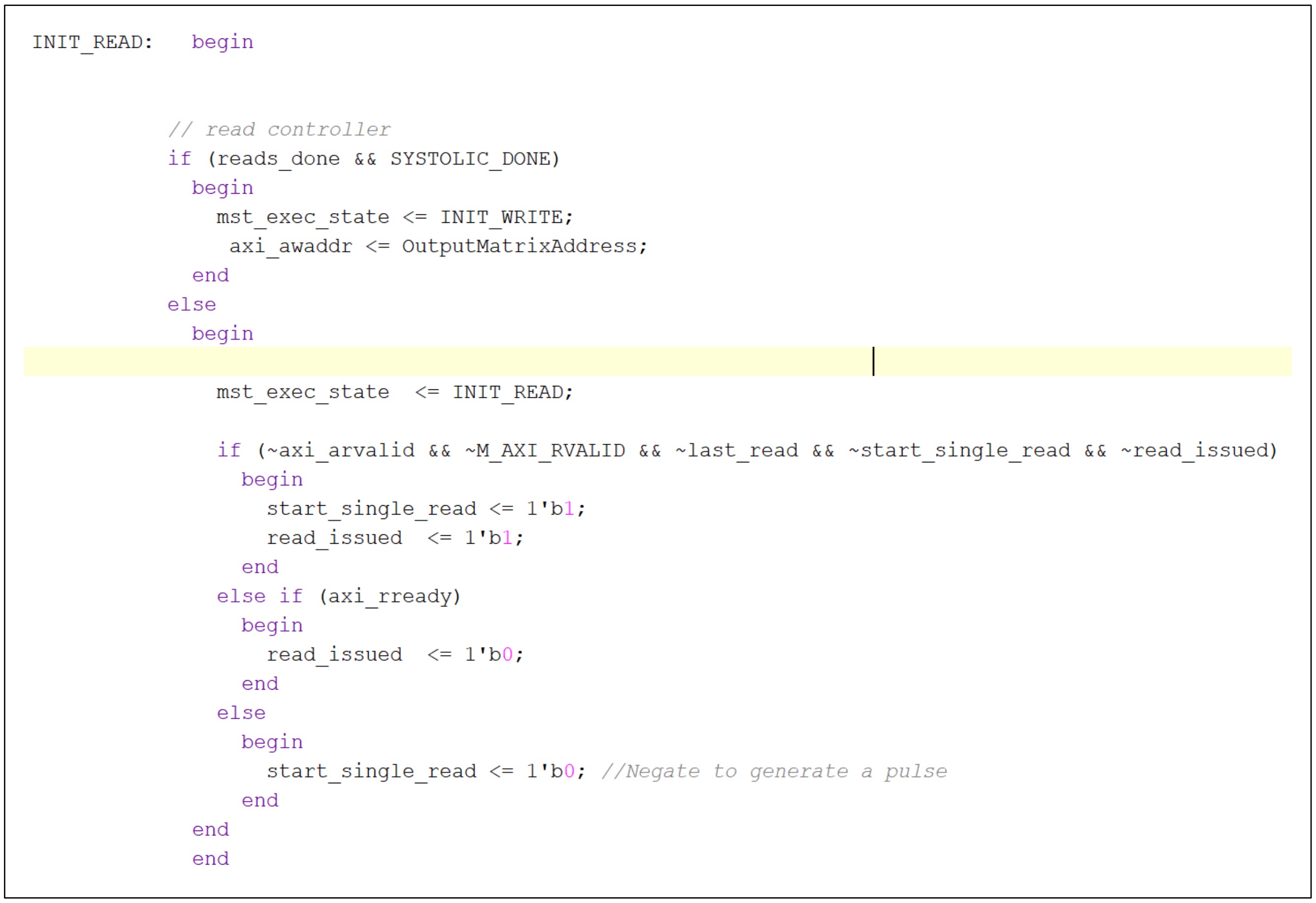
1. `IDLE`: This is the initial state where the module waits for an `INIT_AXI_TXN` pulse to initiate a new transaction.
2. `INIT_READ`: It reads data from the two input arrays, Matrix A and Matrix B, sequentially.
3. `INIT_WRITE`It writes the computed output to the specified output address.
4. `INIT_COMPARE`:

```
IDLE:
         // This state is responsible to initiate
         // AXI transaction when init_txn_pulse is asserted
           if ( init_txn_pulse == 1'b1 )
             begin
               mst_exec_state  <= INIT_READ;
                ARRAY_A_DIM <= c_MatrixADimensions ;
               ARRAY_A_ADDR <= c_MatrixAAddress ; // Starting address of Array A
              ARRAY_B_ADDR <= c_MatrixBAddress ;  // Starting address of Array B
               ARRAY_B_DIM <= c_MatrixBDimensions;
               ERROR <= 1'b0;
               compare_done <= 1'b0;
             end
           else
             begin
               mst_exec_state  <= IDLE;
             end
```

This code block represents the `IDLE` state of the FSM. In this state, the module waits for the `init_txn_pulse` signal to be asserted, indicating the start of a new transaction.

If `init_txn_pulse` is asserted, the following actions are performed:

1. The FSM state is transitioned to `INIT_READ`.
2. The dimensions of Matrix A (`ARRAY_A_DIM`) and Matrix B (`ARRAY_B_DIM`) are loaded from the input signals `c_MatrixADimensions` and `c_MatrixBDimensions`, respectively.
3. The starting addresses of Matrix A (`ARRAY_A_ADDR`) and Matrix B (`ARRAY_B_ADDR`) are loaded from the input signals `c_MatrixAAddress` and `c_MatrixBAddress`, respectively.



```
INIT_READ:   begin


           // read controller
           if (reads_done && SYSTOLIC_DONE)
             begin
               mst_exec_state <= INIT_WRITE;
                axi_awaddr <= OutputMatrixAddress;
             end
           else
             begin

               mst_exec_state  <= INIT_READ;

               if (~axi_arvalid && ~M_AXI_RVALID && ~last_read && ~start_single_read && ~read_issued)
                 begin
                   start_single_read <= 1'b1;
                   read_issued  <= 1'b1;
                 end
               else if (axi_rready)
                 begin
                   read_issued  <= 1'b0;
                 end
               else
                 begin
                   start_single_read <= 1'b0; //Negate to generate a pulse
                 end
             end
             end
```

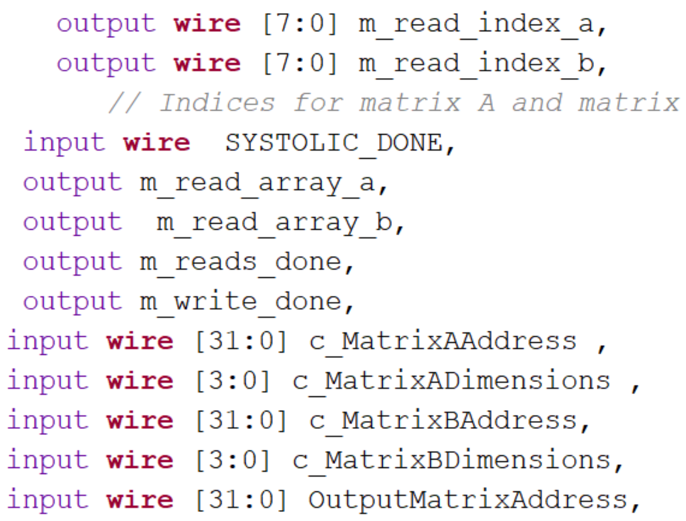
If both `reads_done` and `SYSTOLIC_DONE` signals are asserted, indicating that the read operation and systolic array computation are complete, the following actions are performed:

1. The FSM state is transitioned to `INIT_WRITE`.
2. The `axi_awaddr` signal is set to `OutputMatrixAddress`, which is the starting address for writing the output matrix.

If `reads_done` and `SYSTOLIC_DONE` are not both asserted, the following actions are performed:

1. The FSM state remains in `INIT_READ`.

PORTS:

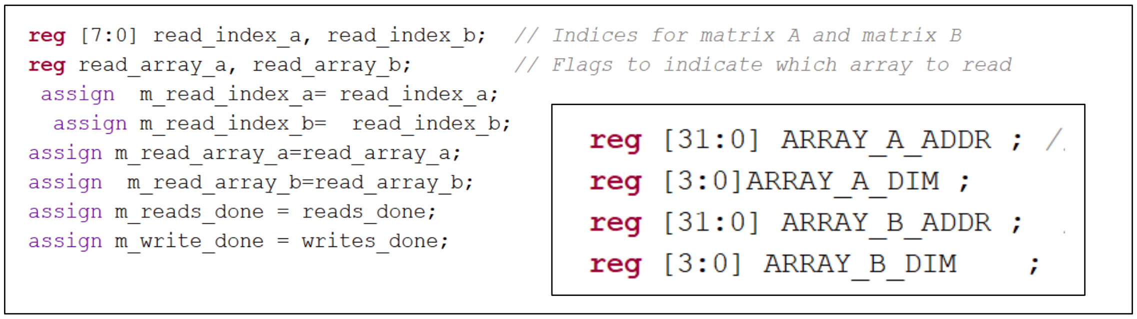


```
    output wire [7:0] m_read_index_a,
    output wire [7:0] m_read_index_b,
       // Indices for matrix A and matrix
  input wire  SYSTOLIC_DONE,
  output m_read_array_a,
  output  m_read_array_b,
  output m_reads_done,
  output m_write_done,
 input wire [31:0] c_MatrixAAddress ,
 input wire [3:0] c_MatrixADimensions ,
 input wire [31:0] c_MatrixBAddress,
 input wire [3:0] c_MatrixBDimensions,
 input wire [31:0] OutputMatrixAddress,
```

Ports:

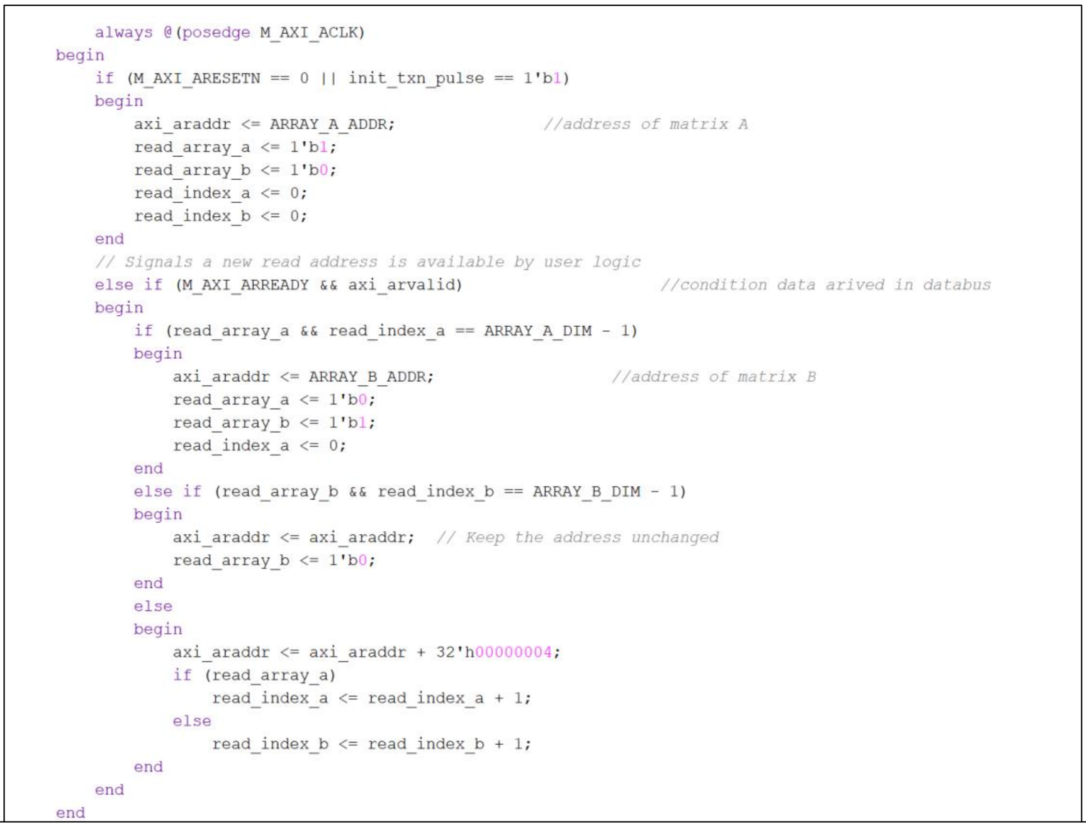
1. `output wire [7:0] m_read_index_a`: This is an output port that provides the current read index for Matrix A. The index is an 8-bit wide signal.
2. `output wire [7:0] m_read_index_b`: This is an output port that provides the current read index for Matrix B. The index is an 8-bit wide signal.
3. `output m_read_array_a`: This is an output port that indicates whether Matrix A is being read or not. It is a single-bit signal.
4. `output m_read_array_b`: This is an output port that indicates whether Matrix B is being read or not. It is a single-bit signal.
5. `output m_reads_done`: This is an output port that indicates when the read operation for both Matrix A and Matrix B is completed.
6. `output m_write_done`: This is an output port that indicates when the write operation for the output matrix is completed.
7. `input wire [31:0] c_MatrixAAddress`: This is an input port that provides the starting address of Matrix A in memory from controller.
8. `input wire [3:0] c_MatrixADimensions`: This is an input port that provides the dimensions (size) of Matrix A.
9. `input wire [31:0] c_MatrixBAddress`: This is an input port that provides the starting address of Matrix B in memory.
10. `input wire [3:0] c_MatrixBDimensions`: This is an input port that provides the dimensions (size) of Matrix B.

    These lines declare internal registers `read_index_a` and `read_index_b` to keep track of the current read indices for Matrix A and Matrix B, respectively. `read_array_a` and `read_array_b` are flags that indicate which array is being read. The output ports `m_read_index_a`, `m_read_index_b`, `m_read_array_a`, and `m_read_array_b` are assigned the values of these internal registers.



```
reg [7:0] read_index_a, read_index_b;  // Indices for matrix A and matrix B
reg read_array_a, read_array_b;        // Flags to indicate which array to read
 assign  m_read_index_a= read_index_a;
  assign m_read_index_b=  read_index_b;
assign m_read_array_a=read_array_a;
assign  m_read_array_b=read_array_b;
assign m_reads_done = reads_done;
assign m_write_done = writes_done;
```

```
reg [31:0] ARRAY_A_ADDR ; /
reg [3:0]ARRAY_A_DIM ;
reg [31:0] ARRAY_B_ADDR ;
reg [3:0] ARRAY_B_DIM    ;
```



```
    always @(posedge M_AXI_ACLK)
begin
    if (M_AXI_ARESETN == 0 || init_txn_pulse == 1'b1)
    begin
        axi_araddr <= ARRAY_A_ADDR;                //address of matrix A
        read_array_a <= 1'b1;
        read_array_b <= 1'b0;
        read_index_a <= 0;
        read_index_b <= 0;
    end
    // Signals a new read address is available by user logic
    else if (M_AXI_ARREADY && axi_arvalid)                    //condition data arived in databus
    begin
        if (read_array_a && read_index_a == ARRAY_A_DIM - 1)
        begin
            axi_araddr <= ARRAY_B_ADDR;                 //address of matrix B
            read_array_a <= 1'b0;
            read_array_b <= 1'b1;
            read_index_a <= 0;
        end
        else if (read_array_b && read_index_b == ARRAY_B_DIM - 1)
        begin
            axi_araddr <= axi_araddr;  // Keep the address unchanged
            read_array_b <= 1'b0;
        end
        else
        begin
            axi_araddr <= axi_araddr + 32'h00000004;
            if (read_array_a)
                read_index_a <= read_index_a + 1;
            else
                read_index_b <= read_index_b + 1;
        end
    end
end
```

It manages the read operation for Matrix A and Matrix B.

Initially, if the reset signal (`M_AXI_ARESETN`) is low or the `init_txn_pulse` is high, the following actions are performed:

1. The `axi_araddr` (AXI read address) is set to `ARRAY_A_ADDR`, which is the starting address of Matrix A.
2. The `read_array_a` flag is set to 1, indicating that Matrix A will be read first.
3. The `read_array_b` flag is set to 0, indicating that Matrix B will not be read initially.
4. The `read_index_a` and `read_index_b` are reset to 0.

The next block of code is executed when a new read address is available, indicated by the conditions `M_AXI_ARREADY` (AXI read address channel is ready) and `axi_arvalid` (AXI read address is valid).

If `read_array_a` is set (reading Matrix A) and `read_index_a` is equal to `ARRAY_A_DIM - 1` (reached the end of Matrix A), the following actions are performed:

1. The `axi_araddr` is set to `ARRAY_B_ADDR`, which is the starting address of Matrix B.
2. The `read_array_a` flag is set to 0, indicating that Matrix A has been read.
3. The `read_array_b` flag is set to 1, indicating that Matrix B will be read next.
4. The `read_index_a` is reset to 0.

If `read_array_b` is set (reading Matrix B) and `read_index_b` is equal to `ARRAY_B_DIM - 1` (reached the end of Matrix B), the following actions are performed:

1. The `axi_araddr` remains unchanged.
2. The `read_array_b` flag is set to 0, indicating that Matrix B has been read.

If neither of the above conditions is met (not at the end of either array), the following actions are performed:

1. The `axi_araddr` is incremented by 4 bytes (32'h00000004) to point to the next word in the current array.
2. If `read_array_a` is set, `read_index_a` is incremented by 1. Otherwise, `read_index_b` is incremented by 1.

```
  always @(posedge M_AXI_ACLK)
  begin
    if (M_AXI_ARESETN == 0 || init_txn_pulse == 1'b1)
      last_read <= 1'b0;

    //The last read should be associated with a read address ready response
    else if ((read_index == (ARRAY_A_DIM+ARRAY_B_DIM)) && (M_AXI_ARREADY) )
      last_read <= 1'b1;
    else
      last_read <= last_read;
  end

  always @(posedge M_AXI_ACLK)
  begin
    if (M_AXI_ARESETN == 0 || init_txn_pulse == 1'b1)
      reads_done <= 1'b0;

    //The reads_done should be associated with a read ready response
    else if (last_read && M_AXI_RVALID && axi_rready)
      reads_done <= 1'b1;
    else
      reads_done <= reads_done;
    end
```

This `always` block sets the `last_read` flag when the read operation for both Matrix A and Matrix B is completed If the sum of the read indices (`read_index`) is equal to the sum of the dimensions of Matrix A and Matrix B (`ARRAY_A_DIM + ARRAY_B_DIM`), and the AXI read address channel is ready (`M_AXI_ARREADY`), the `last_read` flag is set to 1, indicating that the last read operation has been completed. And read_done signal become high.

```
module input_memory #(parameter M_ROW = 9, parameter M_COL = 9 )(
    input M_AXI_ACLK,
    input M_AXI_ARESETN,
    input init_txn_pulse,
    input M_AXI_RVALID,
    input axi_rready,
    input [31:0] M_AXI_RDATA,
    input read_array_a,
    input [7:0] read_index_a,
    input systolic_array_start,
    output reg [M_COL*32-1:0] in_data,
    output reg systolic_array_done
);

reg [31:0] array_a [0:M_COL - 1] [0:M_COL - 1]; // 2D array to store elements of matrix A
integer row, col;
reg [7:0] row_counter;

always @(posedge M_AXI_ACLK) begin
    if (M_AXI_ARESETN == 0 || init_txn_pulse == 1'b1) begin
        for (row = 0; row < M_COL; row = row + 1) begin
            for (col = 0; col < M_COL; col = col + 1) begin
                array_a[row][col] <= 0;
            end
        end
        row_counter <= 0;
        systolic_array_done <= 0;
    end
    else if (M_AXI_RVALID && axi_rready) begin
        if (read_array_a) begin
            array_a[read_index_a / M_COL][read_index_a % M_COL] <= M_AXI_RDATA;
        end
    end
```

The `input_memory` module is responsible for storing the elements of Matrix A in the `array_a` register array as they are received from the AXI read channel. The `read_array_a` and `read_index_a` signals are used to determine when to store the data and at which location in the `array_a`.

Here's how the `input_memory` module stores Matrix A:

1. The module is parameterized with `M_ROW` and `M_COL`, which represent the dimensions of the input matrix. In this case, both are set to 9, indicating a 9x9 matrix.
2. A two-dimensional register array `array_a` is declared with dimensions `[0:M_COL - 1][0:M_COL - 1]`. This array is used to store the elements of Matrix A.
3. The `always` block is triggered on the positive edge of the `M_AXI_ACLK` clock signal.
4. Inside the `always` block, there is an `if` condition that checks for reset (`M_AXI_ARESETN == 0`) or the `init_txn_pulse` signal. If either of these conditions is true, the `array_a` is initialized with zeros using nested `for` loops, the `row_counter` is reset to 0, and the `systolic_array_done` signal is set to 0.
5. The next `else if` condition checks if the AXI read data channel is valid (`M_AXI_RVALID`) and the module is ready to accept data (`axi_rready`).
6. If the `read_array_a` signal is asserted (indicating that Matrix A is being read), the module stores the received `M_AXI_RDATA` (32-bit data from the AXI read channel) into the corresponding element of the `array_a` array, using the `read_index_a` to calculate the row and column indices (`read_index_a / M_COL` and `read_index_a % M_COL`, respectively).

```
reg [31:0] array_b [0:M_ROW - 1] [0:M_ROW - 1]; // 2D array to store elements of matrix b
integer row, col;
reg [7:0] col_counter;

always @(posedge M_AXI_ACLK) begin
    if (M_AXI_ARESETN == 0 || init_txn_pulse == 1'b1) begin
        for (row = 0; row < M_ROW; row = row + 1) begin
            for (col = 0; col < M_ROW; col = col + 1) begin
                array_b[row][col] <= 0;
            end
        end
        col_counter <= 0;
        systolic_array_done <= 0;
    end
    else if (M_AXI_RVALID && axi_rready) begin
        if (read_array_b) begin
            array_b[read_index_b / M_COL][read_index_b % M_COL] <= M_AXI_RDATA;
        end
    end
    else if (systolic_array_start) begin
        if (col_counter < M_COL) begin
            wt_data <= 0;
            for (row = 0; row < M_ROW; row = row + 1) begin
                wt_data <= {wt_data, array_b[row][col_counter]};
            end
            col_counter <= col_counter + 1;
        end
        else begin
            systolic_array_done <= 1;
        end
    end
end
```

The `weight_memory` module is responsible for storing the elements of Matrix B in the `array_b` register array as they are received from the AXI read channel. The `read_array_b` and `read_index_b` signals are used to determine when to store the data and at which location in the `array_b`.