ENGR 100 Section 270 Fall 25'

# BLUE CHIP Engineering

Exemplifying Excellence in Engineering

Hayagreev Jeyandran
Henry LeCompte
Raymond Shattuck
Tyler Clark
e100-bluechip-fa25@googlegroups.com

---

## Code Documentation - Matrix Multiplier Verilog

This document contains an explanation of the hardware description language used to multiply two fixed-size, 3x3 matrices in our ENGR 100-270 final project. It outlines the Verilog modules, presents the code for each module, describes the processes and state machines, and finally, presents a top-down structure of how they integrate.

Please direct any questions about the code and/or documentation to the corresponding authors, `Henry Lecompte` or `Hayagreev Jeyandran`.

---

## Chapter 1 Directory Structure

The project directory has been made open-source and is free to download from GitHub (recommended) or Google Drive (the latter is accessible only for users within the University of Michigan).

Within the directory, there exists a `src` folder, which contains all the Verilog modules and testbenches. The root has some critical project-wide files used by Quartus Prime for successful compilation and programming, including:

- `SignalTapConfig.stp` (configuration for on-chip debugging with the logic analyser)
- `Matrix_mult.qpf` (main project file)
- `mat_process.sdc` (clock definitions and design constraints)
- `mat_process.qsf` (project-level implementation settings).

If you want to recreate our logic, you must download these files as well, as compilation and programming will fail without them.

---

## Chapter 2 Verilog Notes

This section documents some common properties of the Verilog HDL; it will be helpful to refer back to these as you encounter them in our code in Sections 3 onwards.

### 2.1 Signal Types

1. **NETS**
    - Represent a physical connection to the FPGA. They cannot store values - must be driven by a signal, continuously.
    - Example: `wire, tri` (tri indicates that multiple sources drive a signal)
    - You cannot assign a value to a net in a procedural block.

2. **VARIABLES**
    - Represents storage in Verilog. They hold a value until a procedural block of code changes them.
    - Variables are used in `always` and `initial` blocks.
    - Example: `reg[MSBIndex:LSBIndex], bit, integer, logic`

**2.2 Assignment Types**

1. **Continuous Assignment**
    - Used to couple a wire/net and drive the signal only. Remember, wire/net cannot be assigned procedurally.
    - Always active - represents **combinational logic.**
    - As soon as RHS changes, the LHS net is updated.
    - Used outside procedural blocks.
    - `assign x = ~(A & B)| ~(B & C)`
2. **Procedural Assignment**
    - Used inside a procedural block (`always/initial`)
    - The LHS should be a variable; it cannot be used on a net.
    - **Blocking** (=) is executed immediately, used for combinational logic (like a high-level programming assignment). However, certain applications will demand the use of a blocking assignment within clocked procedural logic, such as `always @(posedge clksignal)`.
    - Example:
```
bit x;
always @(*) begin
    x = ~(A & B) | ~(B & C); //immediate updates
end
```
    - **Non-blocking** (<=) is executed at the end of the time step, after all blocking updates have been completed, but before the start of the next time step. Used in sequential logic - requires proper clocked logic. A non-blocking assignment is usually not used when the sensitivity list for the procedural loop is a wildcard (`*`).
    - Example:
```
bit x;
always @(posedge_clk) begin
    x <= b;
end
```

**2.3 Procedural Blocks**

1. **Initial blocks -** run at time t = 0, only used in simulations, and do not get synthesised into real hardware.
2. **Always blocks** - always blocks use triggers to execute procedural code. For example:
```
always @(*) // executes all the time - combinational logic
always @(posedge_clk) // Positive edge triggered
always @(negedge_clk) // Negative edge triggered
```

`Always @(enable or d) // Level triggered`
The parenthesised variables should contain a sensitivity list separated with an `or` or commas (`,`).

### 2.4 Miscellaneous notes

a. All identifiers in module signatures are automatically declared as wires; therefore, they must be continuously driven unless we want them to be a register/bit, which must be specified explicitly as `input/output` **`reg`** `<name>`.
b. localparams are constants used to codename states. Instead of 0, 1, etc, we use READ_A, READ_B as constants that are integers in memory for the purposes of readability.

---

## Chapter 3 Module 1: Clock Enable

### 3.1 Source Code

```
01 `timescale 1ns/1ps
02
03 module clk_enable(
04     input clk,
05     input rst_n,
06     output reg clk_en,
07     output clk_out
08 );
09 reg[15:0] counter;
10 assign clk_out = counter[15];
11
12 always @(posedge clk, negedge rst_n)
13 begin
14     if (!rst_n) begin // reset loop
15         counter <= 16'b0;
16         clk_en <= 1'b0;
17     end else begin
18         if (counter == 16'hFFFF) begin
19             clk_en <= 1'b1;
20             counter <= 16'b0;
21         end else begin
22             clk_en <= 1'b0;
23             counter <= counter + 16'b1;
24         end
25     end
26 end
```

### 3.2 Documentation

- Line 01: Timescale sets the units for simulation but is not synthesised into hardware.
- Lines 03-08: This consists of the module declaration. **`Clk`** is the original FPGA oscillator clock, running at 50MHz. **`clk_out`** is only used for external clock information and is not on the clock tree. The reset signal, **`rst_n`**, is active low and wired to the Arduino for a single point of control.
- Line 12: The procedural loop is usually triggered for every rising edge of the source clock, but since we need to reset the FPGA during arbitrary states of the clock, we included **`negedge rst_n`** in the sensitivity list.
- Lines 14-16: When the reset is triggered, the counter and enable clock are set to 0.
- Lines 17-25: Otherwise, we increment the counter until it reaches **`FFFF`** (after $2^{16}$ cycles of **`clk`**), and schedule a non-blocking update to set **`clk_en`** to **`HIGH`**. After **`FFFF`**, the counter is reset to 0 for the process to repeat.

**Function summary:** Divides the source clock of the FPGA by a **factor of $2^{16}$** for it to be compatible with the Arduino's loop().
**State Machine:** Clock enable is purely sequential and does not have an FSM.

---

## Chapter 4 Module 2: Matrix Accumulate

### 4.1 Source Code

```
01 module mat_accum(
02     input               i_clk,
03     input               i_clk_e,
04     input               i_rst_n,
05     // ---------
06     input signed [7:0]  s_axis_data,
07     output              s_axis_ready,
08     input               s_axis_valid,
09     input               s_axis_last,
10     // ---------
11     output signed [7:0] m_axis_res_data,
12     input               m_axis_res_ready,
13     output              m_axis_res_valid,
14     output reg          m_axis_res_last
15 );
16 
17 localparam READING = 0;
18 localparam OUTPUTTING = 1;
19 
20 reg [0:0] current_state;
```

```
21
22 reg signed [7:0] matrix [8:0];
23
24 reg [2:0] iteration;
25 reg [3:0] idx;
26
27
28 assign m_axis_res_data  = matrix[idx];
29 assign m_axis_res_valid = current_state == OUTPUTTING;
30
31 assign s_axis_ready = current_state == READING;
32
33 always @(posedge i_clk, negedge i_rst_n)
34 begin
35     if (!i_rst_n) begin
36         iteration      <= 0;
37         idx            <= 0;
38         current_state  <= READING;
39     end else
40     if (i_clk_e) begin
41         case (current_state)
42             READING: begin
43                 if (s_axis_valid) begin
44                     if (idx == 8) begin
45                         idx       <= 0;
46                         iteration <= iteration + 1;
47                         if (iteration == 2) begin
48                             current_state <= OUTPUTTING;
49                         end
50                     end
51                     else begin
52                         idx <= idx + 1;
53                     end
54
55                     if (iteration == 0) begin
56                         matrix[idx] <= s_axis_data;
57                     end else begin
58                         matrix[idx] <= matrix[idx] + s_axis_data;
59                     end
60                 end
61             end
62
63             OUTPUTTING: begin
64                 if (idx == 8) begin
65                     idx           <= 0;
66                     iteration     <= 0;
67                     current_state <= READING;
```

```
68                  end
69                  else begin
70                      idx <= idx + 1;
71                  end
72              end
73          endcase
74      end
75  end
76
77  endmodule
```

### 4.2 Documentation

- 02-04: Clock and reset inputs. **`i_clk`** is the FPGA's master clock, and **`i_clk_e`** is the reduced clock.
- 06-09: AXIStream signals for when the FPGA is in the **`READING`** state (slave)
- 11-13: Master AXIStream signals for FPGA to stream the result matrix back to the Arduino.
- 20-25: Counters for the module to work properly. **`current_state`** holds the integer defined in lines 17-18, **`matrix`** is a Q4.4 array of 9 elements in row-major order. **`Iteration`** defines the matrix count, while **`idx`** is used to index over individual matrix elements.
- 28-31: Wires are managed here. **`m_axis_res_valid`** will correspond to whether the FPGA is outputting, and **`s_axis_ready`** will correspond to whether the FPGA is reading.
- 40-41: Only if the reduced **`i_clk_e`** is **`HIGH`** should the FPGA's logic execute. Specifically, it splits into cases depending on the state.
- Case 1: **`reading`** (42-62): **`idx`** is incremented on every clock cycle until it hits 8. If **`iteration`** is 0, then the first outer product is being accumulated, which necessitates replacement in line 45. We need to add subsequent outer products to the existing elements in the result, as seen on line 58.
- When **`idx`** is 8, the **`iteration`** increments by 9 elements of the previous outer product that have been entered.
- Case 2: **`outputting`** (63-72): **`idx`** needs to be incremented. On line 28, **`m_axis_res_data`** was set to be driven by **`matrix[idx]`**, so on every edge of **`i_clk_e`**, it will transmit the next element in the result matrix to the Arduino.

**Function summary:** Accumulates outer products into a result matrix and streams them back to the Arduino synchronously.

[Finite State Machine Diagram]

---

## Chapter 5 Module 3: Matrix Deinterleave

### 5.1 Source Code

```
01 module mat_deinterleave(
02     input          i_clk,
03     input          i_clk_e,
04     input          i_rst_n,
05     // ---------
06     input          s_axis_valid,
07     output         s_axis_ready,
08     // ---------
09     output         m_axis_a_valid,
10     input          m_axis_a_ready,
11     // ---------
12     output         m_axis_b_valid,
13     input          m_axis_b_ready
14 );
15 
16 reg [1:0] data_count;
17 
18 reg out_stream;
19 
20 assign s_axis_ready = out_stream ? m_axis_a_ready : m_axis_b_ready;
21 
22 assign m_axis_b_valid = out_stream ? 0 : s_axis_valid;
23 assign m_axis_a_valid = out_stream ? s_axis_valid : 0;
24 
25 
26 always @(posedge i_clk, negedge i_rst_n)
27 begin
28     if (!i_rst_n) begin
29         data_count <= 0;
30         out_stream <= 0;
31     end else
32     if (i_clk_e) begin
33         if (s_axis_valid && s_axis_ready && data_count != 2) begin
34             data_count <= data_count + 1;
35         end
36         else if (data_count == 2) begin
37             out_stream <= out_stream + 1;
38             data_count <= 0;
39         end
40     end
41 end
42 
43 endmodule
```

**5.2 Documentation**

- 06-13: There is one pair of inputs for the data being entered in (slave mode for this module), and two pairs of outputs to indicate to downstream modules whether the data is from B or A (master mode for this module).
- 16-18: **data_count** iterates from 0-2 and is used to indicate which element, either in a row of B or a column of A, is being entered into the partial multiplier. **out_stream** is the state variable, which can either be 0 (matrix B) or 1 (matrix A).
- 20-23: These lines are critical since they are part of the core multiplexing logic. They allow if-condition-like behaviour outside the procedural loop.
    - 20: Is equivalent to

```
if (out_stream) begin
    assign s_axis_ready = m_axis_a_ready;
end else begin
    assign s_axis_ready = m_axis_b_ready;
else
```

    - Of course, this is illegal Verilog because conditionals are only allowed inside procedural blocks. This is used to illustrate the use of the ternary operator (**?**).
    - 21-22: Is equivalent to

```
if (out_stream) begin
    assign m_axis_b_valid = 0;
    assign m_axis_a_valid = s_axis_valid;
end else begin
    assign m_axis_b_valid = s_axis_valid;
    assign m_axis_a_valid = 0;
```

- 20-23 are casting upstream valid/ready inputs to the downstream inputs, depending on the state. For example, since **`out_stream = 0`** corresponds to matrix B, **`m_axis_a_valid`** should be **`LOW,`** and **`m_axis_b_valid`** should be the same as whether all incoming data is valid, since there is one unified stream carrying the interleaved matrices (**`s_axis_valid`**).
- 26 onwards: Procedural logic used to control output state. For the first sequence of **`data_coun`**t from 0 through 2, we are reading in the first row of B, then the first column of A for the next 0-2 of **`data_count`**, and so on.
- **`out_stream`** is a single bit, and incrementing it by 1 constantly on line 37 will cause it to overflow and alternate between 0 and 1 (matrix B and matrix A).

**Function summary:** two-state multiplexer telling the downstream module which matrix is active during that clock cycle, based on **`out_stream`** and **`data_count`**. Does not touch the data at all.

**Finite State Machine Diagram**

---

# Chapter 6 Module 4: Matrix Partial Multiply

### 6.1 Source Code

```
module mat_partial_mult(
    input               i_clk,
    input               i_clk_e,
    input               i_rst_n,
    input signed [7:0] i_a_num,
    input               i_a_num_valid,
    input signed [7:0] i_b_num,
    input               i_b_num_valid,
    output              o_a_ready,
    output              o_b_ready,
    input               i_res_ready,
    output reg signed [7:0] o_res_data,
    output reg          o_res_valid,
    output reg          o_res_last
);

localparam READ_A = 0;
localparam READ_B = 1;
localparam MULTIPLY = 2;
localparam OUTPUT_MULTIPLY = 3;

reg signed [7:0] col_a [0:2];
reg signed [7:0] row_b [0:2];
reg [15:0] tmp;
reg [1:0] a_count;
reg [1:0] b_count;
reg [2:0] current_state;
reg [2:0] iterations;
reg [2:0] mult_iterations;

assign o_a_ready = current_state == READ_A;
assign o_b_ready = current_state == READ_B;

always @(posedge i_clk, negedge i_rst_n)
begin
    if (!i_rst_n) begin
         a_count <= 0;
         b_count <= 0;
         current_state <= READ_B;
         iterations <= 0;
         o_res_last <= 0;
         mult_iterations <= 0;
         o_res_valid <= 0;
```

```
44      end
45      if(i_clk_e) begin
46          case (current_state)
47              READ_B: begin
48                  o_res_valid <= 0;
49                  o_res_last <= 0;
50                  if (i_b_num_valid) begin
51                      row_b[b_count] <= i_b_num;
52                      if (b_count == 2) begin
53                          current_state <= READ_A;
54                          b_count <= 0;
55                      end else begin
56                          b_count <= b_count + 1;
57                      end
58                  end
59              end
60              READ_A: begin
61                  if (i_a_num_valid) begin
62                      col_a[a_count] <= i_a_num;
63                      if (a_count == 2) begin
64                          current_state <= OUTPUT_MULTIPLY;
65                          a_count <= 0;
66                      end else begin
67                          a_count <= a_count + 1;
68                      end
69                  end
70              end
71
72              OUTPUT_MULTIPLY: begin
73                  if (i_res_ready) begin
74                      o_res_valid <= 1;
75
76                      tmp = col_a[mult_iterations] * row_b[a_count];
77
78                      o_res_data <= tmp[11:4];
79                      if (a_count == 2) begin
80                          a_count <= 0;
81
82                          if (mult_iterations == 2) begin
83                              mult_iterations <= 0;
84                              current_state <= READ_B;
85                              if (iterations == 2) begin
86                                  iterations <= 0;
87                                  o_res_last <= 1;
88                              end else begin
89                                  iterations <= iterations + 1;
90                              end
```

```
91                          end
92                          else begin
93                              mult_iterations <= mult_iterations + 1;
94                          end
95                      end
96                      else begin
97                          a_count <= a_count + 1;
98                      end
90                  end
100             end
101         endcase
102     end
103 end
104
105 endmodule
```

**6.2 Documentation**

- As seen on lines 5, 6, 7, and 8, this module finally splits the input data into instances of matrix A's entries and matrix B's entries.
- Lines 22-29 feature important declarations. Specifically, signed 8-bit arrays are used to hold **`col_a`** and **`row_b`**, and a **`tmp`** variable is declared to hold the 16-bit result of fixed-point multiplication. **`a_count`** and **`b_count`** hold the index numbers of the column and row entries of A and B, respectively, and **`current_state`** is the state variable. The distinction between **`iterations`** and **`mult_iterations`** is a subtle one:
    - **`iterations`** counts how many total rows/columns were read in, and increments for each complete partial multiplication.
    - **`mult_iterations`** is used to index over the column of A during multiplication, and **`a_count`** is used dually to index over each entry of **`row_b.`**
- The upstream ready signal is set to reflect the current state on lines 31 and 32, similar to the accumulate module.
- Lines 47-59: **`READ_B`** increments **`b_count`** and stores the input data in **`row_b[b_count]`** until **`b_count = 2`**, reading in the three numbers in the row of b. Note that **`i_b_num`** and **`i_a_num`** come from the same deinterleaved stream, but their interpretation as elements of **`row_b`** and **`col_a`** changes because of the **`i_a_num_valid/i_b_num_valid`** signal sent by the deinterleaver. After reading in the row, the state is set to change to read A.
- Lines 60-71: **`READ_A`** functions similarly, but at the end, the state transitions to **`OUTPUT_MULTIPLY`**.
- Lines 72 onwards: Going through the multiply and output logic step-by-step is critical. When **`i_res_ready`** is HIGH on the positive edge of **`i_clk_e`**, we multiply **`col_a[mult_iterations]`** and **`row_b[a_count]`**, and according to fixed-point logic, we take the middle 8 bits and assign them to the output register **`o_res_data`**.

- The logic on lines 82-98 is best illustrated using a time-chart of instructions per positive edge.

| State | iterations | mult_iterations | a_count | b_count | action |
|---|---|---|---|---|---|
| READ_B | 0 | 0 | 0 | 0 | store row_b |
| READ_B | 0 | 0 | 0 | 1 | store row_b |
| READ_B | 0 | 0 | 0 | 2 | store row_b |
| READ_A | 0 | 0 | 0 | 0 | store col_a |
| READ_A | 0 | 0 | 1 | 0 | store col_a |
| READ_A | 0 | 0 | 2 | 0 | store col_a |
| MULT | 0 | 0 | 0 | 0 | col_a[0] * row_b[0] |
| MULT | 0 | 0 | 1 | 0 | col_a[0] * row_b[1] |
| MULT | 0 | 0 | 2 | 0 | col_a[0] * row_b[2] |
| MULT | 0 | 1 | 0 | 0 | … |
| MULT | 0 | 2 | 0 | 0 | … (Products are streamed in parallel on line 78) |
| READ_B | 1 | 0 | 0 | 0 | … repeat reading B again |
| … (Second outer product) | | | | | |
| MULT | 1 | 2 | 0 | 0 | … (multiply second row of B with second column of A) |
| READ_B | 2 | 0 | 0 | 0 | … (repeat reading B again) |
| … (Third outer product) | | | | | |
| MULT | 2 | 2 | 0 | 0 | (multiply third row of B with third column of A, all done!) |

  As seen above, this logic ensures that the outer products get computed correctly, and get computed 3 times (for the 3 rows of B and 3 columns of A that are interleaved).
- Line 78 sets the multiplication result to **`o_res_data`**, which is tied to the module's output on line 12.

**Function summary:** Reads in a row of B, then a column of A, and streams the outer product matrix's elements column by column. This is done for all 3 rows and columns of A, so the full matrix is multiplied correctly here.

[Finite State Machine Diagram]

---

### Chapter 7 Module 5: Integrate - Final Module

### 7.1 Source Code

```
 1 | module mat_process(
 2 |     input axis_clk,
 3 |     input axis_rst_n,
 4 |     input [7:0] S_AXIS_DATA,
 5 |     output S_AXIS_READY,
 6 |     input S_AXIS_VALID,
 7 |     output [7:0] M_AXIS_RES,
 8 |     output M_AXIS_VALID,
 9 |     output M_AXIS_LAST,
10 |     input M_AXIS_READY,
11 |     output clk_out
12 |    );
13 |
14 |
15 | wire axis_a_ready, axis_a_valid, axis_b_ready, axis_b_valid;
```

```
16 | wire [7:0] partial_mult_data;
17 | wire partial_mult_ready, partial_mult_valid, partial_mult_last;
18 | wire axis_clk_e;
19 |
20 |
21 | clock_enable clk_gate(
22 |  .clk(axis_clk),
23 |  .rst_n(axis_rst_n),
24 |  .clk_en(axis_clk_e),
25 |  .clk_out(clk_out)
26 | );
27 |
28 | mat_deinterleave deinter (
29 |     .i_clk(axis_clk),
30 |     .i_clk_e(axis_clk_e),
31 |     .i_rst_n(axis_rst_n),
32 |     // ---------
33 |     .s_axis_valid(S_AXIS_VALID),
34 |     .s_axis_ready(S_AXIS_READY),
35 |     // ---------
36 |     .m_axis_a_valid(axis_a_valid),
37 |     .m_axis_a_ready(axis_a_ready),
38 |     // ---------
39 |     .m_axis_b_valid(axis_b_valid),
40 |     .m_axis_b_ready(axis_b_ready)
41 | );
42 |
43 | mat_partial_mult multiplier(
44 |     .i_clk(axis_clk),
45 |   .i_clk_e(axis_clk_e),
46 |     .i_rst_n(axis_rst_n),
47 |     .i_a_num(S_AXIS_DATA),
48 |     .i_a_num_valid(axis_a_valid),
49 |     .i_b_num(S_AXIS_DATA),
50 |     .i_b_num_valid(axis_b_valid),
51 |     .o_a_read(axis_a_ready),
52 |     .o_b_read(axis_b_ready),
53 |     .i_res_ready(partial_mult_ready),
54 |     .o_res_data(partial_mult_data),
55 |     .o_res_valid(partial_mult_valid),
56 |     .o_res_last(partial_mult_last)
57 | );
58 |
59 | mat_accum accumulator(
60 |  .i_clk(axis_clk),
61 |  .i_clk_e(axis_clk_e),
62 |  .i_rst_n(axis_rst_n),
```

```
63 |  // ---------
64 |  .s_axis_data(partial_mult_data),
65 |  .s_axis_ready(partial_mult_ready),
66 |  .s_axis_valid(partial_mult_valid),
67 |  .s_axis_last(partial_mult_last),
68 |  // ---------
69 |  .m_axis_res_data(M_AXIS_RES),
70 |  .m_axis_res_ready(M_AXIS_READY),
71 |  .m_axis_res_valid(M_AXIS_VALID),
72 |  .m_axis_res_last(M_AXIS_LAST)
73 | );
74 | endmodule
```

### 7.2 Documentation

- The process module is responsible for linking together all the modules from 1 through 4 seen in chapters 3 through 6.
- In the module signature from lines 1 to 13, the interface with the Arduino Mega and the FPGA's Clock Oscillator are present. The Arduino interface includes AXIStream data (eg: **S_AXIS_READY, M_AXIS_VALID, axis_rst_n**), as well as the data in the form of signed 8-bit Q4.4 (**S_AXIS_DATA, M_AXIS_RES**)
- For passing signals within the modules, wires are declared from lines 15-18.
- Module 1: clk_gate produces **i_clk_e** which is used by all other modules.
- Module 2: deinter drives the intermediate signals **axis_a_valid, axis_a_ready, axis_b_valid, axis_b_ready** which are used to split **S_AXIS_DATA** into **i_a_num** and **i_b_num** by the multiplier.
- The multiplier outputs **partial_mult_data,** and associated AXIStream signals, which are then used by the accumulator module.
- Lastly, the accumulator module outputs **M_AXIS_RES, M_AXIS_READY, M_AXIS_VALID,** and **M_AXIS_LAST** back to the Arduino for it to display the matrix result.
- The uppercase wires and variables are connected to the Arduino, while the lowercase variables and wires link intermediate modules.

**Function Summary:** Overall integration of all the modules for synthesis in hardware.

[Top-down Structure Diagram]

---

## Chapter 8 Matrix Full Multiply (Alternative pipeline)

### 8.1 Source Code

```
1  module mat_mult(
2      input               i_clk,
3      input               i_clk_e,
```

```
 4      input              i_rst_n,
 5      input [7:0]        i_a_num,
 6      input              i_a_num_valid,
 7      input [7:0]        i_b_num,
 8      input              i_b_num_valid,
 9      output             o_a_ready,
10      output             o_b_ready,
11      input              i_res_ready,
12      output [7:0]       o_res_data,
13      output             o_res_valid,
14      output reg         o_res_last
15      );
16
17  localparam READ_A    = 1;
18  localparam READ_B    = 2;
19  localparam SHIFT_ADD = 3;
20  localparam OUTPUT    = 4;
21
22  reg signed [7:0] col_a [0:2];
23  reg signed [7:0] row_b [0:2];
24
25  reg [1:0] a_count;
26  reg [1:0] b_count;
27
28  reg [2:0] current_state;
29  reg [1:0] b_idx_store;
30  reg [2:0] iterations;
31
32  reg [3:0] output_idx;
33
34  reg signed [15:0] int_res [0:2];
35  reg signed [7:0]  mat_res [0:8];
36
37  assign o_a_ready   = current_state == READ_A;
38  assign o_b_ready   = current_state == READ_B;
39
40  assign o_res_valid = current_state == OUTPUT;
41  assign o_res_data  = mat_res[output_idx];
42
43  integer i;
44
45  initial
46  begin
47      a_count      <= 0;
48      b_count      <= 0;
49      current_state <= READ_A;
50      iterations   <= 0;
```

```
51      output_idx   <= 0;
52      o_res_last   <= 0;
53      for(i = 0; i < 9; i = i+1) begin
54          mat_res[i] = 8'h00;
55      end
56  end
57
58  always @(posedge i_clk, negedge i_rst_n)
59  begin
60      if (!i_rst_n) begin
61          a_count      <= 0;
62          b_count      <= 0;
63          current_state <= READ_A;
64          iterations   <= 0;
65          output_idx   <= 0;
66          for(i = 0; i < 9; i = i+1) begin
67              mat_res[i] = 8'h00;
68          end
69      end
70  end
71
72  always @(posedge i_clk, negedge i_rst_n)
73  begin
74      if (i_clk_e) begin
75          if ((current_state == READ_B || current_state == SHIFT_ADD)
&& b_count > 0) begin
76              int_res[0]  <= col_a[0] * row_b[b_count - 1];
77              int_res[1]  <= col_a[1] * row_b[b_count - 1];
78              int_res[2]  <= col_a[2] * row_b[b_count - 1];
79              b_idx_store <= b_count - 1;
80          end
81      end
82  end
83
84  always @(negedge i_clk)
85  begin
86      if (i_clk_e) begin
87          if ((current_state == READ_B || current_state == SHIFT_ADD)
&& b_count > 1) begin
88              mat_res[0+b_idx_store] <= mat_res[0+b_idx_store] +
int_res[0][11:4];
89              mat_res[3+b_idx_store] <= mat_res[3+b_idx_store] +
int_res[1][11:4];
90              mat_res[6+b_idx_store] <= mat_res[6+b_idx_store] +
int_res[2][11:4];
91          end
92      end
```

```
93  end
94
95  always @(posedge i_clk)
96  begin
97      if (i_clk_e) begin
98          case (current_state)
99              READ_A: begin
100                 if (a_count == 3) begin
101                     current_state <= READ_B;
102                 end else if (i_a_num_valid) begin
103                     col_a[a_count] <= i_a_num;
104                     a_count        <= a_count + 1;
105                 end
106             end
107
108             READ_B: begin
109                 if (b_count == 3) begin
110                     current_state <= SHIFT_ADD;
111                 end else if (i_b_num_valid) begin
112                     row_b[b_count] <= i_b_num;
113                     b_count        <= b_count + 1;
114                 end
115             end
116
117             SHIFT_ADD: begin
118                 a_count    <= 0;
119                 b_count    <= 0;
120                 iterations <= iterations + 1;
121                 if (iterations == 3) begin
122                     current_state <= OUTPUT;
123                 end else begin
124                     current_state <= READ_A;
125                 end
126             end
127
128             OUTPUT: begin
129                 if (output_idx == 8) begin
130                     output_idx    <= 0;
131                     current_state <= READ_A;
132                     a_count       <= 0;
133                     b_count       <= 0;
134                     o_res_last    <= 1;
135                 end else if (i_res_ready) begin
136                     output_idx <= output_idx + 1;
137                 end
138             end
139         endcase
```

```
140     end
141 end
142
143 endmodule
```

**8.2 Documentation**

- This module is a substitute for the partial multiply-accumulate pipeline covered by modules 4 and 6. This was an alternate implementation that was supposed to run much faster, but we ended up abandoning this approach because it was very difficult to explain and required that all the matrix elements fit on one EPM240 FPGA, which was the FPGA that was in use initially, and this was not possible. Still, we have documented it for the purposes of completeness.
- Lines 1-14: Module inputs and outputs - features the standard set of AXIStream and Data inputs and outputs.
- State enumeration in 17-20: There are 4 states in this FSM:
    - **`READ_A`**: 1 (read a column of A)
    - **`READ_B`**: 2 (read a column of B)
    - SHIFT_ADD: buffer state to allow one more clock cycle for the full accumulation that happens in lines 84-91, essentially takes the reset logic for **`a_count, b_count,`** and the increment logic for **`iterations`**, and delays it by one clock cycle.
    - OUTPUT: increment **`output_idx, mat_res[output_idx]`** is wired to **`o_res_data.`**
- 22-35 feature important declarations of counters and intermediate arrays. **`col_a, row_b, a_count, b_count, current_state, output_idx,`** and **`iterations`** serve the same purposes as earlier, but we introduce **`b_idx_store,`** a new variable that stores the index of the element in **`row_b`** with which the outer product was computed in the last clock cycle.
- 37-43: Standard wire assignment logic as above.
- This source file synthesizes multiple **`always @(posedge_clk, nededge i_rst_n)`** blocks. They are best interpreted step-by-step.
    - First on line 95 onwards, we enter **`READ_A`** and solely continue here for 3 clock cycles.
    - In **`READ_B,`** we enter our first element, increment **`b_count`**, and on the next clock cycle as we enter **`row_b[1]`**, the outer product computation begins in the block on line 72 onwards simultaneously, computing **`int_res[x] = col_a[x] * row_b[0]`**, (**`row_b[0]`** is already available and idle) and **`b_idx_store`** stores **`0`** to remember which element was used in the outer product.
    - The block on line 84 is not triggered for this **`HIGH`** of the clock. On the next high, **`row_b[2]`** is entered, **`int_res[x] = col_a[x] * row_b[1],`** and on the negative edge of this clock, the first column of the outer product given by **`mat_res[0, 3, 6]`** is populated using the contents of **`int_res`** from the previous clock cycle. Note that the update for

**int_res** from **row_b[1]** is non-blocking, so while lines 88-90 accumulate, **int_res** still retains information from the previous clock cycle.

- In this parallelized fashion, we are able to multiply and accumulate results as **row_b** is populated. This is much faster as a result.

---