REPORT

ON

Matrix Multiplication on Zed-Board and Display on Monitor Using VGA

BY

ADARSH PRAVEEN	**2023A3PS0210G**
PARANJAY DHADWAL	**2023A3PS0500G**
GIRIDHAR NAGAMANGALA	**2023A8PS0522G**

AT

BIRLA INSTITUTE OF TECHNOLOGY SCIENCE, GOA

(November 2025)

BITS Pilani
K K Birla Goa Campus

Topic: Matrix Multiplication on Zed-Board and Display on Monitor Using VGA

Objective: The objective of this project is to implement the multiplication of two M × N matrices on the Xilinx ZedBoard FPGA and visually display both the input matrices and the computed output matrix on a VGA monitor. An additional goal is to explore different fixed-point quantization formats (such as 16-bit and 8-bit) to analyze their effects on numerical precision, performance, and hardware resource utilization.

Components Used :

● **Xilinx Zed-Board FPGA:** The Xilinx ZedBoard [ZedBoard Zynq Evaluation and Development Kit (xc7z020clg484-1)] is the central hardware platform used to implement the matrix-multiplication system. In this project, the programmable logic is used to build the systolic array, Cannon's algorithm control logic, multiply–accumulate units, data-shifting network, fixed-point computation modules, and the VGA display interface.

● **VGA Monitor:** The VGA monitor provides real-time visual output of the input matrices and the computed results. Using the ZedBoard's VGA interface, the system displays matrix values in a clear 2D grid.

Design Approach

System Architecture Overview

The design implements a matrix multiplication visualization system that takes two 3×3 matrices as inputs, performs multiplication using Cannon's algorithm, and displays all three matrices on a VGA monitor. The system uses a modular architecture separating computation, conversion, and display functions. The design is fully parameterized to support different matrix dimensions and data formats.

Input Interface: VIO (Virtual Input/Output)

The system uses Xilinx's VIO core for input during development and testing. Two 144-bit wide probe outputs provide the flattened representations of matrix A and matrix B, with each matrix element represented as a 16-bit value in Q8.8 fixed-point format (8 bits integer, 8 bits fractional). The VIO also provides a reset signal and monitors output signals including compute_done and VGA timing signals for debugging.

Computational Core: Cannon's Algorithm Implementation

The cannon module implements Cannon's algorithm as a systolic array with dedicated MAC (Multiply-Accumulate) units at each output matrix position. The module is parameterized with N (matrix dimension) and WIDTH (element bit width).

Initial Skewing and MAC Operation

Input matrices undergo initial skewing: matrix A rows are circularly left-shifted by row index i, and matrix B columns are upward-shifted by column index j. This ensures correct element pairs arrive at each MAC unit simultaneously. Each MAC unit contains three registers: a_reg and b_reg for operands, and a 34-bit c_reg accumulator (WIDTH*2 + ceil(log2(N)) bits to prevent overflow). The systolic array implements a toroidal mesh with horizontal data flow for matrix A and vertical flow for matrix B.

State Machine Control

A four-state FSM (IDLE, LOAD_INIT, COMPUTE, OUTPUT) orchestrates computation. LOAD_INIT loads pre-skewed data and clears accumulators in one cycle. COMPUTE runs for N+1 cycles, with active computation during cycles 0 to N-1 and a

final stabilization cycle. The OUTPUT state provides results through direct wire assignments to stage0-stage8 outputs, making all results immediately accessible for display.

Fixed-Point to Decimal Conversion

The fixed_to_decimal module converts fixed-point computation results to decimal digits for display. It's parameterized with WIDTH and FRAC_BITS.

The conversion extracts the integer part through right-shifting, isolates the fractional part with bitwise AND masking, then converts the fraction to decimal by multiplying by 10,000 and dividing by 2^FRAC_BITS. Both parts are decomposed into BCD digits using division and modulo operations. The output is a 24-bit concatenated value: two fractional digits, decimal point indicator (4'b1100), and three integer digits.

Display Subsystem: VGA Controller and Character ROM

The display subsystem generates VGA 640×480@60Hz video output using horizontal and vertical counter modules for timing signals. A clock divider produces the 25MHz pixel clock from the system clock.

Character Display and Layout

Each matrix element displays as a six-character number (XX.XX) using the digit_8x16 module, which contains a BRAM-based ROM storing 8×16 pixel bitmaps for digits 0-9 and decimal point. The module calculates whether current pixel coordinates fall within its character boundary and fetches the appropriate ROM pixel value.

All digit and bracket modules output pixon signals that are aggregated using reduction OR operations. When any signal is active, RGB channels are set to maximum (4'hF), producing white pixels on black background for maximum contrast.

Working Principle :

Cannon's Algorithm is a well-known parallel algorithm for performing matrix multiplication efficiently on a two-dimensional processor grid. It is especially suited for hardware architectures such as systolic arrays, where computation and data movement can occur simultaneously in a regular pattern. The algorithm greatly reduces inter-processor communication by carefully "skewing" the input matrices before computation and then performing cyclic data shifts during the multiplication process. This results in very high throughput and predictable timing, making Cannon's algorithm a natural fit for hardware accelerators and FPGA/ASIC implementations.

The algorithm executes in three distinct phases: **Initial Alignment**, **Iterative Computation**, and **Result Extraction**.

Phase A: Initial Skewing (Data Alignment)

Before computation begins, the matrices must be aligned so that the correct pairs (A_i,k, B_k,j) meet at the correct Processing Element (PE) at time t=0..

- **Matrix A (Rows):** Each row i is cyclically shifted **left** by i positions.
    - Row 0: Shift 0.
    - Row 1: Shift 1.
    - ...
    - Row N-1: Shift N-1.
- **Matrix B (Columns):** Each column j is cyclically shifted **up** by j positions.
    - Col 0: Shift 0.
    - Col 1: Shift 1.
    - ...
    - Col N-1: Shift N-1.

This skewing ensures that PE (i,j) initially holds the valid starting operands for the dot product sequence.

Phase B: The Systolic Cycle (Compute & Shift)

The algorithm proceeds through N synchronized clock cycles. In every cycle, all PEs perform the following operations in parallel:

1. Multiply-Accumulate (MAC):
   Each PE multiplies its currently held value of A and B and adds the product to its local accumulator (register C).
2. Cyclic Shift (Data Movement):
   After the multiplication, data is passed to neighbors to prepare for the next cycle:
   a. **Matrix A:** Shifted **Left** by 1 (wrapping around toroidally).
   b. **Matrix B:** Shifted **Up** by 1 (wrapping around toroidally).

Because the initial skewing aligned the sequence, this simple "shift-one" logic guarantees that over N cycles, every PE receives exactly the row of A and column of B required to complete its dot product.

Phase C: Result Validity

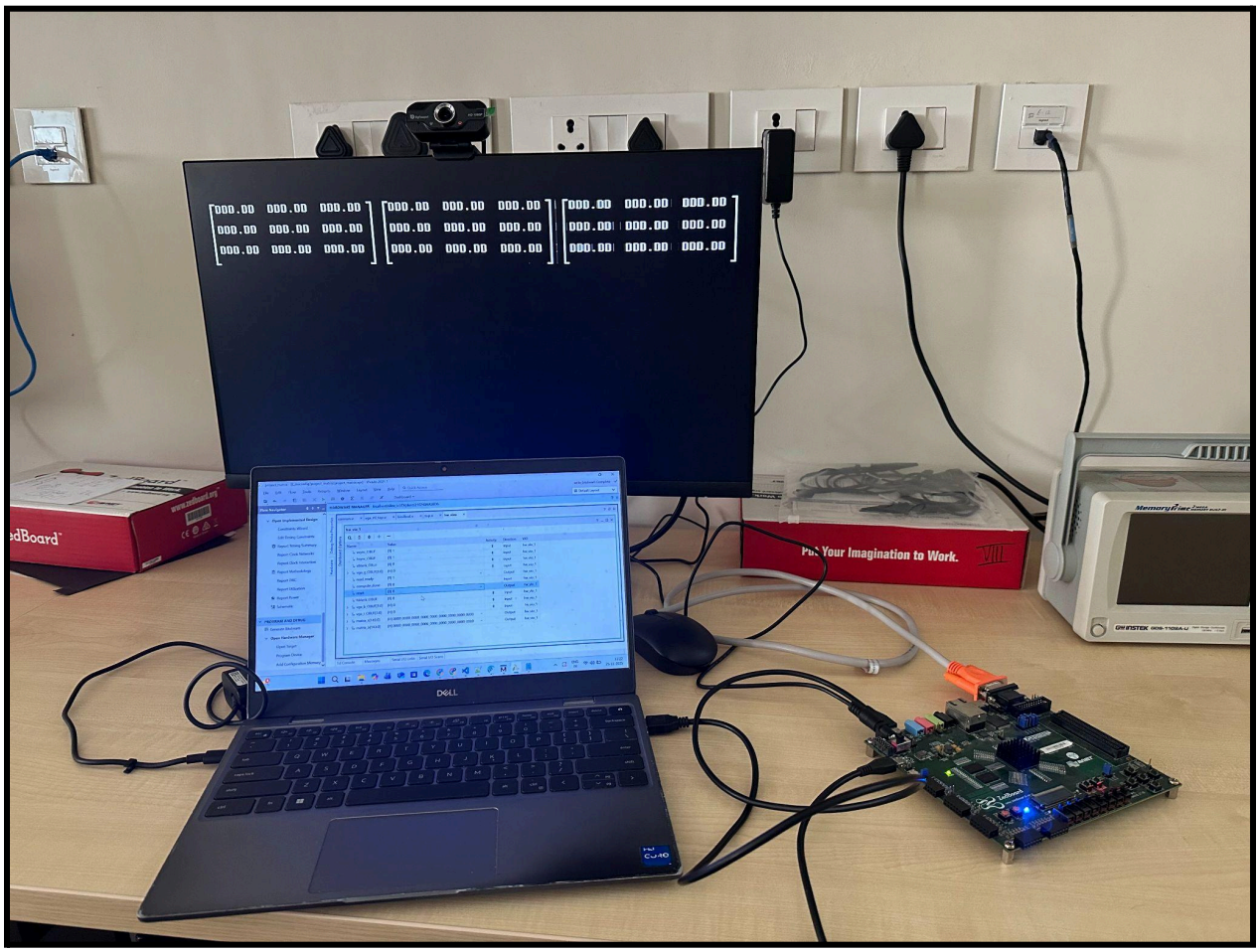
After exactly N cycles, every PE (i,j) has accumulated the complete sum. For a 3x3 matrix the accumulated final sum looks like::

$$
\begin{aligned}
P_{00} &= A_{00}B_{00} + A_{01}B_{10} + A_{02}B_{20},\\
P_{01} &= A_{00}B_{01} + A_{01}B_{11} + A_{02}B_{21},\\
P_{02} &= A_{00}B_{02} + A_{01}B_{12} + A_{02}B_{22}.
\end{aligned}
$$

$$
\begin{aligned}
P_{10} &= A_{10}B_{00} + A_{11}B_{10} + A_{12}B_{20},\\
P_{11} &= A_{10}B_{01} + A_{11}B_{11} + A_{12}B_{21},\\
P_{12} &= A_{10}B_{02} + A_{11}B_{12} + A_{12}B_{22}.
\end{aligned}
$$

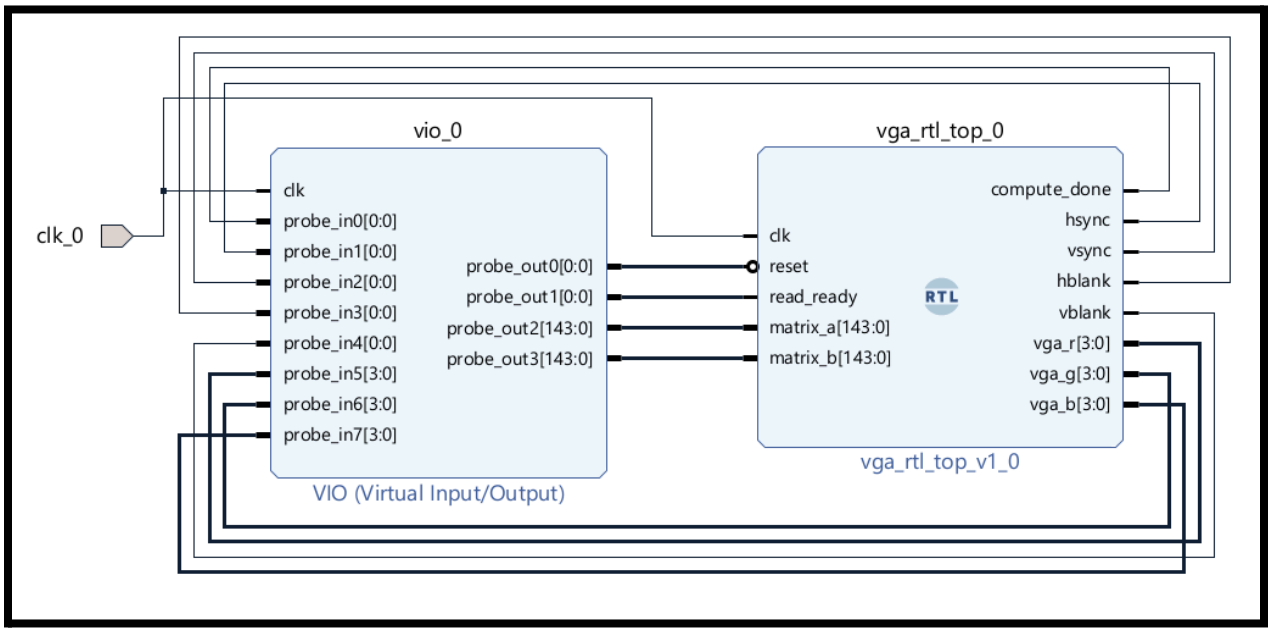
$$
\begin{aligned}
P_{20} &= A_{20}B_{00} + A_{21}B_{10} + A_{22}B_{20},\\
P_{21} &= A_{20}B_{01} + A_{21}B_{11} + A_{22}B_{21},\\
P_{22} &= A_{20}B_{02} + A_{21}B_{12} + A_{22}B_{22}.
\end{aligned}
$$

Working Setup:

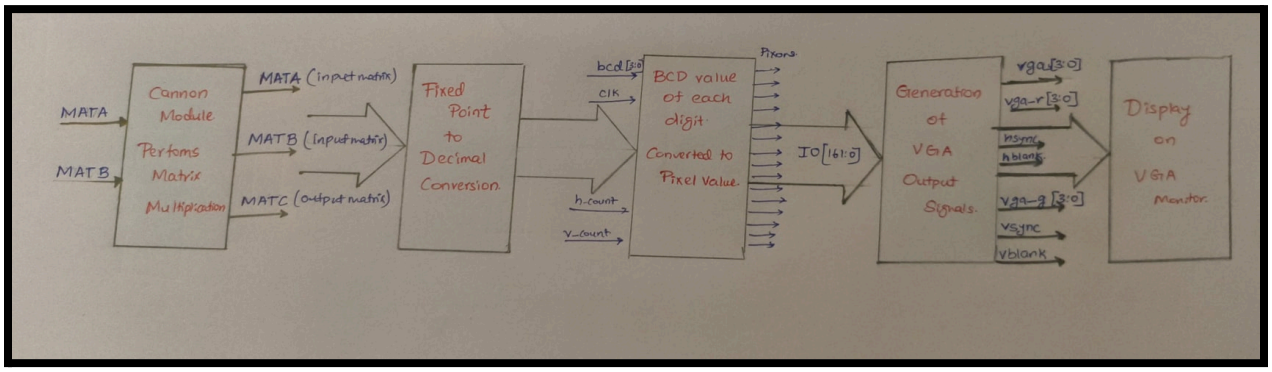


The VIO running on the laptop is used to provide the input values for matrices A and B, which are then displayed on the VGA monitor through the FPGA interface.

Block Design:

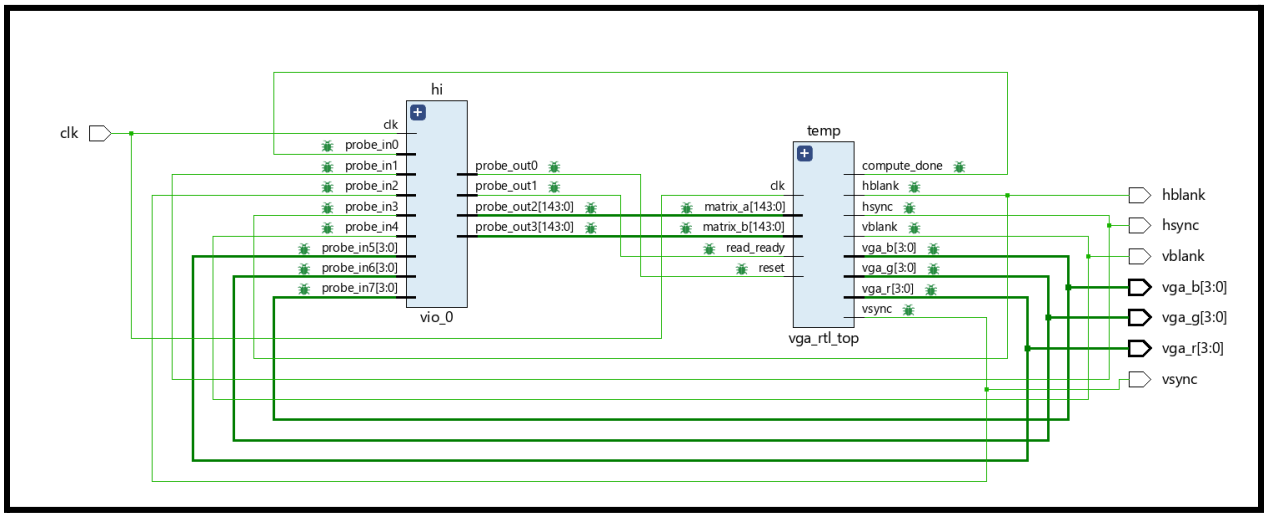


BLOCK DIAGRAM:

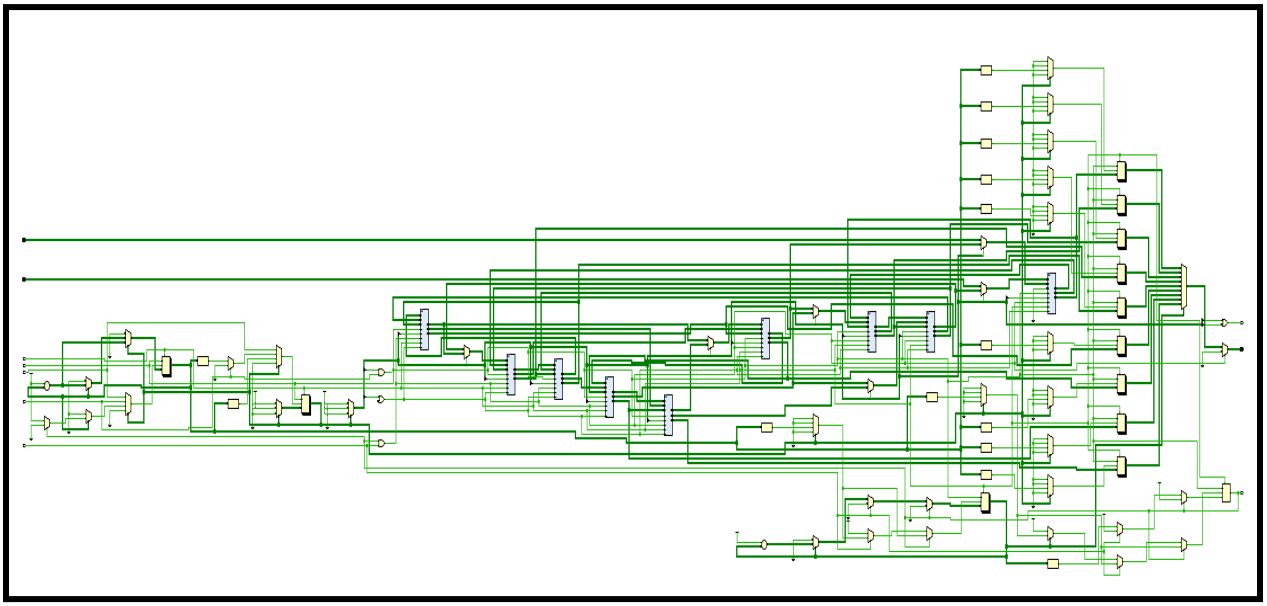


VIO is used to provide input matrices A and B.

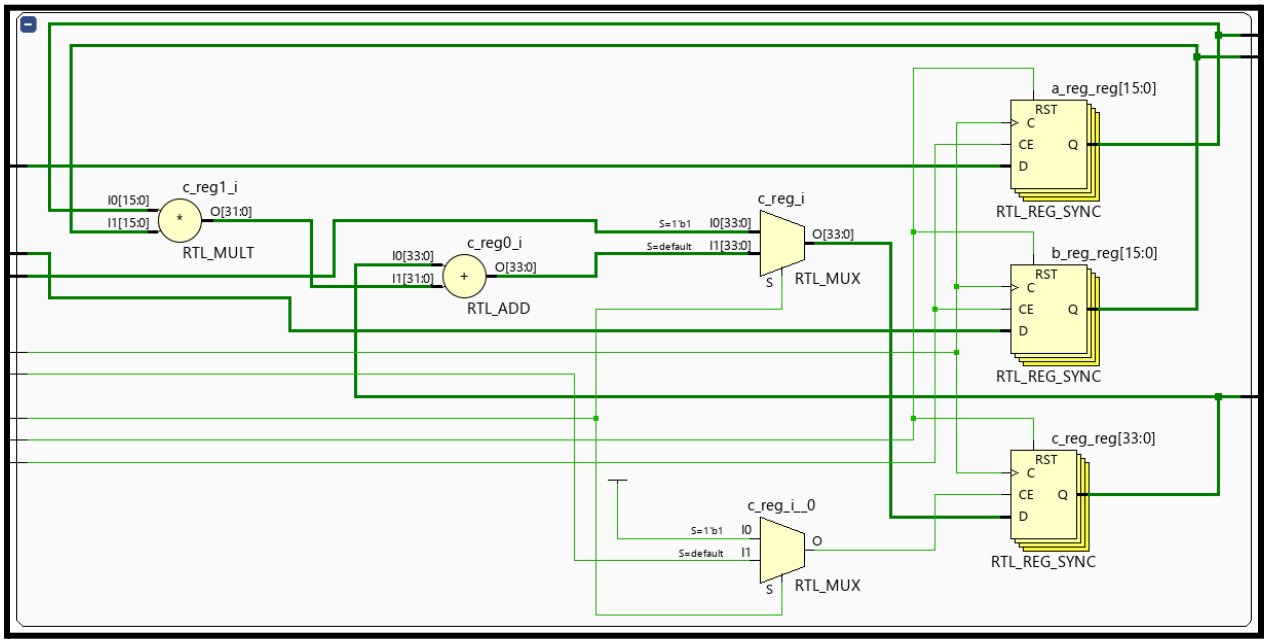
RTL Schematic:



RTL Schematic of the top module and the VIO showing input output connections.

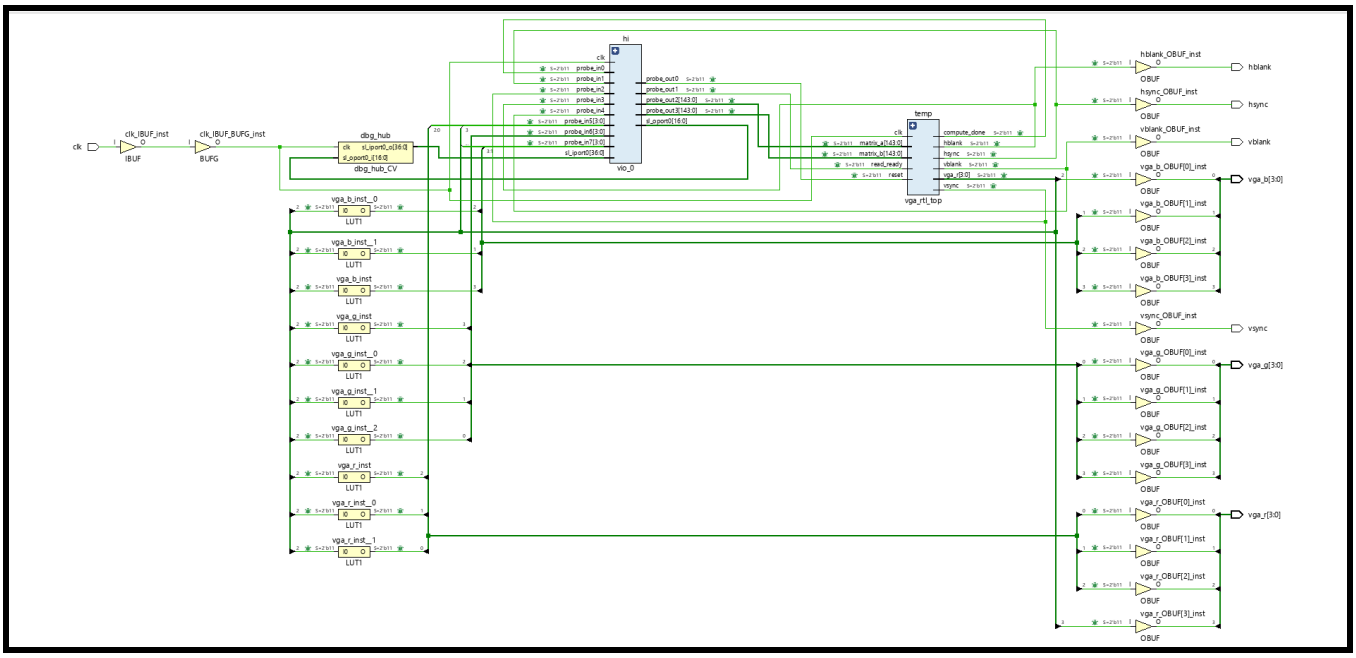


RTL Schematic of the Canons Algorithm, showing MAC units for a 3*3 matrix.



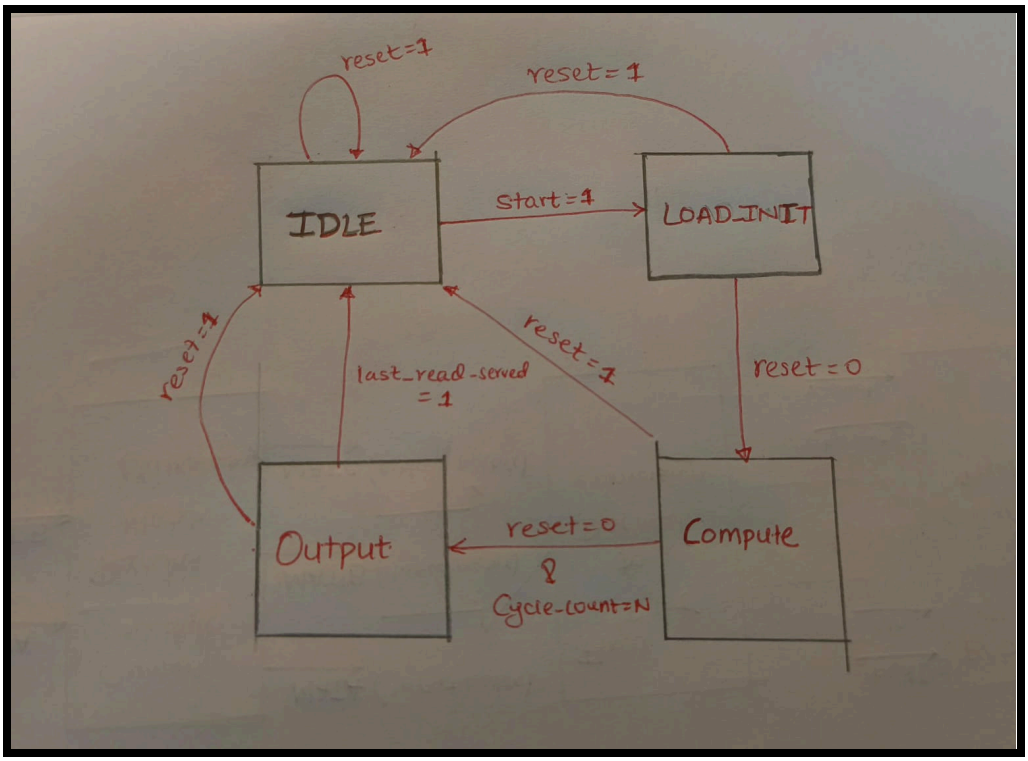
RTL Schematic of each MAC unit.

Synthesis Schematic:



Synthesis Schematic of the top module showing interconnections.

STATE DIAGRAM (For Cannon Module) :



Timing Report:

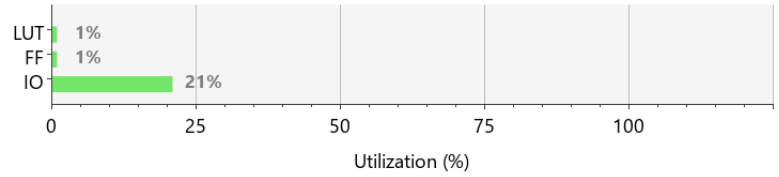
Setup		Hold		Pulse Width	
Worst Negative Slack (WNS):	23.134 ns	Worst Hold Slack (WHS):	0.096 ns	Worst Pulse Width Slack (WPWS):	15.250 ns
Total Negative Slack (TNS):	0.000 ns	Total Hold Slack (THS):	0.000 ns	Total Pulse Width Negative Slack (TPWS):	0.000 ns
Number of Failing Endpoints:	0	Number of Failing Endpoints:	0	Number of Failing Endpoints:	0
Total Number of Endpoints:	1036	Total Number of Endpoints:	1028	Total Number of Endpoints:	483

All user specified timing constraints are met.

Timing Report for the Top module.

Utilization Report:

Resource	Utilization	Available	Utilization %
LUT	18	53200	0.03
FF	14	106400	0.01
IO	41	200	20.50

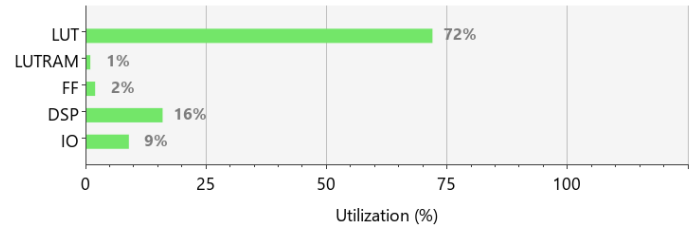


Utilization Report for the parametrized Matrix Multiplication Module.

Name	Slice LUTs (53200)	Slice Registers (106400)	Slice (13300)	LUT as Logic (53200)	Bonded IOB (200)	BUFGCTRL (32)
N cannon	18	14	6	18	41	1

Cannon module resource utilization, including slice count, flip-flop usage, etc.

Resource	Utilization	Available	Utilization %
LUT	38422	53200	72.22
LUTRAM	24	17400	0.14
FF	2437	106400	2.29
DSP	36	220	16.36
IO	17	200	8.50



Utilization Report for the Top Module.

Name	Slice LUTs (53200)	Slice Registers (106400)	F7 Muxes (26600)	F8 Muxes (13300)	Slice (13300)	LUT as Logic (53200)	LUT as Memory (17400)	DSPs (220)	Bonded IOB (200)	BUFGCTRL (32)	BSCANE2 (4)
top	38422	2437	949	309	11530	38398	24	36	17	2	1
dbg_hub (dbg_hub)	449	741	0	0	241	425	24	0	0	1	1
hi (vio_0)	312	983	0	0	417	312	0	0	0	0	0
temp (vga_rtl_top)	37657	713	949	309	11297	37657	0	36	0	0	0
cannon_inst (canno	2453	593	59	17	1064	2453	0	9	0	0	0
clk_div_inst (clk_div	2	3	0	0	1	2	0	0	0	0	0
horiz_counter_inst (	422	25	10	1	275	422	0	0	0	0	0
row_loop_A[0].col_	703	0	0	0	245	703	0	1	0	0	0
row_loop_A[0].col_	703	0	0	0	251	703	0	1	0	0	0
row_loop_A[0].col_	705	0	0	0	252	705	0	1	0	0	0
row_loop_A[1].col_	699	0	0	0	247	699	0	1	0	0	0

Utilization of the different functional modules within the top module.

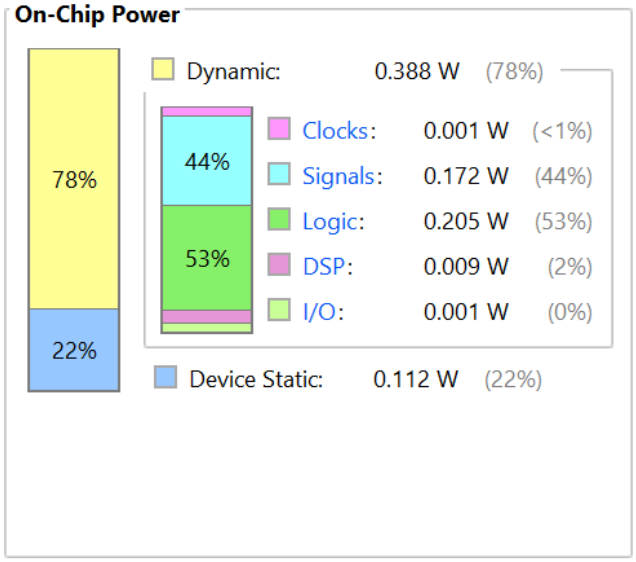
On Chip Power Report:

Summary

Power analysis from Implemented netlist. Activity derived from constraints files, simulation files or vectorless analysis.

Total On-Chip Power:	**0.499 W**
Design Power Budget:	**Not Specified**
Power Budget Margin:	**N/A**
Junction Temperature:	**30.8°C**
Thermal Margin:	54.2°C (4.5 W)
Effective ϑJA:	11.5°C/W
Power supplied to off-chip devices:	0 W
Confidence level:	Low

Launch Power Constraint Advisor to find and fix invalid switching activity



Results and Conclusion:

The implementation successfully integrates the systolic matrix multiplication core with a real-time VGA visualization interface. The system operates as a cohesive pipeline where data transitions smoothly from raw binary inputs to a human-readable decimal display on the monitor.

Matrix Multiplication Execution

Upon asserting the system reset, the 3x3 input matrices are latched from the VIO interface into the computational core. The data flow within the CANNON module follows a precise, rhythmic pattern. During the initialization phase, the input matrices undergo spatial skewing—rows of Matrix A shift left and columns of Matrix B shift up—aligning the first set of operands at their respective Multiply-Accumulate (MAC) units.

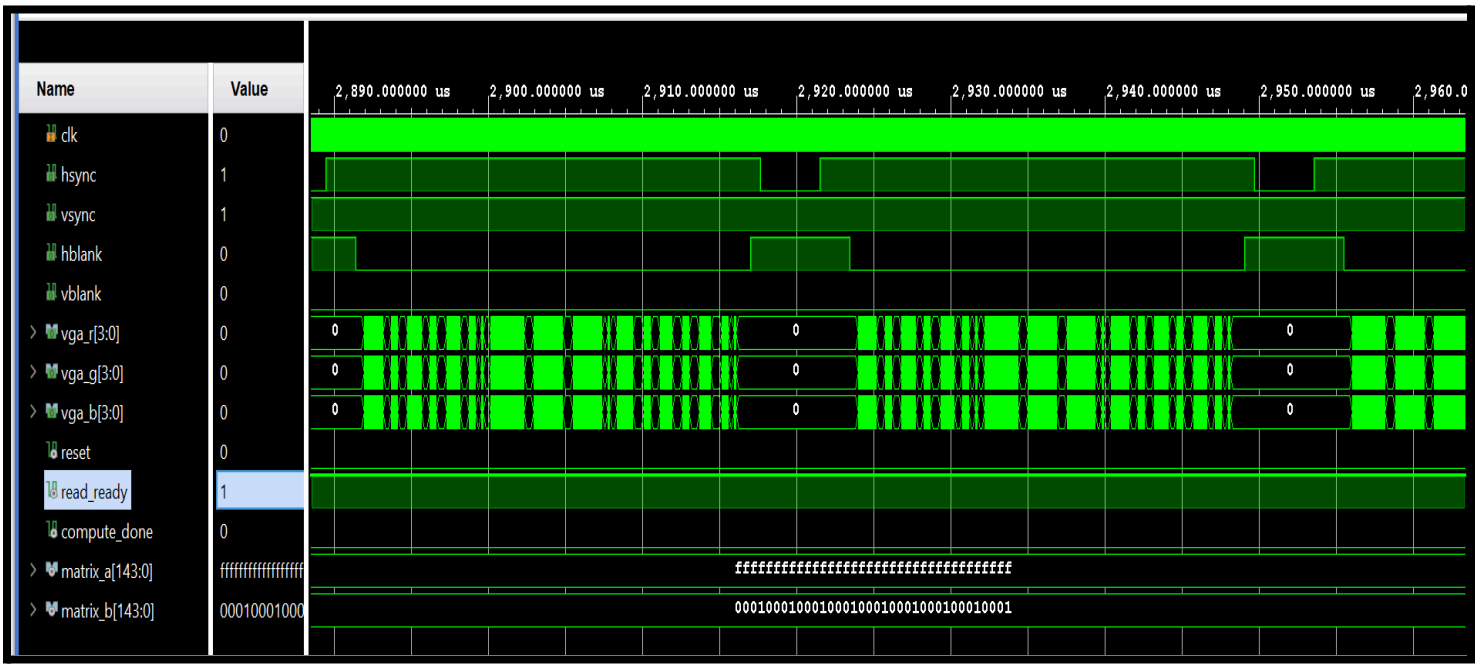
As the Finite State Machine transitions to the COMPUTE state, the systolic array activates. Over three synchronized clock cycles, data propagates through the toroidal mesh: calculating partial products, shifting to adjacent Processing Elements, and accumulating totals. By the end of the execution window, the final dot-product values are stabilized and latched into the output registers, confirming the correct execution of the shift-and-accumulate logic.

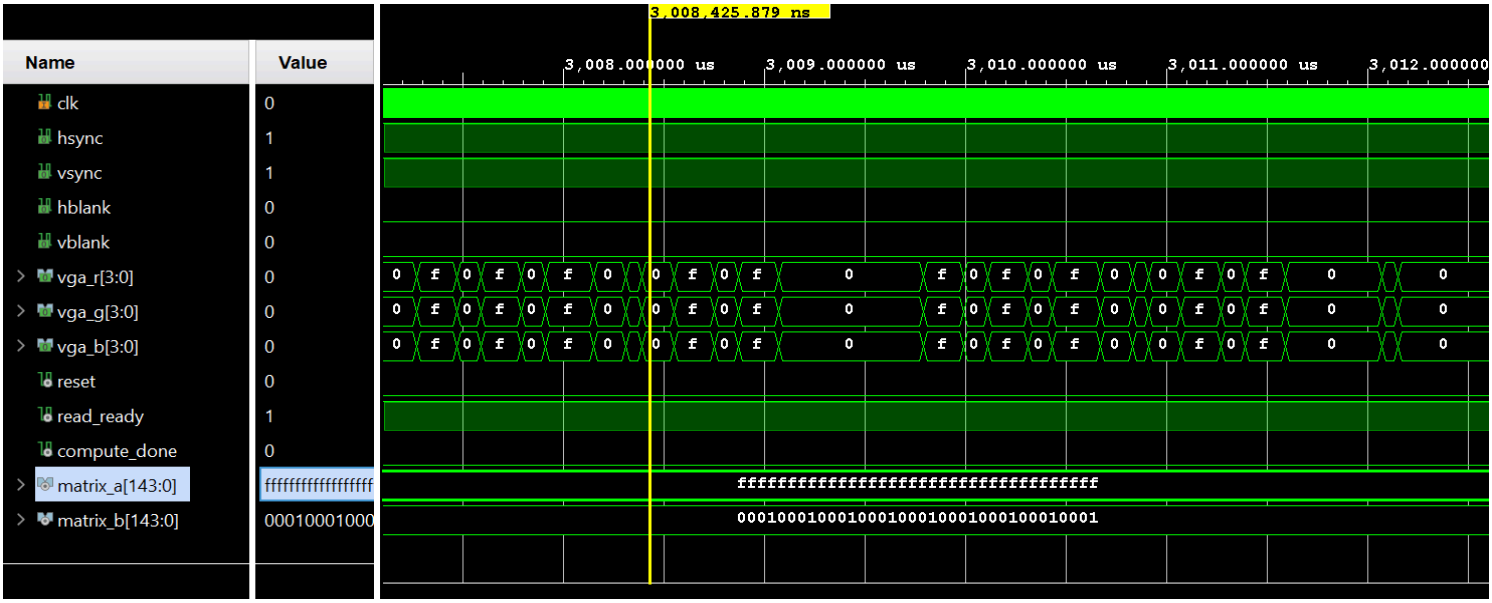
VGA Controller and Visual Output

Once computation is complete, the data flow shifts from parallel processing to serial pixel generation. The raw 34-bit fixed-point results are first intercepted by conversion modules that decompose the binary values into Binary Coded Decimal (BCD) format, separating integer and fractional parts.

Simultaneously, the VGA controller generates the horizontal and vertical synchronization signals required to drive the 640×480 screen resolution. As the monitor is scanned pixel by pixel, logic modules compare the current scan coordinates against the defined boundaries for each matrix element. When the scan beam crosses a valid character region, the system fetches the appropriate bitmap from the Character ROM and drives the RGB channels. This renders the matrices as sharp white text against a black background, effectively translating the internal register values into a real-time visual output.

Simulation Results:





Conclusion:

The project successfully demonstrates the hardware implementation of Cannon's Algorithm on the Xilinx Zed-Board (). The visual output on the VGA monitor matches the theoretical results, verifying the accuracy of the systolic array design. By effectively decoupling high-speed computation from the display logic, the system validates the capability of FPGAs to handle complex, synchronized data flows in embedded applications.

This final design was built after many rounds of synthesis and hardware debugging. In the beginning, we focused on getting the horizontal and vertical counters perfectly synchronized so the display would not show any glitches. For the matrix multiplication block, we tried both serial and parallel output methods to see which one transferred data to the display controller more smoothly. We also tested different matrix sizes and screen layouts to make the output easier to read.

Overall, combining all these parts into one working system taught us two important things: keep the Verilog design modular, and always manage timing constraints carefully when connecting computation logic to a real-time display.

Applications:

- **Real-Time Matrix Multiplication in Embedded Systems**
  The design can be used for fast matrix operations in embedded applications such as control systems, signal processing, robotics, and sensor fusion, where low-latency computation is essential.

- **Hardware Acceleration for Machine Learning**
  Since matrix multiplication is the core of neural network operations, the systolic design can serve as a hardware accelerator for small neural network layers or edge-AI applications.

- **Digital Signal Processing (DSP)**
  Many DSP algorithms rely on frequent matrix operations (e.g., filtering, transforms, and correlation). FPGA implementation can act as a dedicated matrix engine to speed up these computations in communication or audio systems.

- **Custom Hardware Co-Processors**
  The design can be integrated as a specialized co-processor in larger FPGA-based systems to offload repeated matrix calculations, improving system throughput and freeing the main processor for other tasks.

References:

- https://www.avnet.com/americas/products/avnet-boards/avnet-board-families/zedboard/zedboard-board-family/
- https://www.comrevo.com/2020/11/cannons-algorithm-for-matrix-multiplication.html
- https://www3.nd.edu/~zxu2/acms60212-40212-S12/Lec-07-3.pdf
- https://www.youtube.com/watch?v=_o4FbVFLbuw&list=PLXHMvqUANAFOviU0J8HSp0E91lLJInzX1&index=38
- https://www.youtube.com/watch?v=BcJU234Jh3k
- https://stackoverflow.com/questions/18576409/how-to-interface-a-vga-monitor-to-fpga-using-verilog
- https://www.realdigital.org/doc/fec48fabee69c0be03c8b6228afde63c

Verilog Codes:

```
module cannon #(
    parameter N = 3,          // Matrix size N x N
    parameter WIDTH = 16      // Data width of A and B elements
) (
    input clk,
    input reset,
    input start,
    input [N*N*WIDTH - 1:0] mat_a_in,
    input [N*N*WIDTH - 1:0] mat_b_in,

    output [WIDTH-1:0] mata1,
    output [WIDTH-1:0] mata2,
    output [WIDTH-1:0] mata3,
    output [WIDTH-1:0] mata4,
    output [WIDTH-1:0] mata5,
    output [WIDTH-1:0] mata6,
    output [WIDTH-1:0] mata7,
    output [WIDTH-1:0] mata8,
    output [WIDTH-1:0] mata9,

    output [WIDTH-1:0] matb1,
    output [WIDTH-1:0] matb2,
    output [WIDTH-1:0] matb3,
    output [WIDTH-1:0] matb4,
    output [WIDTH-1:0] matb5,
    output [WIDTH-1:0] matb6,
    output [WIDTH-1:0] matb7,
    output [WIDTH-1:0] matb8,
    output [WIDTH-1:0] matb9,

    input read_ready,
    output [WIDTH*2-1+$clog2(N):0] serial_c_out,
    output output_valid,
     output [WIDTH*2-1+$clog2(N):0] stage0,
     output [WIDTH*2-1+$clog2(N):0] stage1,
```

```
    output [WIDTH*2-1+$clog2(N):0] stage2,
    output [WIDTH*2-1+$clog2(N):0] stage3,
    output [WIDTH*2-1+$clog2(N):0] stage4,
    output [WIDTH*2-1+$clog2(N):0] stage5,
    output [WIDTH*2-1+$clog2(N):0] stage6,
    output [WIDTH*2-1+$clog2(N):0] stage7,
    output [WIDTH*2-1+$clog2(N):0] stage8
);

      assign mata1 =(reset)? 16'b0000000000000000 : mat_a_in[N*N*WIDTH - 1          :
((N*N)-1)*WIDTH];
       assign mata2 =(reset)? 16'b0000000000000000 : mat_a_in[((N*N)-1)*WIDTH - 1 :
((N*N)-2)*WIDTH];
       assign mata3 =(reset)? 16'b0000000000000000 : mat_a_in[((N*N)-2)*WIDTH - 1 :
((N*N)-3)*WIDTH];
       assign mata4 =(reset)? 16'b0000000000000000 : mat_a_in[((N*N)-3)*WIDTH - 1 :
((N*N)-4)*WIDTH];
       assign mata5 =(reset)? 16'b0000000000000000 : mat_a_in[((N*N)-4)*WIDTH - 1 :
((N*N)-5)*WIDTH];
       assign mata6 =(reset)? 16'b0000000000000000 : mat_a_in[((N*N)-5)*WIDTH - 1 :
((N*N)-6)*WIDTH];
       assign mata7 =(reset)? 16'b0000000000000000 : mat_a_in[((N*N)-6)*WIDTH - 1 :
((N*N)-7)*WIDTH];
       assign mata8 =(reset)? 16'b0000000000000000 : mat_a_in[((N*N)-7)*WIDTH - 1 :
((N*N)-8)*WIDTH];
       assign mata9 =(reset)? 16'b0000000000000000 : mat_a_in[((N*N)-8)*WIDTH - 1 :
((N*N)-9)*WIDTH];

      assign matb1 =(reset)? 16'b0000000000000000 : mat_b_in[N*N*WIDTH - 1          :
((N*N)-1)*WIDTH];
       assign matb2 =(reset)? 16'b0000000000000000 : mat_b_in[((N*N)-1)*WIDTH - 1 :
((N*N)-2)*WIDTH];
       assign matb3 =(reset)? 16'b0000000000000000 : mat_b_in[((N*N)-2)*WIDTH - 1 :
((N*N)-3)*WIDTH];
       assign matb4 =(reset)? 16'b0000000000000000 : mat_b_in[((N*N)-3)*WIDTH - 1 :
((N*N)-4)*WIDTH];
       assign matb5 =(reset)? 16'b0000000000000000 : mat_b_in[((N*N)-4)*WIDTH - 1 :
((N*N)-5)*WIDTH];
       assign matb6 =(reset)? 16'b0000000000000000 : mat_b_in[((N*N)-5)*WIDTH - 1 :
((N*N)-6)*WIDTH];
```

```
        assign matb7 =(reset)? 16'b0000000000000000 : mat_b_in[((N*N)-6)*WIDTH - 1 : ((N*N)-7)*WIDTH];
        assign matb8 =(reset)? 16'b0000000000000000 : mat_b_in[((N*N)-7)*WIDTH - 1 : ((N*N)-8)*WIDTH];
        assign matb9 =(reset)? 16'b0000000000000000 : mat_b_in[((N*N)-8)*WIDTH - 1 : ((N*N)-9)*WIDTH];



    localparam C_WIDTH = WIDTH*2 + $clog2(N);
    localparam NUM_CELLS = N*N;
    localparam STATE_BITS = 2;

    localparam IDLE=2'd0, LOAD_INIT=2'd1, COMPUTE=2'd2, OUTPUT=2'd3;

    reg [STATE_BITS-1:0] state;
    reg [$clog2(N):0] cycle_count;
    reg [$clog2(NUM_CELLS):0] out_count;

    wire [WIDTH-1:0] a_chain [0:NUM_CELLS-1];
    wire [WIDTH-1:0] b_chain [0:NUM_CELLS-1];
    wire [C_WIDTH-1:0] c_result_array [0:NUM_CELLS-1];

    reg [C_WIDTH-1:0] output_regs [0:NUM_CELLS-1];
    integer idx;

    // --- Timing Fix: Gated Enables ---
    // Computation and Shifting should only happen for cycles 0 to N-1.
    // Cycle N is reserved for capturing the final result.


    shift_register_34bit final(     clk,
     reset,
     output_valid,
     serial_c_out,
     stage0,
    stage1,
    stage2,
     stage3,
   stage4,
```

```
  stage5,
 stage6,
 stage7,
stage8);
wire active_compute_cycle = (cycle_count < N);

wire load_init_en = (state == LOAD_INIT);
wire compute_en = (state == COMPUTE) && active_compute_cycle;
wire shift_en = load_init_en || compute_en;

reg last_read_served;

always @(posedge clk) begin
    if (reset) begin
        state <= IDLE;
        cycle_count <= 0;
        out_count <= 0;
        last_read_served <= 0;
        for (idx = 0; idx < NUM_CELLS; idx = idx + 1) begin
            output_regs[idx] <= 0;
        end
    end else begin
        last_read_served <= 0;

        case (state)
            IDLE: begin
                if (start) begin
                    state <= LOAD_INIT;
                    cycle_count <= 0;
                    out_count <= 0;
                    last_read_served <= 0;
                end
            end

            LOAD_INIT: begin
                state <= COMPUTE;
                cycle_count <= 0;
            end

            COMPUTE: begin
```

```
            // Run for cycles 0 to N-1 to compute.
            // On cycle N, the final MAC results are stable and ready to capture.
            if (cycle_count == N) begin
               // TIMING FIX: Capture results AFTER the Nth computation finishes
               for (idx = 0; idx < NUM_CELLS; idx = idx + 1) begin
                  output_regs[idx] <= c_result_array[idx];
               end
               state <= OUTPUT;
               out_count <= 0;
            end else begin
               cycle_count <= cycle_count + 1;
            end
         end

         OUTPUT: begin
            if (last_read_served) begin
               state <= IDLE;
               out_count <= 0;
            end else if (read_ready) begin
               if (out_count == NUM_CELLS - 1) begin
                  last_read_served <= 1;
               end else begin
                  out_count <= out_count + 1;
               end
            end
         end
         default: state <= IDLE;
      endcase
   end
end

assign output_valid = (state == OUTPUT) && !last_read_served;
assign serial_c_out = output_regs[out_count];



// --- Initial Skewing and MAC Generation (Same as before) ---
wire [WIDTH-1:0] mat_a_elements [0:NUM_CELLS-1];
wire [WIDTH-1:0] mat_b_elements [0:NUM_CELLS-1];
wire [WIDTH-1:0] a_init_skewed [0:NUM_CELLS-1];
```

```
wire [WIDTH-1:0] b_init_skewed [0:NUM_CELLS-1];

genvar g;
generate
   for (g=0; g<NUM_CELLS; g=g+1) begin : input_decompose
      assign mat_a_elements[g] = mat_a_in[g*WIDTH +: WIDTH];
      assign mat_b_elements[g] = mat_b_in[g*WIDTH +: WIDTH];
   end
endgenerate

genvar i_skew, j_skew;
generate
   for (i_skew=0; i_skew<N; i_skew=i_skew+1) begin : row_skew_gen
      for (j_skew=0; j_skew<N; j_skew=j_skew+1) begin : col_skew_gen
         localparam k = i_skew*N + j_skew;
         localparam a_src_j = (j_skew + i_skew) % N;
         localparam a_src_k = i_skew * N + a_src_j;
         assign a_init_skewed[k] = mat_a_elements[a_src_k];

         localparam b_src_i = (i_skew + j_skew) % N;
         localparam b_src_k = b_src_i * N + j_skew;
         assign b_init_skewed[k] = mat_b_elements[b_src_k];
      end
   end
endgenerate

genvar i, j;
generate
   for (i=0; i<N; i=i+1) begin : row_gen
      for (j=0; j<N; j=j+1) begin : col_gen
         localparam k = i*N + j;
         localparam k_prev_a = i*N + (j == 0 ? N-1 : j-1);
         localparam k_prev_b = (i == 0 ? N-1 : i-1)*N + j;

         wire [WIDTH-1:0] a_in_mac = load_init_en ? a_init_skewed[k] : a_chain[k_prev_a];
         wire [WIDTH-1:0] b_in_mac = load_init_en ? b_init_skewed[k] : b_chain[k_prev_b];

         mac #(.WIDTH(WIDTH), .SIZE(N)) u_mac (
            .clk(clk),
            .reset(reset),
```

```
                .shift_en(shift_en),
                .compute_en(compute_en),
                .a_in(a_in_mac),
                .b_in(b_in_mac),
                .a_out(a_chain[k]),
                .b_out(b_chain[k]),
                .c_result(c_result_array[k])
            );
        end
    end
  endgenerate
endmodule



module mac #(
  parameter WIDTH=16,      // Data width of A and B elements
   parameter SIZE=3           // Matrix size N (used for calculating C_result width: 2*WIDTH +
log2(N))
) (
  input clk,
  input reset,
    input shift_en,      // Enables shifting of A and B registers (active during LOAD_INIT and
COMPUTE)
     input compute_en,    // Enables the Multiply-Accumulate operation (active only during
COMPUTE)

  // Inputs
  input [WIDTH-1:0] a_in,
  input [WIDTH-1:0] b_in,

  // Outputs
  output [WIDTH-1:0] a_out,
  output [WIDTH-1:0] b_out,
  output [WIDTH*2-1+$clog2(SIZE):0] c_result // Final accumulated result C_i,j
);

  // C width = 2*WIDTH + ceil(log2(SIZE)) for summation over SIZE elements
  localparam C_WIDTH = WIDTH*2 + $clog2(SIZE);
```

```
    // Registers to store matrix elements A, B, and the accumulating result C
    reg [WIDTH-1:0] a_reg;
    reg [WIDTH-1:0] b_reg;
    reg [C_WIDTH-1:0] c_reg;

    // Outputs are the registered values
    assign a_out = a_reg;
    assign b_out = b_reg;
    assign c_result = c_reg;

    always @(posedge clk) begin
        if (reset) begin
            a_reg <= 0;
            b_reg <= 0;
            c_reg <= 0;
        end else begin

            // 1. Shifting A/B and C Initialization
            if (shift_en) begin
                a_reg <= a_in;
                b_reg <= b_in;

                    // C Accumulator Initialization: Only during the one-cycle LOAD_INIT (shift_en=1,
compute_en=0)
                if (!compute_en) begin
                    c_reg <= 0; // Initialize C accumulator to zero for a new computation
                end
            end

            // 2. Compute (Multiply-Accumulate)
            // Uses the a_reg and b_reg values from the *previous* cycle (systolic pipeline)
            if (compute_en) begin
                // C(i,j) = C(i,j) + A(i,k) * B(k,j)
                c_reg <= c_reg + a_reg * b_reg;
            end
        end
    end
endmodule

module shift_register_34bit (
```

```
    input wire clk,
    input wire reset,
    input wire enable,
    input wire [33:0] data_in,      // Q18.16 format (34-bit)
    output wire [33:0] stage0,
    output wire [33:0] stage1,
    output wire [33:0] stage2,
    output wire [33:0] stage3,
    output wire [33:0] stage4,
    output wire [33:0] stage5,
    output wire [33:0] stage6,
    output wire [33:0] stage7,
    output wire [33:0] stage8
);
    // Internal registers for 9 stages (16-bit)
    reg [33:0] shift_reg [0:8];

    // Assign outputs
    assign stage0 = shift_reg[0];
    assign stage1 = shift_reg[1];
    assign stage2 = shift_reg[2];
    assign stage3 = shift_reg[3];
    assign stage4 = shift_reg[4];
    assign stage5 = shift_reg[5];
    assign stage6 = shift_reg[6];
    assign stage7 = shift_reg[7];
    assign stage8 = shift_reg[8];

    // Quantization conversion: Q18.16 to Q10.6
    // Q18.16 has 16 fractional bits, Q10.6 has 6 fractional bits
    // Right shift by 10 bits to convert (16 - 6 = 10)
    wire [33:0] quantized_data = data_in[33:0];

    integer i;

    always @(posedge clk or posedge reset) begin
        if (reset) begin
            // Reset all stages to 0
            for (i = 0; i < 9; i = i + 1) begin
                shift_reg[i] <= 0;
```

```
            end
        end
        else if (enable) begin
            // Shift quantized data through the pipeline
            shift_reg[0] <= quantized_data;
            for (i = 1; i < 9; i = i + 1) begin
                shift_reg[i] <= shift_reg[i-1];
            end
        end
    end
endmodule

module digit_8x16 #(
    parameter XSTART = 100,
    parameter YSTART = 50
)(
            input wire clk,
    input wire [10:0] h_count,
    input wire [10:0] v_count,
            input wire [3:0]  bcd,
            output wire pixon
);

    wire [3:0] row = v_count - YSTART;
    wire [2:0] col = h_count - XSTART;
        wire factive = (h_count >= XSTART) && (h_count < XSTART + 8) && (v_count >=
YSTART) && (v_count < YSTART + 16);
    wire fpixout;

    // Instance
    vga_digit_rom U1 (
        .clk(clk),
        .digit_code(bcd),
        .row(row),
        .col(7 - col),
        .pixel(fpixout)
    );

    assign pixon = factive ? fpixout : 1'b0;
```

```
endmodule
module clk_divider (
    input  wire clk_in,
    input  wire reset,
    output reg  clk_out,
    output reg clk_ram
);

    parameter integer DIVIDE_BY = 4; // Must be >= 2

    // Counter to track clock cycles
    reg [$clog2(DIVIDE_BY)-1:0] counter;

    always @(posedge clk_in) begin
        if (reset) begin
            counter <= 32'd0;
            clk_out <= 1'b0;
            clk_ram <= 1'b0;
        end else begin
            if (counter == (DIVIDE_BY - 1)) begin
                clk_out <= 1'b1; // Toggle enable clock pulse
                counter <= 32'd0;    // Reset counter
            end else begin
                counter <= counter + 1; // Increment counter
                clk_out <= 1'b0;
            end
            if (counter == (DIVIDE_BY / 2 - 1))
                clk_ram <= ~clk_ram; // Output out-of tree clock specific to digit lookup map
        end
    end

endmodule
`timescale 1ns / 1ps
//////////////////////////////////////////////////////////////////////////////////
// Company:
// Engineer:
//
// Create Date: 23.11.2025 11:08:12
// Design Name:
// Module Name: top
```

```
// Project Name:
// Target Devices:
// Tool Versions:
// Description:
//
// Dependencies:
//
// Revision:
// Revision 0.01 - File Created
// Additional Comments:
//
//////////////////////////////////////////////////////////////////////////////////


module top(
    input clk,
    output hsync,
    output vsync,
    output hblank,
    output vblank,
    output [3:0] vga_r,
    output [3:0] vga_g,
    output [3:0] vga_b
    );


    wire reset,read_ready,compute_done;
    wire [3*3*16-1:0] matrix_a, matrix_b;
    vga_rtl_top#(
    .X0(20),
    .Y0(50),
    .MATRIX_N(3),
    .MATRIX_M(3),
    .DIGITS(6),
    .WIDTH(16)
)temp(
    clk,
    reset,
    read_ready,
    matrix_a,
```

```
    matrix_b,
    compute_done,
    hsync,
    vsync,
    hblank,
    vblank,
    vga_r,
    vga_g,
    vga_b
);

    vio_0 hi (
  .clk(clk),                // input wire clk
  .probe_in0(compute_done),    // input wire [0 : 0] probe_in0
  .probe_in1(hsync),    // input wire [0 : 0] probe_in1
  .probe_in2(vsync),    // input wire [0 : 0] probe_in2
  .probe_in3(hblank),    // input wire [0 : 0] probe_in3
  .probe_in4(vblank),    // input wire [0 : 0] probe_in4
  .probe_in5(vga_r),    // input wire [3 : 0] probe_in5
  .probe_in6(vga_g),    // input wire [3 : 0] probe_in6
  .probe_in7(vga_b),    // input wire [3 : 0] probe_in7
  .probe_out0(reset),  // output wire [0 : 0] probe_out0
  .probe_out1(read_ready),  // output wire [0 : 0] probe_out1
  .probe_out2(matrix_a),  // output wire [143 : 0] probe_out2
  .probe_out3(matrix_b)  // output wire [143 : 0] probe_out3
);
endmodule

module vga_rtl_top #(
    parameter X0 = 20,
    parameter Y0 = 50,
    parameter MATRIX_N = 3,
    parameter MATRIX_M = 3,
    parameter DIGITS = 6,
    parameter WIDTH = 16
)(
    input  wire clk,
    input  wire reset,
    input read_ready,
    input [MATRIX_M*MATRIX_N*WIDTH - 1:0] matrix_a,
```

```
    input [MATRIX_M*MATRIX_N*WIDTH - 1:0] matrix_b,
    output wire compute_done,
    output wire hsync,
    output wire vsync,
    output wire hblank,
    output wire vblank,
    output wire [3:0] vga_r,
    output wire [3:0] vga_g,
    output wire [3:0] vga_b
);

    reg [4:0] outcount;
    reg start;

    always @(posedge clk) begin
        if (reset) begin
            start <= 0;
            outcount <= 0;
        end
        else begin
            if(outcount < 24) begin
                outcount <= outcount + 1;
                start <= 1;
            end
            else start <= 0;
        end
    end

    wire pixel_clk, bram_clk, v_en;
    wire [10:0] h_count, v_count;
    wire [(MATRIX_M * MATRIX_N * DIGITS * 3)-1:0] pixel;
    wire [2:0] brackets;

    // FIXED: Separate storage for input matrices (16-bit) and result matrix (34-bit)
    wire [WIDTH-1:0] matrices_input [0:MATRIX_N-1][0:(MATRIX_M*2)-1];  // A and B
    wire [33:0] matrices_result [0:MATRIX_N-1][0:MATRIX_M-1];          // C (34-bit)

    clk_divider #(
        .DIVIDE_BY(4)
    ) clk_div_inst (
```

```
    .clk_in(clk),
    .reset(reset),
    .clk_out(pixel_clk),
    .clk_ram(bram_clk)
);

// FIXED: Proper output assignments with correct bit widths
cannon #(
    .N(3),
    .WIDTH(16)
) cannon_inst (
    .clk(clk),
    .reset(reset),
    .start(start),
    .read_ready(read_ready),
    .mat_a_in(matrix_a),
    .mat_b_in(matrix_b),
    .output_valid(compute_done),
    .mata1(matrices_input[0][0]),
    .mata2(matrices_input[0][1]),
    .mata3(matrices_input[0][2]),
    .mata4(matrices_input[1][0]),
    .mata5(matrices_input[1][1]),
    .mata6(matrices_input[1][2]),
    .mata7(matrices_input[2][0]),
    .mata8(matrices_input[2][1]),
    .mata9(matrices_input[2][2]),
    .matb1(matrices_input[0][3]),
    .matb2(matrices_input[0][4]),
    .matb3(matrices_input[0][5]),
    .matb4(matrices_input[1][3]),
    .matb5(matrices_input[1][4]),
    .matb6(matrices_input[1][5]),
    .matb7(matrices_input[2][3]),
    .matb8(matrices_input[2][4]),
    .matb9(matrices_input[2][5]),
    .stage0(matrices_result[0][0]),
    .stage1(matrices_result[0][1]),
    .stage2(matrices_result[0][2]),
    .stage3(matrices_result[1][0]),
```

```
    .stage4(matrices_result[1][1]),
    .stage5(matrices_result[1][2]),
    .stage6(matrices_result[2][0]),
    .stage7(matrices_result[2][1]),
    .stage8(matrices_result[2][2]),
    .serial_c_out()
);

horizontal_counter horiz_counter_inst (
    .clk(clk),
    .clk_en(pixel_clk),
    .reset(reset),
    .hsync(hsync),
    .hblank(hblank),
    .en_v_count(v_en),
    .h_count(h_count)
);

vertical_counter vert_counter_inst (
    .clk(clk),
    .clk_en(pixel_clk),
    .reset(reset),
    .en_v_count(v_en),
    .vsync(vsync),
    .vblank(vblank),
    .v_count(v_count)
);

genvar i, k, l;
generate
    // Matrix A (j=0) - Q8.8 quantization
    for (i = 0; i < MATRIX_N; i = i + 1) begin : row_loop_A
        for (k = 0; k < MATRIX_M; k = k + 1) begin : col_loop_A
            wire [(DIGITS*4)-1:0] bcd_num_A;
            fixed_to_decimal #(
                .WIDTH(WIDTH),
                .FRAC_BITS(8)
            ) fixed_inst_A (
                .fixed_in(matrices_input[i][0 * MATRIX_M + k]),
                .concat(bcd_num_A)
```

```
            );
            for (l = 0; l < DIGITS; l = l + 1) begin : digit_loop_A
               digit_8x16 #(
                  .XSTART(X0 + (l * 8) + (k * 63) + (0 * 205)),
                  .YSTART(Y0 + (i * 30))
               ) digit_inst_A (
                  .clk(bram_clk),
                  .h_count(h_count),
                  .v_count(v_count),
                  .bcd(bcd_num_A[(l*4) +: 4]),
                   .pixon(pixel[(i * MATRIX_M * DIGITS * 3) + (0 * MATRIX_M * DIGITS) + (k
* DIGITS) + l])
               );
            end
         end
      end

      // Matrix B (j=1) - Q8.8 quantization
      for (i = 0; i < MATRIX_N; i = i + 1) begin : row_loop_B
         for (k = 0; k < MATRIX_M; k = k + 1) begin : col_loop_B
            wire [(DIGITS*4)-1:0] bcd_num_B;
            fixed_to_decimal #(
               .WIDTH(WIDTH),
               .FRAC_BITS(8)
            ) fixed_inst_B (
               .fixed_in(matrices_input[i][1 * MATRIX_M + k]),
               .concat(bcd_num_B)
            );
            for (l = 0; l < DIGITS; l = l + 1) begin : digit_loop_B
               digit_8x16 #(
                  .XSTART(X0 + (l * 8) + (k * 63) + (1 * 205)),
                  .YSTART(Y0 + (i * 30))
               ) digit_inst_B (
                  .clk(bram_clk),
                  .h_count(h_count),
                  .v_count(v_count),
                  .bcd(bcd_num_B[(l*4) +: 4]),
                   .pixon(pixel[(i * MATRIX_M * DIGITS * 3) + (1 * MATRIX_M * DIGITS) + (k
* DIGITS) + l])
               );
```

```
            end
        end
    end

    // Matrix C (j=2) - Q18.16 quantization (FIXED: Now uses 34-bit input)
    for (i = 0; i < MATRIX_N; i = i + 1) begin : row_loop_C
        for (k = 0; k < MATRIX_M; k = k + 1) begin : col_loop_C
            wire [(DIGITS*4)-1:0] bcd_num_C;
            fixed_to_decimal #(
                .WIDTH(34),
                .FRAC_BITS(16)
            ) fixed_inst_C (
                .fixed_in(matrices_result[i][k]),  // FIXED: Use 34-bit result
                .concat(bcd_num_C)
            );
            for (l = 0; l < DIGITS; l = l + 1) begin : digit_loop_C
                digit_8x16 #(
                    .XSTART(X0 + (l * 8) + (k * 63) + (2 * 205)),
                    .YSTART(Y0 + (i * 30))
                ) digit_inst_C (
                    .clk(bram_clk),
                    .h_count(h_count),
                    .v_count(v_count),
                    .bcd(bcd_num_C[(l*4) +: 4]),
                     .pixon(pixel[(i * MATRIX_M * DIGITS * 3) + (2 * MATRIX_M * DIGITS) + (k
* DIGITS) + l])
                );
            end
        end
    end
endgenerate

square_brackets_vga #(
    .LENGTH(90),
    .THICKNESS(2),
    .BRACES(6),
    .SEPARATION(63 + 63 + 48 + 6)
) brackets_inst (
    .x(h_count),
    .y(v_count),
```

```
        .X0(X0 - 6),
        .Y0(Y0),
        .pixel_on(brackets[0])
    );

    square_brackets_vga #(
        .LENGTH(90),
        .THICKNESS(2),
        .BRACES(6),
        .SEPARATION(63 + 63 + 48 + 6)
    ) brackets_inst2 (
        .x(h_count),
        .y(v_count),
        .X0(X0 + 205 - 6),
        .Y0(Y0),
        .pixel_on(brackets[1])
    );

    square_brackets_vga #(
        .LENGTH(90),
        .THICKNESS(2),
        .BRACES(6),
        .SEPARATION(63 + 63 + 48 + 6)
    ) brackets_inst3 (
        .x(h_count),
        .y(v_count),
        .X0(X0 + 410 - 6),
        .Y0(Y0),
        .pixel_on(brackets[2])
    );

    assign vga_r = (|pixel) | (|brackets) ? 4'hf : 4'h0;
    assign vga_g = (|pixel) | (|brackets) ? 4'hf : 4'h0;
    assign vga_b = (|pixel) | (|brackets) ? 4'hf : 4'h0;

endmodule
module horizontal_counter (
    input  wire clk,
    input wire clk_en,
    input  wire reset,
```

```
    output wire  hsync,
    output wire  hblank,
    output wire en_v_count,
    output reg [10:0] h_count
);

    // VGA 640x480 @60Hz timing parameters
    parameter H_VISIBLE_AREA   = 640;
    parameter H_FRONT_PORCH    = 16;
    parameter H_SYNC_PULSE     = 96;
    parameter H_BACK_PORCH     = 48;
        parameter   H_TOTAL                    = H_VISIBLE_AREA  +  H_FRONT_PORCH  +
H_SYNC_PULSE + H_BACK_PORCH;

     assign hsync = ~(h_count >= (H_VISIBLE_AREA + H_FRONT_PORCH) && h_count <
(H_VISIBLE_AREA + H_FRONT_PORCH + H_SYNC_PULSE)); // Generate hsync pulse
    assign hblank = (h_count >= H_VISIBLE_AREA); // Generate horizontal blanking signal
    assign en_v_count = (h_count == H_TOTAL - 2); // Generate vertical count enable signal

    always @(posedge clk) begin
        if (reset) begin
            h_count <= 11'd0;
        end else begin
            if (clk_en)
                h_count <= (h_count < H_TOTAL) ? h_count + 1 : 11'd0;
        end
    end

endmodule
module vertical_counter (
    input  wire clk,
    input wire clk_en,
    input  wire reset,
    input wire en_v_count,
    output wire  vsync,
    output wire  vblank,
    output reg [10:0] v_count
);

    // VGA 640x480 @60Hz timing parameters
```

```
    parameter V_VISIBLE_AREA   = 480;
    parameter V_FRONT_PORCH    = 10;
    parameter V_SYNC_PULSE     = 2;
    parameter V_BACK_PORCH     = 33;
         parameter   V_TOTAL                    =  V_VISIBLE_AREA  +  V_FRONT_PORCH  +
V_SYNC_PULSE + V_BACK_PORCH;

     assign vsync = ~(v_count >= (V_VISIBLE_AREA + V_FRONT_PORCH) && v_count <
(V_VISIBLE_AREA + V_FRONT_PORCH + V_SYNC_PULSE)); // Generate vsync pulse
    assign vblank = (v_count >= V_VISIBLE_AREA); // Generate vertical blanking signal

    always @(posedge clk) begin
        if (reset) begin
            v_count <= 11'd0;
        end else begin
            if (clk_en && en_v_count) begin
                if (v_count == V_TOTAL) v_count <= 11'd0;
                else v_count <= v_count + 1;
            end
        end
    end

endmodule
module square_brackets_vga #(
    parameter LENGTH = 100,        // Vertical length of brackets
    parameter THICKNESS = 5,       // Thickness of bracket lines
    parameter BRACES = 20,         // Length of horizontal parts
    parameter SEPARATION = 50      // Distance between brackets
)(
    input [9:0] x,          // Current pixel x coordinate
    input [9:0] y,          // Current pixel y coordinate
    input [9:0] X0,         // Top left corner x position
    input [9:0] Y0,         // Top left corner y position
    output reg pixel_on     // High when pixel is part of brackets
);

    // Left bracket components
    wire left_vertical, left_top_horiz, left_bottom_horiz;
    // Right bracket components
    wire right_vertical, right_top_horiz, right_bottom_horiz;
```

```
    // Left bracket - vertical bar
    assign left_vertical = (x >= X0) && (x < X0 + THICKNESS) &&
                           (y >= Y0) && (y < Y0 + LENGTH);

    // Left bracket - top horizontal brace
    assign left_top_horiz = (x >= X0) && (x < X0 + BRACES) &&
                            (y >= Y0) && (y < Y0 + THICKNESS);

    // Left bracket - bottom horizontal brace
    assign left_bottom_horiz = (x >= X0) && (x < X0 + BRACES) &&
                               (y >= Y0 + LENGTH - THICKNESS) && (y < Y0 + LENGTH);

    // Right bracket starting position
    wire [9:0] right_x0;
    assign right_x0 = X0 + BRACES + SEPARATION;

    // Right bracket - vertical bar (on the right side)
      assign right_vertical = (x >= right_x0 + BRACES - THICKNESS) && (x < right_x0 +
BRACES) &&
                            (y >= Y0) && (y < Y0 + LENGTH);

    // Right bracket - top horizontal brace
    assign right_top_horiz = (x >= right_x0) && (x < right_x0 + BRACES) &&
                             (y >= Y0) && (y < Y0 + THICKNESS);

    // Right bracket - bottom horizontal brace
    assign right_bottom_horiz = (x >= right_x0) && (x < right_x0 + BRACES) &&
                                (y >= Y0 + LENGTH - THICKNESS) && (y < Y0 + LENGTH);

    // Combine all components - pixel is on if it's part of any bracket component
    always @(*) begin
       pixel_on = left_vertical | left_top_horiz | left_bottom_horiz |
                  right_vertical | right_top_horiz | right_bottom_horiz;
    end

endmodule
`timescale 1ns / 1ps
module vga_digit_rom (
    input        clk,
```

```
    input  [3:0] digit_code, // 0â€“9
    input  [3:0] row,        // 0â€–15
    input  [2:0] col,        // 0â€–7
    output wire  pixel
);


    reg [7:0] rom [0:12][0:15];


    always @(posedge clk) begin
        // Zero
        rom [0][0] <= 8'h00; rom [0][1] <= 8'h00; rom [0][2] <= 8'h7c; rom [0][3] <= 8'h66;
        rom [0][4] <= 8'h66; rom [0][5] <= 8'h66; rom [0][6] <= 8'h66; rom [0][7] <= 8'h66;
        rom [0][8] <= 8'h66; rom [0][9] <= 8'h66; rom [0][10] <= 8'h66; rom [0][11] <= 8'h66;
        rom [0][12] <= 8'h7c; rom [0][13] <= 8'h00; rom [0][14] <= 8'h00; rom [0][15] <= 8'h00;


        //One
        rom [1][0] <= 8'h00; rom [1][1] <= 8'h00; rom [1][2] <= 8'h18; rom [1][3] <= 8'h38;
        rom [1][4] <= 8'h18; rom [1][5] <= 8'h18; rom [1][6] <= 8'h18; rom [1][7] <= 8'h18;
        rom [1][8] <= 8'h18; rom [1][9] <= 8'h18; rom [1][10] <= 8'h18; rom [1][11] <= 8'h18;
        rom [1][12] <= 8'h7e; rom [1][13] <= 8'h00; rom [1][14] <= 8'h00; rom [1][15] <= 8'h00;

// Two
rom [2][0] <= 8'h00; rom [2][1] <= 8'h00; rom [2][2] <= 8'h7c; rom [2][3] <= 8'h66;
rom [2][4] <= 8'h06; rom [2][5] <= 8'h0c; rom [2][6] <= 8'h18; rom [2][7] <= 8'h30;
rom [2][8] <= 8'h60; rom [2][9] <= 8'h60; rom [2][10] <= 8'h66; rom [2][11] <= 8'h66;
rom [2][12] <= 8'h7e; rom [2][13] <= 8'h00; rom [2][14] <= 8'h00; rom [2][15] <= 8'h00;



        // Three
        rom [3][0] <= 8'h00; rom [3][1] <= 8'h00; rom [3][2] <= 8'h7c; rom [3][3] <= 8'h66;
        rom [3][4] <= 8'h06; rom [3][5] <= 8'h3c; rom [3][6] <= 8'h06; rom [3][7] <= 8'h06;
        rom [3][8] <= 8'h66; rom [3][9] <= 8'h66; rom [3][10] <= 8'h66; rom [3][11] <= 8'h66;
        rom [3][12] <= 8'h7c; rom [3][13] <= 8'h00; rom [3][14] <= 8'h00; rom [3][15] <= 8'h00;

// Four
rom [4][0] <= 8'h00; rom [4][1] <= 8'h00; rom [4][2] <= 8'h0c; rom [4][3] <= 8'h1c;
```

```
rom [4][4] <= 8'h3c; rom [4][5] <= 8'h6c; rom [4][6] <= 8'hcc; rom [4][7] <= 8'hfe;
rom [4][8] <= 8'h0c; rom [4][9] <= 8'h0c; rom [4][10] <= 8'h0c; rom [4][11] <= 8'h0c;
rom [4][12] <= 8'h0c; rom [4][13] <= 8'h00; rom [4][14] <= 8'h00; rom [4][15] <= 8'h00;


    // Five
    rom [5][0] <= 8'h00; rom [5][1] <= 8'h00; rom [5][2] <= 8'hfc; rom [5][3] <= 8'hc0;
    rom [5][4] <= 8'hc0; rom [5][5] <= 8'hc0; rom [5][6] <= 8'hf8; rom [5][7] <= 8'h06;
    rom [5][8] <= 8'h06; rom [5][9] <= 8'h06; rom [5][10] <= 8'h66; rom [5][11] <= 8'h66;
    rom [5][12] <= 8'h7c; rom [5][13] <= 8'h00; rom [5][14] <= 8'h00; rom [5][15] <= 8'h00;


    // Six
    rom [6][0] <= 8'h00; rom [6][1] <= 8'h00; rom [6][2] <= 8'h7c; rom [6][3] <= 8'h66;
    rom [6][4] <= 8'h60; rom [6][5] <= 8'hc0; rom [6][6] <= 8'hf8; rom [6][7] <= 8'h66;
    rom [6][8] <= 8'h66; rom [6][9] <= 8'h66; rom [6][10] <= 8'h66; rom [6][11] <= 8'h66;
    rom [6][12] <= 8'h7c; rom [6][13] <= 8'h00; rom [6][14] <= 8'h00; rom [6][15] <= 8'h00;


    // Seven
    rom [7][0] <= 8'h00; rom [7][1] <= 8'h00; rom [7][2] <= 8'hfe; rom [7][3] <= 8'h66;
    rom [7][4] <= 8'h06; rom [7][5] <= 8'h0c; rom [7][6] <= 8'h18; rom [7][7] <= 8'h30;
    rom [7][8] <= 8'h60; rom [7][9] <= 8'h60; rom [7][10] <= 8'h60; rom [7][11] <= 8'h60;
    rom [7][12] <= 8'h60; rom [7][13] <= 8'h00; rom [7][14] <= 8'h00; rom [7][15] <= 8'h00;


    // Eight
    rom [8][0] <= 8'h00; rom [8][1] <= 8'h00; rom [8][2] <= 8'h7c; rom [8][3] <= 8'h66;
    rom [8][4] <= 8'h66; rom [8][5] <= 8'h66; rom [8][6] <= 8'h7c; rom [8][7] <= 8'h66;
    rom [8][8] <= 8'h66; rom [8][9] <= 8'h66; rom [8][10] <= 8'h66; rom [8][11] <= 8'h66;
    rom [8][12] <= 8'h7c; rom [8][13] <= 8'h00; rom [8][14] <= 8'h00; rom [8][15] <= 8'h00;


    // Nine
    rom [9][0] <= 8'h00; rom [9][1] <= 8'h00; rom [9][2] <= 8'h7c; rom [9][3] <= 8'h66;
    rom [9][4] <= 8'h66; rom [9][5] <= 8'h66; rom [9][6] <= 8'h7c; rom [9][7] <= 8'h06;
    rom [9][8] <= 8'h06; rom [9][9] <= 8'h06; rom [9][10] <= 8'h06; rom [9][11] <= 8'h06;
    rom [9][12] <= 8'h7c; rom [9][13] <= 8'h00; rom [9][14] <= 8'h00; rom [9][15] <= 8'h00;
```

```
        // multiplication symbol
        rom [10][0] <= 8'h00; rom [10][1] <= 8'h00; rom [10][2] <= 8'h00; rom [10][3] <= 8'h00;
        rom [10][4] <= 8'h81; rom [10][5] <= 8'h42; rom [10][6] <= 8'h24; rom [10][7] <= 8'h18;
        rom [10][8] <= 8'h18; rom [10][9] <= 8'h24; rom [10][10] <= 8'h42; rom [10][11] <= 8'h81;
           rom [10][12] <= 8'h00; rom [10][13] <= 8'h00; rom [10][14] <= 8'h00; rom [10][15] <=
8'h00;


        // Equals symbol
        rom [11][0] <= 8'h00; rom [11][1] <= 8'h00; rom [11][2] <= 8'h00; rom [11][3] <= 8'h00;
        rom [11][4] <= 8'h00; rom [11][5] <= 8'hff; rom [11][6] <= 8'hff; rom [11][7] <= 8'h00;
        rom [11][8] <= 8'h00; rom [11][9] <= 8'hff; rom [11][10] <= 8'hff; rom [11][11] <= 8'h00;
           rom [11][12] <= 8'h00; rom [11][13] <= 8'h00; rom [11][14] <= 8'h00; rom [11][15] <=
8'h00;


        // Decimal point
        rom [12][0] <= 8'h00; rom [12][1] <= 8'h00; rom [12][2] <= 8'h00; rom [12][3] <= 8'h00;
        rom [12][4] <= 8'h00; rom [12][5] <= 8'h00; rom [12][6] <= 8'h00; rom [12][7] <= 8'h00;
        rom [12][8] <= 8'h00; rom [12][9] <= 8'h00; rom [12][10] <= 8'h00; rom [12][11] <= 8'h18;
           rom [12][12] <= 8'h18; rom [12][13] <= 8'h00; rom [12][14] <= 8'h00; rom [12][15] <=
8'h00;


    end


    assign pixel = rom[digit_code][row][col];


endmodule

`timescale 1ns / 1ps
//////////////////////////////////////////////////////////////////////////////////
// Company:
// Engineer:
//
// Create Date: 23.11.2025 11:08:12
// Design Name:
// Module Name: top
```

```
// Project Name:
// Target Devices:
// Tool Versions:
// Description:
//
// Dependencies:
//
// Revision:
// Revision 0.01 - File Created
// Additional Comments:
//
//////////////////////////////////////////////////////////////////////////////////


module top(
    input clk,
    output hsync,
    output vsync,
    output hblank,
    output vblank,
    output [3:0] vga_r,
    output [3:0] vga_g,
    output [3:0] vga_b
    );


    wire reset,read_ready,compute_done;
    wire [3*3*16-1:0] matrix_a, matrix_b;
    vga_rtl_top#(
    .X0(20),
    .Y0(50),
    .MATRIX_N(3),
    .MATRIX_M(3),
    .DIGITS(6),
    .WIDTH(16)
)temp(
    clk,
    reset,
    read_ready,
    matrix_a,
```

```
    matrix_b,
    compute_done,
    hsync,
    vsync,
    hblank,
    vblank,
    vga_r,
    vga_g,
    vga_b
);

    vio_0 hi (
  .clk(clk),                // input wire clk
  .probe_in0(compute_done),    // input wire [0 : 0] probe_in0
  .probe_in1(hsync),    // input wire [0 : 0] probe_in1
  .probe_in2(vsync),    // input wire [0 : 0] probe_in2
  .probe_in3(hblank),    // input wire [0 : 0] probe_in3
  .probe_in4(vblank),    // input wire [0 : 0] probe_in4
  .probe_in5(vga_r),    // input wire [3 : 0] probe_in5
  .probe_in6(vga_g),    // input wire [3 : 0] probe_in6
  .probe_in7(vga_b),    // input wire [3 : 0] probe_in7
  .probe_out0(reset),  // output wire [0 : 0] probe_out0
  .probe_out1(read_ready),  // output wire [0 : 0] probe_out1
  .probe_out2(matrix_a),  // output wire [143 : 0] probe_out2
  .probe_out3(matrix_b)  // output wire [143 : 0] probe_out3
);
endmodule
```

Testbench Code:

```
`timescale 1ns / 1ps

module tb_vga_rtl_top;

     parameter X0 = 20;
    parameter Y0 = 50;
    parameter MATRIX_N = 3;
    parameter MATRIX_M = 3;
    parameter DIGITS = 6;
    parameter WIDTH = 16;

    reg clk;
```

```
reg reset;
reg read_ready;

// Flattened inputs
reg [MATRIX_M*MATRIX_N*WIDTH - 1:0] matrix_a;
reg [MATRIX_M*MATRIX_N*WIDTH - 1:0] matrix_b;

// Outputs
wire compute_done;
wire hsync;
wire vsync;
wire hblank;
wire vblank;
wire [3:0] vga_r;
wire [3:0] vga_g;
wire [3:0] vga_b;


reg [WIDTH-1:0] tb_mat_a [0:MATRIX_N-1][0:MATRIX_M-1];
reg [WIDTH-1:0] tb_mat_b [0:MATRIX_N-1][0:MATRIX_M-1];

vga_rtl_top #(
    .X0(X0),
    .Y0(Y0),
    .MATRIX_N(MATRIX_N),
    .MATRIX_M(MATRIX_M),
    .DIGITS(DIGITS),
    .WIDTH(WIDTH)
) uut (
    .clk(clk),
    .reset(reset),
    .read_ready(read_ready),
    .matrix_a(matrix_a),
    .matrix_b(matrix_b),
    .compute_done(compute_done),
    .hsync(hsync),
    .vsync(vsync),
    .hblank(hblank),
    .vblank(vblank),
    .vga_r(vga_r),
```

```
    .vga_g(vga_g),
    .vga_b(vga_b)
);

initial begin
    clk = 0;
    forever #5 clk = ~clk; // 10ns period
end


integer i, j;


function [15:0] to_fixed;
    input integer val;
    begin

        to_fixed = val << 8;
    end
endfunction

initial begin
     reset = 1;
    read_ready = 0;
    matrix_a = 0;
    matrix_b = 0;


    tb_mat_a[0][0] = to_fixed(1); tb_mat_a[0][1] = to_fixed(0); tb_mat_a[0][2] = to_fixed(0);
    tb_mat_a[1][0] = to_fixed(0); tb_mat_a[1][1] = to_fixed(1); tb_mat_a[1][2] = to_fixed(0);
    tb_mat_a[2][0] = to_fixed(0); tb_mat_a[2][1] = to_fixed(0); tb_mat_a[2][2] = to_fixed(1);


    tb_mat_b[0][0] = to_fixed(1); tb_mat_b[0][1] = to_fixed(2); tb_mat_b[0][2] = to_fixed(3);
    tb_mat_b[1][0] = to_fixed(4); tb_mat_b[1][1] = to_fixed(5); tb_mat_b[1][2] = to_fixed(6);
    tb_mat_b[2][0] = to_fixed(7); tb_mat_b[2][1] = to_fixed(8); tb_mat_b[2][2] = to_fixed(9);

        for (i = 0; i < MATRIX_N; i = i + 1) begin
         for (j = 0; j < MATRIX_M; j = j + 1) begin
            // Update specific slices of the wide vectors
```

```
            matrix_a[(i*MATRIX_M + j)*WIDTH +: WIDTH] = tb_mat_a[i][j];
            matrix_b[(i*MATRIX_M + j)*WIDTH +: WIDTH] = tb_mat_b[i][j];
        end
    end

    // 4. Release Reset
    #100;
    reset = 0;
    $display("[%0t] Reset released", $time);

        wait(compute_done);
    $display("[%0t] Computation Done Signal Asserted!", $time);


    read_ready = 1;

     repeat(50000) @(posedge clk);

    $display("[%0t] Simulation Finished", $time);
    $finish;
  end


endmodule
```