Design and Physical Implementation of a Neural-Network MAC Tile Accelerator

EE413 Introduction to VLSI Design

Term Project Report

Mustafa Alp Ekici 2521524
Sarp Ilgın Sarısu 2516847

Department of Electrical and Electronics Engineering
Middle East Technical University

February 1, 2026

1 Introduction

This dramatic rise of Artificial Intelligence (AI) and Machine Learning (ML) technologies in edge computing environments has led to the realization that traditional general-purpose Processors are lacking when it comes to intensive computational operations. Deep Neural Networks (DNNs) essentially perform a plethora of operations based entirely on the traditional concept of matrix vector multiply operations. However, considering the natural overheads of traditional CPUs due to their Von Neumann Architecture, ASICs are crucial for application acceleration. This project attempts to bridge this gap by developing a hardware accelerator optimized for the multiply accumulate operation that can perform operations between input neurons and output neurons for a fully connected layer.

Targeting the XFAB 180nm CMOS technology node, this work covers the complete digital design flow starting from RTL verification in Verilog and ending in logic synthesis to final physical implementation. The accelerator can calculate one tile of the neural network expressed as $Y[j] = \sum W[j][i] \cdot X[i]$, with fixed-point arithmetic to balance precision with hardware efficiency. The result of physical design obtains critical VLSI constraints in terms of floorplanning, power distribution, and signal integrity while the final layout should satisfy the targeted timing frequency of 50 MHz under manufacturing design rules.

The core of this report is a comparative analysis between two disparate architectural approaches: a Single-MAC architecture and a Dual-MAC architecture. In the Single-MAC design, area efficiency is pursued through the time-sharing approach of a single arithmetic unit, while in the Dual-MAC, parallelism is exploited to increase computational throughput by a factor of two. This work does a quantitative examination of the trade-offs among silicon area, power consumption, and processing latency by implementing both architectures up to the GDSII stage. Successful post-layout verification in terms of DRC, LVS, and STA for both designs provides clear insights into hardware optimization for embedded AI applications.

2 System Specifications and Constraints

The design of the MAC accelerator is governed by a strict set of specifications. These are defined to create compatibility with embedded system requirements and the limitations of the target technology. The core computational task involves a 4×8 matrix-vector multiplication. Specifically, the system must process an input vector X consisting of 8 elements and a weight matrix W containing 32 elements. These inputs produce an output vector Y consisting of 4 elements.

2.1 Numerical Representation

Precision and dynamic range are important in DSP designs. In this project we use fixed-point arithmetic to avoid the large area and power cost of floating-point hardware. The inputs and weights are 8-bit signed Q1.6 numbers, which has one sign bit, one integer bit, and six fractional bits. By using this format the values in the range $[-2.0, +2.0)$ can be represented. However, in our use case the inputs are constrained to $[-1.0, +1.0)$.

Width selection of the accumulator is a very important because the sum grows in bit-width. Multiplying two 8-bit values produces a 16-bit product. Each output accumulates eight products, hence the worst-case sum needs extra headroom. Adding eight terms requires $\lceil \log_2(8) \rceil = 3$ extra bits. As a result of this, a minimum of 19 bits is necessary to avoid overflow. By taking these into consideration, we selected a 24-bit accumulator to keep a safety margin and to preserve precision. This also avoids adding saturation logic, which would complicate control and can increase the critical path.

2.2 System Interface and Pin Planning

A major limitation in VLSI design is the number of available I/O pins. If all 8 input values and 32 weight values were provided in parallel, more than 320 signal pins would be required. This is clearly impractical for a compact accelerator and would cause high packaging cost and severe routing congestion. For this reason, a serial input interface was chosen.

The accelerator uses a small and simple set of interface signals, including the clock (clk), synchronous reset (rst), and a start signal (start). All input data is transferred through a single 8-bit port, serial_data_in. The weight values and input activations are streamed sequentially over this port. An internal counter together with the FSM routes each incoming byte to the correct internal register.

The output data is also streamed serially through an 8-bit Y_out port and qualified by the output_valid signal. With this approach, the total number of I/O pins is reduced to 24, which significantly simplifies the I/O ring and top-level routing.

The reduction in pin count requires a robust internal control mechanism to track the reception of data. The design assumes that the external system provides valid data in a specific order—first, the 32 weight values, followed by the 8 input activation values—before establishing the start signal to begin the computation.

3 Architectural Design

The internal architecture of the accelerator is divided into two main components: the datapath, which performs the arithmetic operations, and the controller, which orchestrates the overall sequence of operations. The datapath and control structures described in this section are directly derived from the implemented Verilog RTL code. For clarity, detailed block-level diagrams illustrating these architectures are provided in Appendix C.

Two distinct architectures were developed to explore the design space: a baseline Single-MAC architecture and an enhanced Dual-MAC architecture. In both cases, the external interface and transaction protocol remain identical, while the internal datapath organization differs.

3.1 Baseline Architecture: Single-MAC

The block-level organization of the Single-MAC architecture is illustrated in Appendix C (see Figure 1). This architecture is designed to minimize area by reusing a single multiply–accumulate datapath across all computations.

In the Single-MAC design, a single combinational multiplier and adder are used, together with a 24-bit accumulator register. The computation of the four output values ($Y[0]$ to $Y[3]$) is performed sequentially. For each output $Y[j]$, the controller iterates over the inner loop index i from 0 to 7. In every clock cycle, the corresponding weight $W[j][i]$ and input activation $X[i]$ are fetched from the internal register banks and applied to the multiplier. The resulting 16-bit product is sign-extended to match the accumulator width and added to the current accumulated sum.

After all eight multiply–accumulate operations for a given output are completed, the final accumulated value is transferred to an output register. The accumulator is then cleared, and the process is repeated for the next output neuron.

In addition to the cycles needed for data loading and output streaming, the architecture needs a total of 32 MAC cycles for the arithmetic core (4 outputs $\times$ 8 accumulations). While the sequential operation increases latency, the Single-MAC design serves as an area-efficient baseline for comparison with the Dual-MAC variant.

3.2 Enhanced Architecture: Dual-MAC

The Dual-MAC architecture extends the Single-MAC concept by introducing a second multiply–accumulate datapath to increase computational parallelism. The block diagram for this architecture is provided in Appendix C (see Figure 2).

In the Dual-MAC architecture, two identical MAC units operate concurrently under a shared control structure. The input activation memory and weight storage are reused, while the arithmetic datapath is duplicated. Each MAC unit is equipped with its own accumulator register, allowing two independent partial sums to be computed in parallel. The controller schedules these MAC operations such that two multiply–accumulate operations are performed in each clock cycle.

This parallel organization reduces the number of cycles required during the compute phase when compared to the Single-MAC design. Partial sums corresponding to different output neurons can be accumulated simultaneously, effectively halving the arithmetic latency of the MAC stage. Importantly, the external interface and transaction protocol remain unchanged. Output values are still streamed sequentially using the same output_valid and done signaling scheme.

The primary cost of this architectural enhancement is an increase in silicon area and routing complexity due to the duplicated multiplier and adder logic. However, this cost is modest compared to the achieved performance improvement.

3.3 Finite State Machine (FSM) Design

Both the Single-MAC and Dual-MAC architectures are controlled by a centralized Finite State Machine (FSM). The FSM state diagram, which applies to both designs, is shown in Appendix C (see Figure 3).

The FSM starts in the S_IDLE state and waits for the assertion of the start signal. Once activated, the controller moves to the S_LOAD state, where weight and input values are received serially through serial_data_in. A load counter tracks the reception of 32 weights followed by 8 input activations.

After loading is complete, the FSM enters the S_CLEAR state to reset the accumulator and internal indices. The main computation takes place in the S_MAC state. In the Single-MAC design, this state iterates eight times per output, while in the Dual-MAC design, fewer iterations are required due to parallel execution. Once an output value is fully accumulated, the FSM transitions to the S_STORE_Y state, where the result is stored. After all four outputs are computed, the FSM enters the S_STREAM_OUT phase, followed by the S_DONE state.

4 RTL Implementation

This section summarizes the RTL realization of the MAC accelerator in synthesizable Verilog HDL. Quartus RTL Viewer was used to inspect the generated hierarchy and to confirm that the datapath (multiplier/accumulator and memories) and the controller (FSM and counters) are connected as intended.

4.1 Single-MAC RTL View

The Quartus generated RTL views for the Single-MAC design are included in Appendix C. Figure 4 in the appendix displays the top-level hierarchy, highlighting the serial input storage for W and X, the single MAC datapath that is reused across all four outputs, and the controller logic. Figure 5 (Appendix C) provides a detailed view capturing both the arithmetic chain (multiplier and accumulator) and the FSM-based controller.

4.2 Dual-MAC RTL View

The corresponding RTL views for the Dual-MAC design are also provided in **Appendix C**. Figure 6 shows the hierarchy exposing the duplicated arithmetic lanes used to increase parallelism. Figure 7 highlights the parallel MAC structure together with the controller signals used for scheduling and output streaming.

5 Logic Synthesis

Verilog RTL description was synthesized by Cadence Genus. This process transform high level hardware descriptions into gate level netlist using standard cells from the the XFAB 180nm technology library. Design constraints were set up for a target clock frequency of 50 MHz.

The synthesis reports state that both designs were successfully mapped to the target library with no setup-time violations. The synthesis tool, for the Single-MAC design, inferred a logic structure that was compact and required little buffering. For the Dual-MAC design, the tool was able to synthesize two arithmetic pathways in parallel. At this point, the slack values for both designs were positive. This was a sign that the logic depth was restricted for operation at 50 MHz.

6 Physical Implementation

The physical implementation was done using Cadence Innovus for the gate level netlist translation. This step includes layout design, standard cell placement, clock tree construction, and final routing.

6.1 Floorplanning and Power Planning

The floorplan of the chip specifies the boundaries of the integrated circuit and the locations of the input and output pins. For the current design project, the Single MAC and Dual MAC designs were each assigned a die area of 500 $\mu m \times$ 500 μm. This area was sufficient to place standard cells without too much crowding, and it was also adequate for the routing of the power and the placement of buffers.

6.2 Placement

The placement phase involves positioning standard cells within a defined core area. To prevent local congestion and allow track availability for routing, the density restriction is set at a maximum of 75%. Innovus accomplished a timing-driven placement by clustering cells that are on the same critical paths to reduce wire delays.

The single MAC design had lower space utilization due to the smaller number of cells. In contrast, due to the cell increase, there was higher placement density in the dual MAC design. More precisely, the placement density for the dual MAC was about 39%, whereas it was about 35% for the single MAC. This higher density was managed quite efficiently by the tool, and no significant placement violations were observed. Furthermore, the uniform distribution of cells ensures thermal stability and prevents hot spots.

6.3 Clock Tree Synthesis (CTS)

Clock Tree Synthesis is a crucial step to ensure that the clock signal reaches all sequential elements (flip-flops) at approximately the same time. With a large Clock Skew, there is bound to be a hold time violation and functional failure. For the above, Innovus constructed a balanced clock tree using buffers and inverters from the XFAB library for the CTS engine.

The post-CTS analysis for the Dual-MAC design showed a well-balanced tree with minimal skew. The tool reported a fanout distribution where the appropriate buffers maintained signal transition times within limits. The clock tree visualization shows a star-like topology spreading from the central clock entry point into the varied registers distributed across the chip.

6.4 Routing

The final step in the implementation process is routing, in which the logical connections between the cells are accomplished using metal wires. In the XFAB 180nm process, there are several layers of metal available for routing. The routing engine was able to successfully route both designs without any shorts or opens.

In the analysis of the congestion during the routing, the result revealed that the floor plan of size 500 $\mu m\times$ 500 $\mu m.$ had sufficient routing resources available. The global routing and detail routing phases finished with 0 errors and 0 violation messages. The log files indicate the routed implementation connects all nets with no DRC error regarding metal spacing and width. However, there exist antenna rule errors, which shall be validated in the later validation phase.

7 Post-Layout Verification and Analysis

After the physical implementation, a comprehensive verification was performed to evaluate whether the design meets the physical constraints.

7.1 Physical Verification (DRC)

The Design Rule Check (DRC) process was applied to both Single MAC and Dual MAC designs. The only problems identified in both designs were antenna problems, which were solved by adding vias to long metal lines. Therefore, both final designs had no errors in their DRC results. Figures demonstrating the clean DRC status for both designs are provided in Appendix D.

7.2 Layout vs. Schematic (LVS)

The Layout vs. Schematic (LVS) check was performed, verifying the internal connectivity of the physical layout and the net topology being identical to the netlist generated. Nevertheless, the tool pointed out some discrepancies regarding the labels of the I/O pins. It should be noted that they are spurious discrepancies generated due to the limitations imposed by the Cadence label extraction approach, rather than a layout error. As such, they have no effect on the signal flow, hence waived, making the design LVS compatible. Figures demonstrating the LVS status for both designs are provided in Appendix D.

8 Performance Comparison and Discussion

This project focused on comparing Single-MAC and Dual-MAC Architectures. Data from synthesis and physical implementation allows for a comprehensive comparison based on various criteria.

8.1 Area Analysis

The post-synthesis cell report files illustrate the detailed architecture of the two designs and the effect of parallelization on the cost of the hardware. Table 1 shows the total area and instance counts, further categorized by cell type.

Table 1: Detailed Area and Instance Count Comparison

Metric	**Single-MAC**	**Dual-MAC**	**Change (%)**
Total Cell Area (μm^2)	**36,096.397**	**40,283.179**	**+11.60%**
Total Instance Count	1412	1576	+11.61%
Distribution by Cell Type			
Sequential Area (μm^2)	19,221.148	19,965.153	+3.87%
Sequential Instances	390	405	+3.85%
Combinational Logic Area (μm^2)	16,639.346	20,027.683	+20.36%
Combinational Instances	983	1123	+14.24%

While the dual MAC architecture theoretically doubles computational efficiency, the overall area increase is only 11.6 percent. This is due to the different scaling behaviors of the sequential and combinatorial logic components. The area utilized by sequential cells (flip-flops and registers) has not changed much, showing only a slight increase of approximately 3.8%. This serves as a positive validation of the architectural strategy of sharing the core storage components. In both architectures, identical register banks are employed to hold the 8 input activations and the 32 weights. The increase of 15 in the number of sequential components (from 390 to 405) can be accounted for by a small increase in the number of FSM states and the extra control pipeline registers needed to handle parallel data flow. The primary reason for the increase in area is the approximately 20.3% growth in combinatorial logic circuits. This increase stems from the creation of a second arithmetic data bus (an additional multiplier and adder).

8.2 Timing and Latency Analysis

The post-synthesis timing report files provide detailed information about the system's performance relative to the target frequency of the system, which is 50 MHz (20 ns clock period). The timing analysis clearly shows that the Dual-MAC architecture achieves a superior timing margin compared to the Single-MAC. This performance comparison is shown in Table 2.

The critical path in the Single-MAC design starts at the internal loop control registers and terminates at the accumulator storage registers. Total path delay is 14170 ps with a positive slack of 5830 ps. This shows that the design can operate at a maximum delay of 70.5 MHz with a timing margin of 29% at a 50 MHz target. The Dual-MAC architecture has a path delay of 13754 ps, which means there is more positive slack of 6246 ps. The critical path still starts from the shared loop control logic but terminates at one of the parallel accumulator registers. The enhancement in timing indicates that the synthesis tool was able to optimize the parallel datapath more effectively, probably because of less multiplexing, better organization of control logic depth, or more efficient logic in the arithmetic units.

Table 2: Post-Synthesis Timing Analysis Comparison

Metric	**Single-MAC**	**Dual-MAC**
Target Clock Period	20 ns	20 ns
Critical Path Delay (Arrival Time)	14170 ps	13754 ps
Timing Slack (Positive)	**+5,830 ps**	**+6,246 ps**
Maximum Frequency (F_{max})	$\approx$ 70.5 MHz	$\approx$ 72.7 MHz

The Dual-MAC has demonstrated faster performance than a single-MAC, requiring half the number of cycles to complete arithmetic workloads. These two performance criteria prove that Dual-MAC is a far superior architecture for applications requiring high throughput.

8.3 Power-Performance-Area (PPA) Trade-off

The post-synthesis power report files provide information about the power consumption of the two architectures after synthesis performed at a target frequency of 50 MHz. Table 3 presents a summary of the power consumption for both designs.

Table 3: Power Consumption Analysis Comparison

Metric	**Single-MAC**	**Dual-MAC**	**Change (%)**
Leakage Power	8.070 nW	9.021 nW	+11.78%
Dynamic Power	3,059,539 nW	3,429,335 nW	+12.08%
Total Power	**3.060 mW**	**3.429 mW**	**+12.09%**

The Dual-MAC architecture consumes around 3.43 mW, which is only 12.1% higher than the Single-MAC. This increase is closely related to the 11.6% increase in area and is primarily due to the dynamic switching of the additional combinatorial logic in the second MAC unit. However, instantaneous power consumption doesn't tell the whole story. Energy per operation is also an important parameter. ($E = P \times t$).

Since the Dual-MAC architecture completes the computation task in half the clock cycles, 16 cycles compared to 32 cycles, the device's active duration is reduced by 50%. As a result, even though the Dual-MAC uses about 12% more current while running, it operates for only half the time. This leads to a net reduction in total energy consumption of about 44% per inference task when compared to the Single-MAC baseline.

9 Conclusion

This term project has successfully shown the implementation of a digital VLSI accelerator used for neural network inference, which utilized the XFAB 180nm node technology flow. The term project has confirmed the viability of the VLSI accelerator used for AI with specific constraints in terms of the overall area and pins used.

The results of such a comparative analysis of Single-MAC and Dual-MAC architecture designs have been decisive with respect to hardware trade-offs. The Dual-MAC architecture was clearly ahead of its counterpart, resulting in a 2x increase in computation throughput with an associated increase of only 11.6% increase in silicon area. Significantly, even with a 12% increase in instantaneous power consumption, an energy reduction of 44% was observed due to a 50% decrease in execution time through parallel computation.

Considering a physical implementation scenario, both designs achieved 50 MHz timing constraints with positive slack values, and the Dual MAC also achieved better timing margins due to proper synthesis optimization. Further, the designs were also DRC clean after fixing the antenna violation issues, proving that the designs are LVS compliant as well. Also, the use of a serial interface strategy was successful in keeping the number of pins under the limit of 24, thus proving that a large degree of processing inside the chip cannot limit the Input/Output capabilities. Further designs could also consider pipelining the MAC operations to increase the frequency beyond 100 MHz or scaling up the systolic array size.

A Appendix A: RTL Source Codes

A.1 Single-MAC Accelerator RTL

Listing 1: Complete Verilog RTL of the Single-MAC accelerator

```
module mac_accelerator (
    input  wire              clk,
    input  wire              rst,
    input  wire              start,
    input  wire signed [7:0]  serial_data_in,
    input  wire              data_valid,
    output reg  signed [7:0]  Y_out,
    output reg               done,
    output reg               output_valid
);

    reg signed [7:0] X_mem [0:7];
    reg signed [7:0] W_mem [0:31];

    localparam integer ACC_W = 24;
    reg signed [ACC_W-1:0] accumulator;

    reg [5:0] load_counter;
    reg [2:0] i_idx;
    reg [1:0] j_idx;

    reg signed [7:0] Y_mem [0:3];

    localparam S_IDLE       = 3'd0;
    localparam S_LOAD       = 3'd1;
    localparam S_CLEAR      = 3'd2;
    localparam S_MAC        = 3'd3;
    localparam S_STORE_Y    = 3'd4;
    localparam S_STREAM_OUT = 3'd5;
    localparam S_DONE       = 3'd6;

    reg [2:0] state;

    wire [4:0] waddr = {j_idx, i_idx};
    wire signed [7:0] x_val = X_mem[i_idx];
    wire signed [7:0] w_val = W_mem[waddr];
    wire signed [15:0] prod = w_val * x_val;
    wire signed [ACC_W-1:0] prod_ext = {{(ACC_W-16){prod[15]}}, prod
        };

    function [7:0] acc_to_q16;
        input signed [ACC_W-1:0] a;
        reg signed [ACC_W-1:0] shifted;
        begin
            shifted = a >>> 6;
            acc_to_q16 = shifted[7:0];
        end
    endfunction

    integer k;
    always @(posedge clk) begin
        if (rst) begin
            state <= S_IDLE;

            accumulator  <= 24'sd0;
            load_counter <= 6'd0;
            i_idx        <= 3'd0;
            j_idx        <= 2'd0;

            done         <= 1'b0;
            output_valid <= 1'b0;
            Y_out        <= 8'sd0;

            for (k = 0; k < 8;  k = k + 1) X_mem[k] <= 8'sd0;
            for (k = 0; k < 32; k = k + 1) W_mem[k] <= 8'sd0;
            for (k = 0; k < 4;  k = k + 1) Y_mem[k] <= 8'sd0;

        end else begin
            done         <= 1'b0;
            output_valid <= 1'b0;

            case (state)
                S_IDLE: begin
                    load_counter <= 6'd0;
                    i_idx        <= 3'd0;
                    j_idx        <= 2'd0;
                    accumulator  <= 24'sd0;
                    if (start) state <= S_LOAD;
                end

                S_LOAD: begin
                    if (data_valid) begin
                        if (load_counter < 6'd32)
                            W_mem[load_counter[4:0]] <=
                               serial_data_in;
                        else
                            X_mem[load_counter - 6'd32] <=
                               serial_data_in;

                        if (load_counter == 6'd39) begin
                            j_idx  <= 2'd0;
                            state  <= S_CLEAR;
                        end else begin
                            load_counter <= load_counter + 6'd1;
                        end
                    end
                end

                S_CLEAR: begin
                    accumulator <= 24'sd0;
                    i_idx       <= 3'd0;
                    state       <= S_MAC;
                end

                S_MAC: begin
                    accumulator <= accumulator + prod_ext;
                    if (i_idx == 3'd7)
                        state <= S_STORE_Y;
                    else
                        i_idx <= i_idx + 3'd1;
                end

                S_STORE_Y: begin
                    Y_mem[j_idx] <= acc_to_q16(accumulator);
                    if (j_idx == 2'd3) begin
                        j_idx  <= 2'd0;
                        state  <= S_STREAM_OUT;
                    end else begin
```

```
                        j_idx   <= j_idx + 2'd1;
                        state   <= S_CLEAR;
                    end
                end

                S_STREAM_OUT: begin
                    Y_out         <= Y_mem[j_idx];
                    output_valid <= 1'b1;
                    if (j_idx == 2'd3)
                        state <= S_DONE;
                    else
                        j_idx <= j_idx + 2'd1;
                end

                S_DONE: begin
                    done   <= 1'b1;
                    state <= S_IDLE;
                end

                default: state <= S_IDLE;
            endcase
        end
    end

endmodule
```

A.2 Dual-MAC Accelerator RTL

Listing 2: Complete Verilog RTL of the Dual-MAC accelerator

```
module mac_accelerator_dual (
    input  wire                clk,
    input  wire                rst,
    input  wire                start,
    input  wire signed [7:0]   serial_data_in,
    input  wire                data_valid,
    output reg  signed [7:0]   Y_out,
    output reg                 done,
    output reg                 output_valid
);

    reg signed [7:0] X_mem [0:7];
    reg signed [7:0] W_mem [0:31];

    localparam integer ACC_W = 24;

    reg signed [ACC_W-1:0] acc0;
    reg signed [ACC_W-1:0] acc1;

    reg [5:0] load_counter;
    reg [2:0] i_idx;
    reg [1:0] j_base;
    reg [1:0] out_idx;

    reg signed [7:0] Y_mem [0:3];

    localparam S_IDLE       = 3'd0;
    localparam S_LOAD       = 3'd1;
    localparam S_CLEAR      = 3'd2;
    localparam S_MAC        = 3'd3;
    localparam S_STORE_PAIR = 3'd4;
    localparam S_STREAM_OUT = 3'd5;
    localparam S_DONE       = 3'd6;

    reg [2:0] state;

    wire signed [7:0] x_val = X_mem[i_idx];

    wire [4:0] waddr0 = {j_base, i_idx};
    wire [4:0] waddr1 = {(j_base + 2'd1), i_idx};

    wire signed [7:0] w0 = W_mem[waddr0];
    wire signed [7:0] w1 = W_mem[waddr1];

    wire signed [15:0] p0 = w0 * x_val;
    wire signed [15:0] p1 = w1 * x_val;

    wire signed [ACC_W-1:0] p0_ext = {{(ACC_W-16){p0[15]}}, p0};
    wire signed [ACC_W-1:0] p1_ext = {{(ACC_W-16){p1[15]}}, p1};

    function [7:0] acc_to_q16;
        input signed [ACC_W-1:0] a;
        reg signed [ACC_W-1:0] shifted;
        begin
            shifted = a >>> 6;
            acc_to_q16 = shifted[7:0];
        end
    endfunction

    integer k;
    always @(posedge clk) begin
        if (rst) begin
            state <= S_IDLE;

            acc0 <= 24'sd0;
            acc1 <= 24'sd0;

            load_counter <= 6'd0;
            i_idx        <= 3'd0;
            j_base       <= 2'd0;
            out_idx      <= 2'd0;

            done         <= 1'b0;
            output_valid <= 1'b0;
            Y_out        <= 8'sd0;

            for (k = 0; k < 8;  k = k + 1) X_mem[k] <= 8'sd0;
            for (k = 0; k < 32; k = k + 1) W_mem[k] <= 8'sd0;
            for (k = 0; k < 4;  k = k + 1) Y_mem[k] <= 8'sd0;

        end else begin
            done         <= 1'b0;
            output_valid <= 1'b0;

            case (state)
                S_IDLE: begin
                    load_counter <= 6'd0;
                    i_idx        <= 3'd0;
                    j_base       <= 2'd0;
                    out_idx      <= 2'd0;
                    acc0         <= 24'sd0;
                    acc1         <= 24'sd0;
                    if (start) state <= S_LOAD;
                end
```

```
S_LOAD: begin
    if (data_valid) begin
        if (load_counter < 6'd32)
            W_mem[load_counter[4:0]] <=
                serial_data_in;
        else
            X_mem[load_counter - 6'd32] <=
                serial_data_in;

        if (load_counter == 6'd39) begin
            j_base <= 2'd0;
            state  <= S_CLEAR;
        end else begin
            load_counter <= load_counter + 6'd1;
        end
    end
end

S_CLEAR: begin
    acc0  <= 24'sd0;
    acc1  <= 24'sd0;
    i_idx <= 3'd0;
    state <= S_MAC;
end

S_MAC: begin
    acc0 <= acc0 + p0_ext;
    acc1 <= acc1 + p1_ext;

    if (i_idx == 3'd7)
        state <= S_STORE_PAIR;
    else
        i_idx <= i_idx + 3'd1;
end

S_STORE_PAIR: begin
    if (j_base == 2'd0) begin
        Y_mem[0] <= acc_to_q16(acc0);
        Y_mem[1] <= acc_to_q16(acc1);
        j_base   <= 2'd2;
```

```
                        state    <= S_CLEAR;
                    end else begin
                        Y_mem[2] <= acc_to_q16(acc0);
                        Y_mem[3] <= acc_to_q16(acc1);
                        out_idx  <= 2'd0;
                        state    <= S_STREAM_OUT;
                    end
                end

                S_STREAM_OUT: begin
                    Y_out        <= Y_mem[out_idx];
                    output_valid <= 1'b1;

                    if (out_idx == 2'd3)
                        state <= S_DONE;
                    else
                        out_idx <= out_idx + 2'd1;
                end

                S_DONE: begin
                    done  <= 1'b1;
                    state <= S_IDLE;
                end

                default: state <= S_IDLE;
            endcase
        end
    end

endmodule
```

B Appendix B: Cocotb Testbenches

B.1 Single-MAC Cocotb Testbench

Listing 3: Cocotb-based functional verification for Single-MAC accelerator

```
import random
import cocotb
from cocotb.clock import Clock
from cocotb.triggers import RisingEdge

```

```
def wrap_s8(x: int) -> int:
    x &= 0xFF
    return x - 256 if x >= 128 else x


def model_mac_tile(W_flat, X):

    Ys = []
    for j in range(4):
        acc = 0
        for i in range(8):
            w = int(W_flat[j * 8 + i])
            x = int(X[i])
            acc += (w * x)  # Q2.12 integer
        y_shift = acc >> 6  # arithmetic shift
        Ys.append(wrap_s8(y_shift))
    return Ys


async def reset_dut(dut):
    dut.rst.value = 1
    dut.start.value = 0
    dut.data_valid.value = 0
    dut.serial_data_in.value = 0
    await RisingEdge(dut.clk)
    dut.rst.value = 0
    await RisingEdge(dut.clk)


async def load_serial(dut, W_flat, X):

    stream = list(W_flat) + list(X)
    assert len(stream) == 40

    for v in stream:
        dut.data_valid.value = 1
        dut.serial_data_in.value = int(v)
        await RisingEdge(dut.clk)

    dut.data_valid.value = 0
    dut.serial_data_in.value = 0
    await RisingEdge(dut.clk)


async def start_and_collect_outputs(dut):
    ys = []

    while len(ys) < 4:
        await RisingEdge(dut.clk)
        if int(dut.output_valid.value) == 1:
            # safer with new cocotb versions
            ys.append(int(dut.Y_out.value.to_signed()))

    while True:
        await RisingEdge(dut.clk)
        if int(dut.done.value) == 1:
            break

    return ys


@cocotb.test()
async def mac_accelerator_random_tests(dut):
    Clock(dut.clk, 10, unit="ns").start()
    await reset_dut(dut)

    NUM_TESTS = 200
    PRINT_FIRST_N = 5

    directed = []
    directed.append(([0]*32, [0]*8))
    directed.append(([127]*32, [127]*8))
    directed.append(([-128]*32, [-128]*8))
    directed.append(([64 if (k % 2 == 0) else -64 for k in range(32)
       ],
                     [32, -32, 16, -16, 8, -8, 4, -4]))

    tests = directed[:]
    for _ in range(NUM_TESTS):
        W = [wrap_s8(random.randint(-128, 127)) for _ in range(32)]
        X = [wrap_s8(random.randint(-128, 127)) for _ in range(8)]
        tests.append((W, X))

    for t_idx, (W_flat, X) in enumerate(tests):
        exp = model_mac_tile(W_flat, X)

        dut.start.value = 1
        await RisingEdge(dut.clk)
        dut.start.value = 0

        await load_serial(dut, W_flat, X)
        got = await start_and_collect_outputs(dut)

        if t_idx < PRINT_FIRST_N:
            dut._log.info(f"TEST␣#{t_idx}")
            dut._log.info(f"X␣=␣{X}")
            dut._log.info(f"W[0:8]␣␣␣(j0)␣=␣{W_flat[0:8]}")
            dut._log.info(f"W[8:16]␣␣(j1)␣=␣{W_flat[8:16]}")
            dut._log.info(f"W[16:24]␣(j2)␣=␣{W_flat[16:24]}")
            dut._log.info(f"W[24:32]␣(j3)␣=␣{W_flat[24:32]}")
            dut._log.info(f"Expected␣Y␣=␣{exp}")
            dut._log.info(f"Got␣␣␣␣␣␣Y␣=␣{got}")

        assert got == exp, (
            f"Mismatch␣at␣test␣{t_idx}\n"
            f"Expected={exp}\nGot={got}\nW={W_flat}\nX={X}"
        )
```

B.2 Dual-MAC Cocotb Testbench

Listing 4: Cocotb-based functional verification for Dual-MAC accelerator

```python
import random
import cocotb
from cocotb.clock import Clock
from cocotb.triggers import RisingEdge


def wrap_s8(x: int) -> int:
    x &= 0xFF
    return x - 256 if x >= 128 else x


def model_mac_tile(W_flat, X):
    Ys = []
    for j in range(4):
        acc = 0
        for i in range(8):
            w = int(W_flat[j * 8 + i])
            x = int(X[i])
            acc += (w * x)
        y_shift = acc >> 6
        Ys.append(wrap_s8(y_shift))
    return Ys


async def reset_dut(dut):
    dut.rst.value = 1
    dut.start.value = 0
    dut.data_valid.value = 0
    dut.serial_data_in.value = 0
    await RisingEdge(dut.clk)
    dut.rst.value = 0
    await RisingEdge(dut.clk)


async def load_serial(dut, W_flat, X):
    stream = list(W_flat) + list(X)
    assert len(stream) == 40
    for v in stream:
        dut.data_valid.value = 1
        dut.serial_data_in.value = int(v)
        await RisingEdge(dut.clk)

    dut.data_valid.value = 0
    dut.serial_data_in.value = 0
    await RisingEdge(dut.clk)


async def start_and_collect_outputs(dut):
    ys = []

    while len(ys) < 4:
        await RisingEdge(dut.clk)
        if int(dut.output_valid.value) == 1:
            ys.append(int(dut.Y_out.value.to_signed()))

    while True:
        await RisingEdge(dut.clk)
        if int(dut.done.value) == 1:
            break

    return ys


@cocotb.test()
async def mac_accelerator_dual_random_tests(dut):
    Clock(dut.clk, 10, unit="ns").start()
    await reset_dut(dut)

    NUM_TESTS = 200
    PRINT_FIRST_N = 5

    directed = []
    directed.append(([0]*32, [0]*8))
    directed.append(([127]*32, [127]*8))
    directed.append(([-128]*32, [-128]*8))
    directed.append(([64 if (k % 2 == 0) else -64 for k in range(32)
       ],
                      [32, -32, 16, -16, 8, -8, 4, -4]))
    tests = directed[:]
    for _ in range(NUM_TESTS):
        W = [wrap_s8(random.randint(-128, 127)) for _ in range(32)]
        X = [wrap_s8(random.randint(-128, 127)) for _ in range(8)]
        tests.append((W, X))

    for t_idx, (W_flat, X) in enumerate(tests):
        exp = model_mac_tile(W_flat, X)

        dut.start.value = 1
        await RisingEdge(dut.clk)
        dut.start.value = 0

        await load_serial(dut, W_flat, X)
        got = await start_and_collect_outputs(dut)

        if t_idx < PRINT_FIRST_N:
            dut._log.info(f"TEST␣#{t_idx}")
            dut._log.info(f"X␣=␣{X}")
            dut._log.info(f"W[0:8]␣␣␣(j0)␣=␣{W_flat[0:8]}")
            dut._log.info(f"W[8:16]␣␣(j1)␣=␣{W_flat[8:16]}")
            dut._log.info(f"W[16:24]␣(j2)␣=␣{W_flat[16:24]}")
            dut._log.info(f"W[24:32]␣(j3)␣=␣{W_flat[24:32]}")
            dut._log.info(f"Expected␣Y␣=␣{exp}")
            dut._log.info(f"Got␣␣␣␣␣␣Y␣=␣{got}")

        assert got == exp, (
            f"Mismatch␣at␣test␣{t_idx}\n"
            f"Expected={exp}\nGot={got}\nW={W_flat}\nX={X}"
        )
```

C Appendix C: System Diagrams and RTL Views

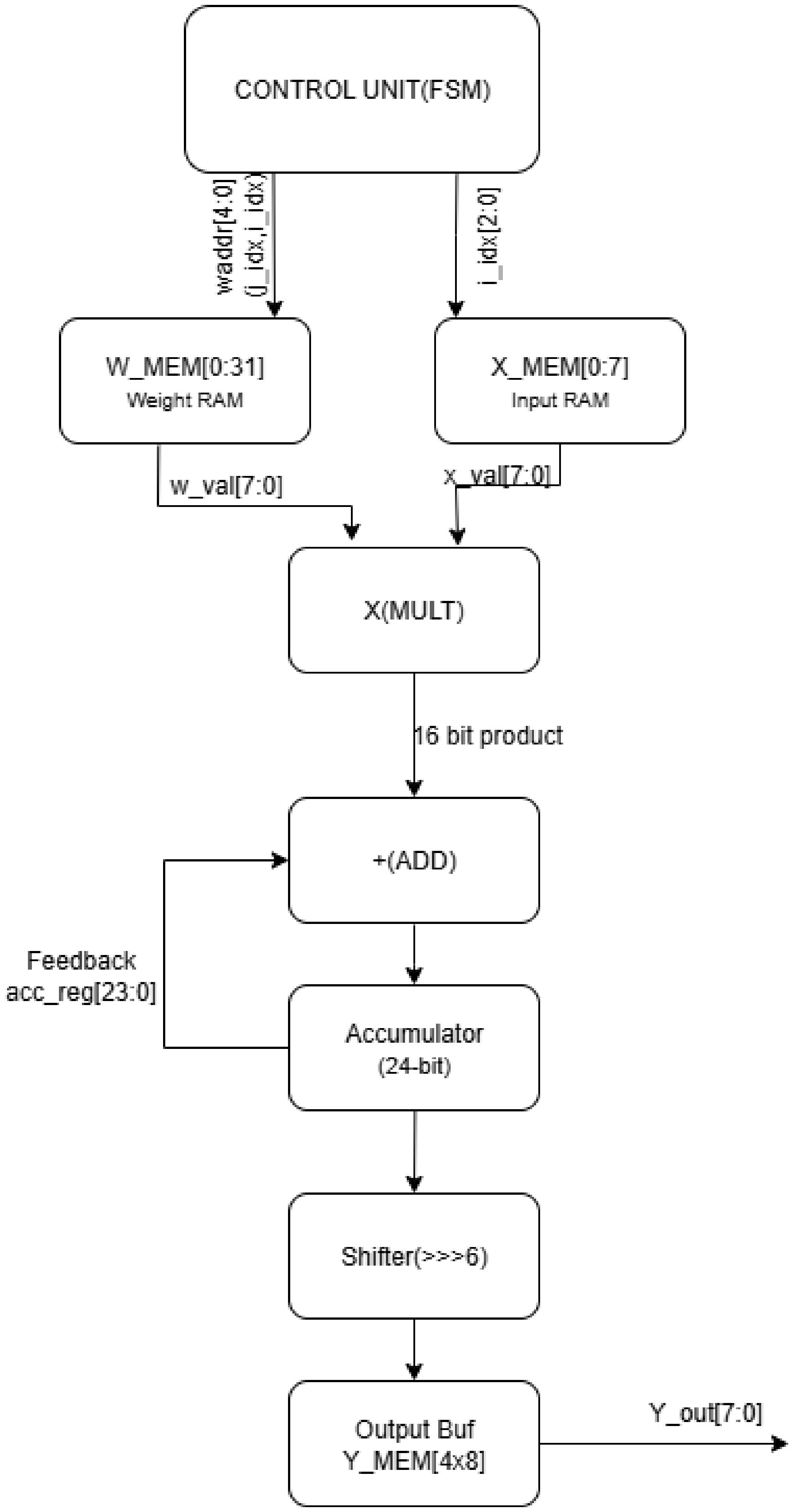


Figure 1: Block-level organization of the Single-MAC Architecture.

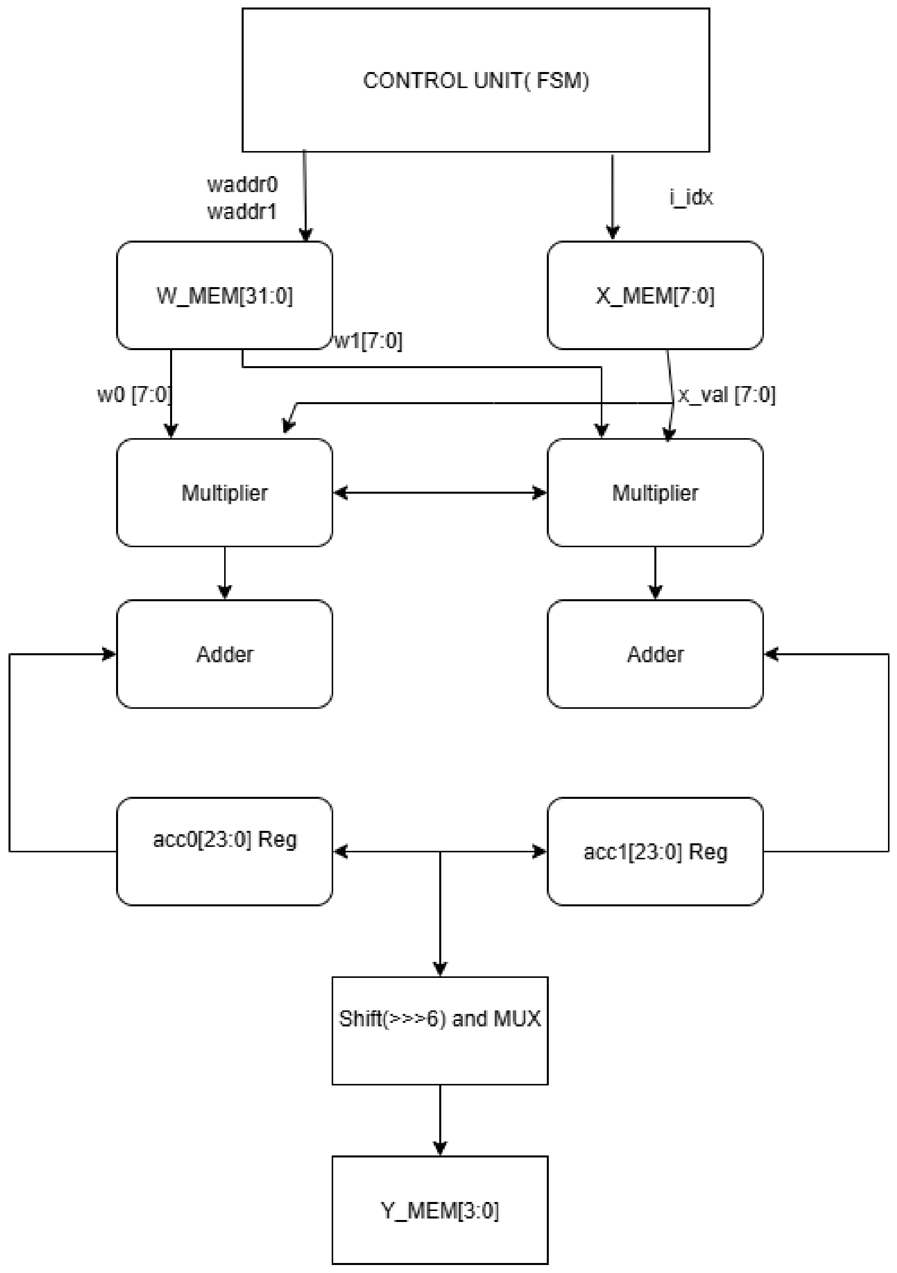


Figure 2: Block-level organization of the Dual-MAC Architecture.

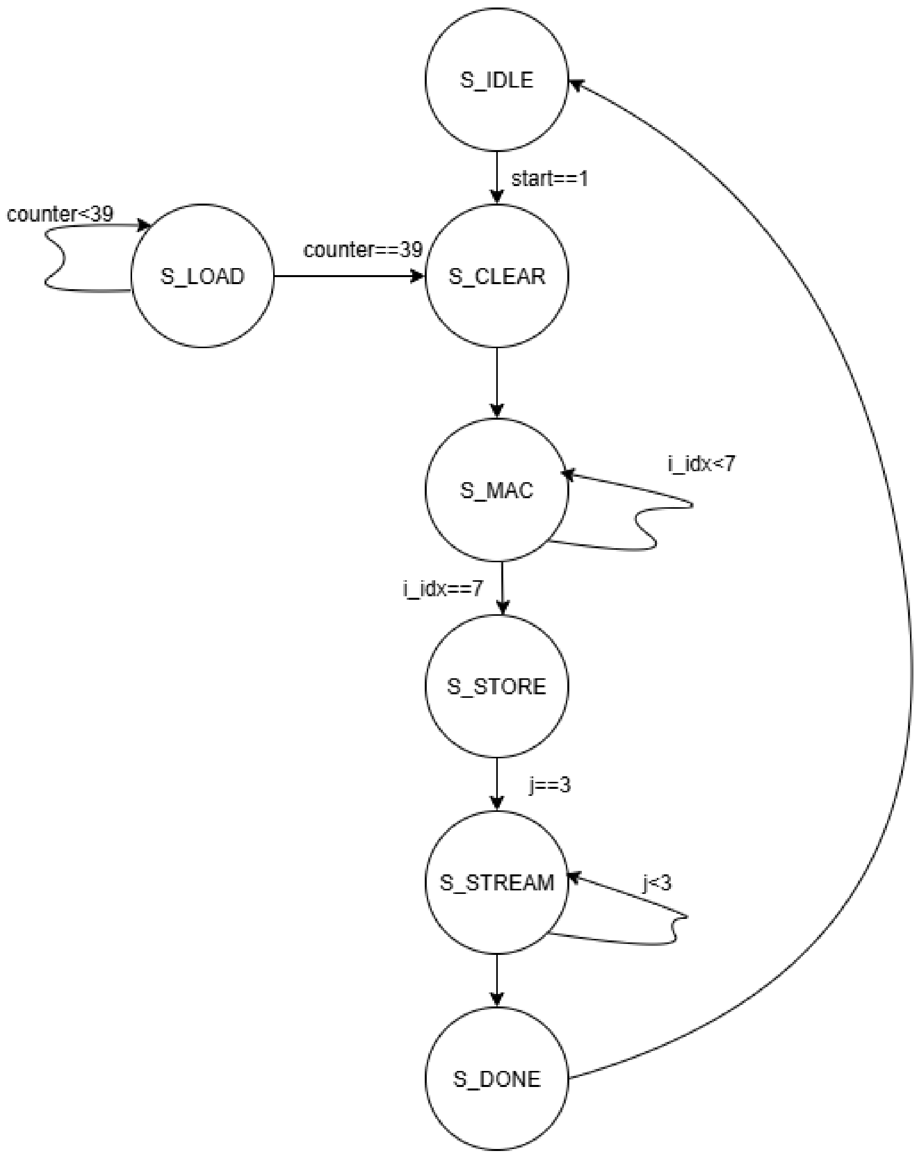


Figure 3: Finite State Machine (FSM) Transition Diagram.

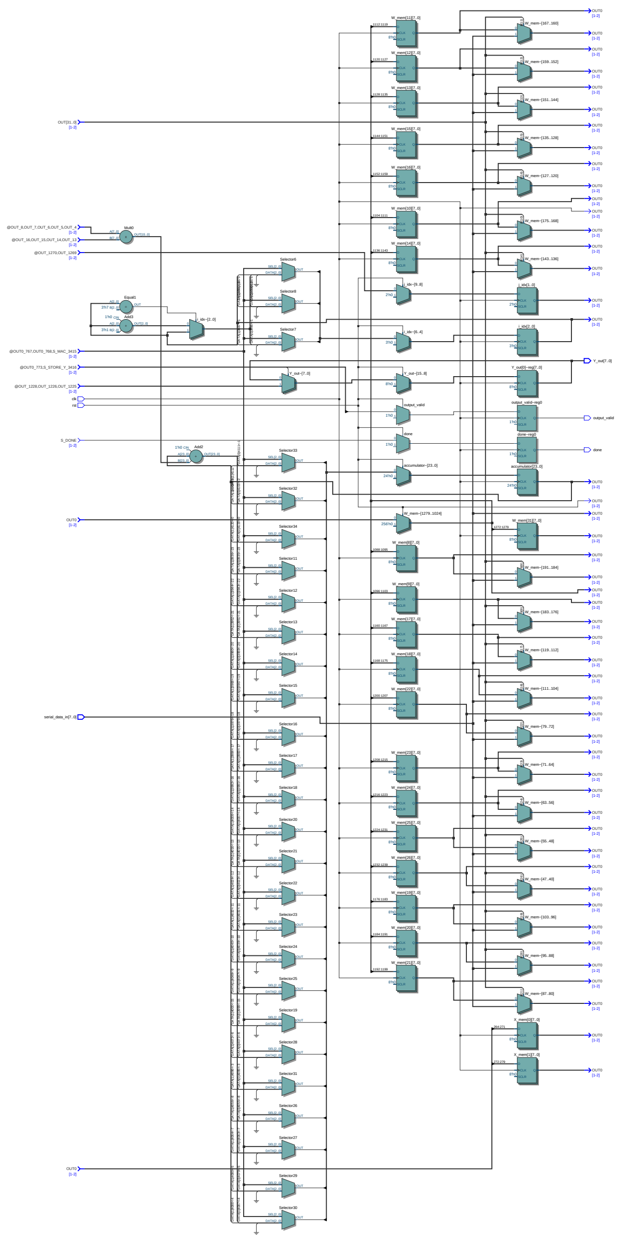


Figure 4: Quartus RTL Viewer - Single-MAC Accelerator (Part 1/2: Overview).

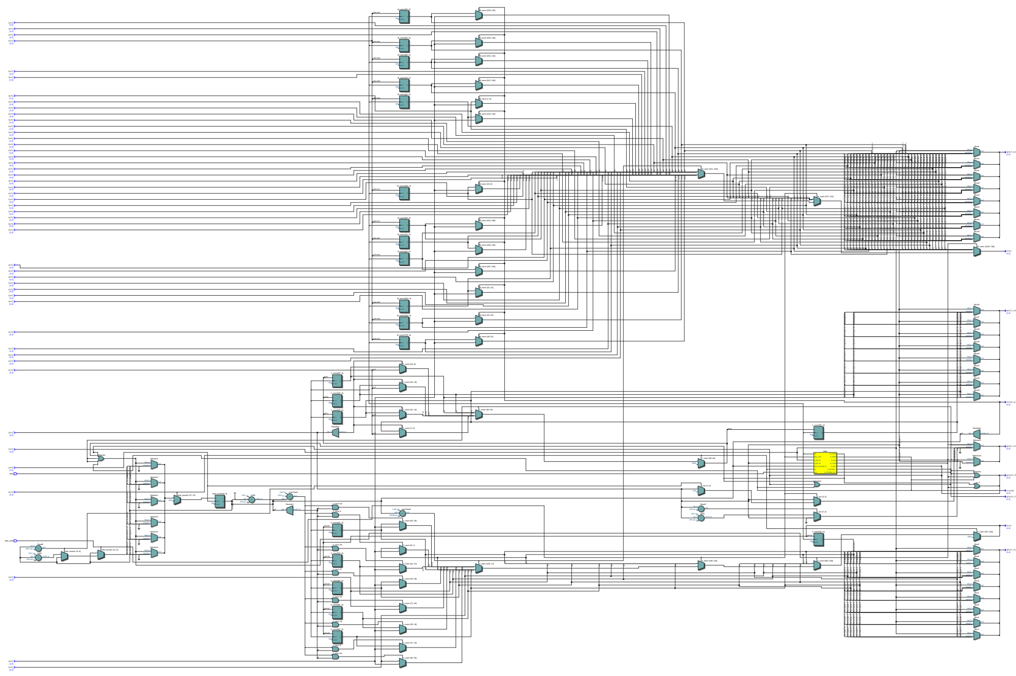


Figure 5: Quartus RTL Viewer - Single-MAC Accelerator (Part 2/2: Detailed Logic).

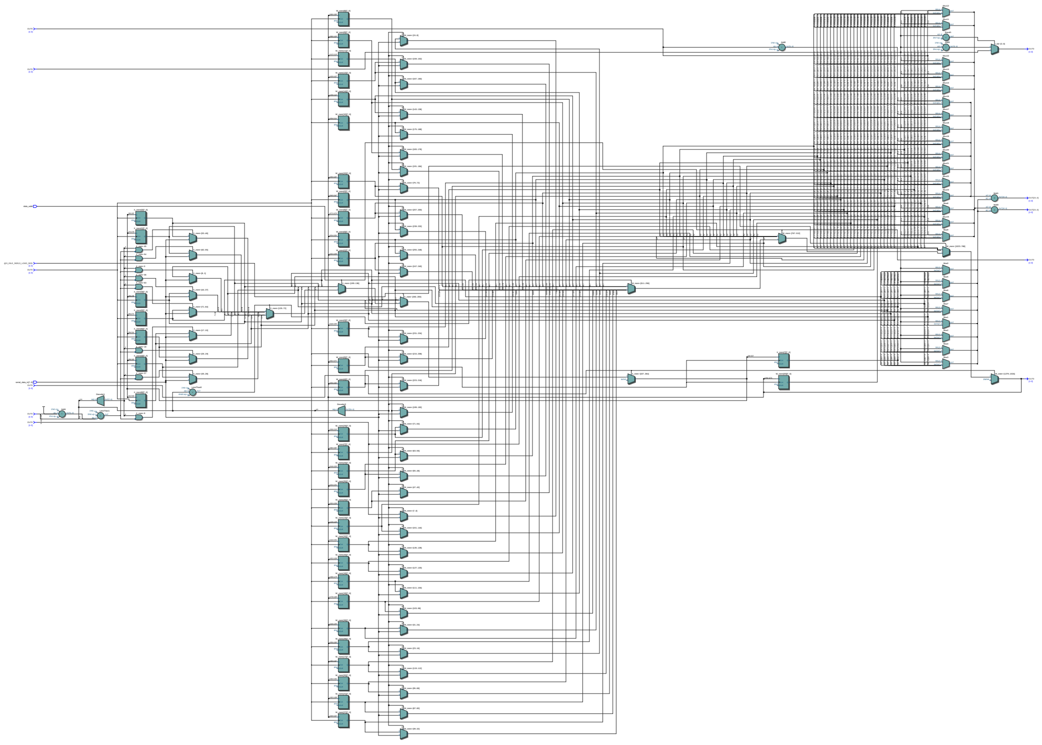


Figure 6: Quartus RTL Viewer - Dual-MAC Accelerator (Part 1/2: Parallel Datapath Overview).

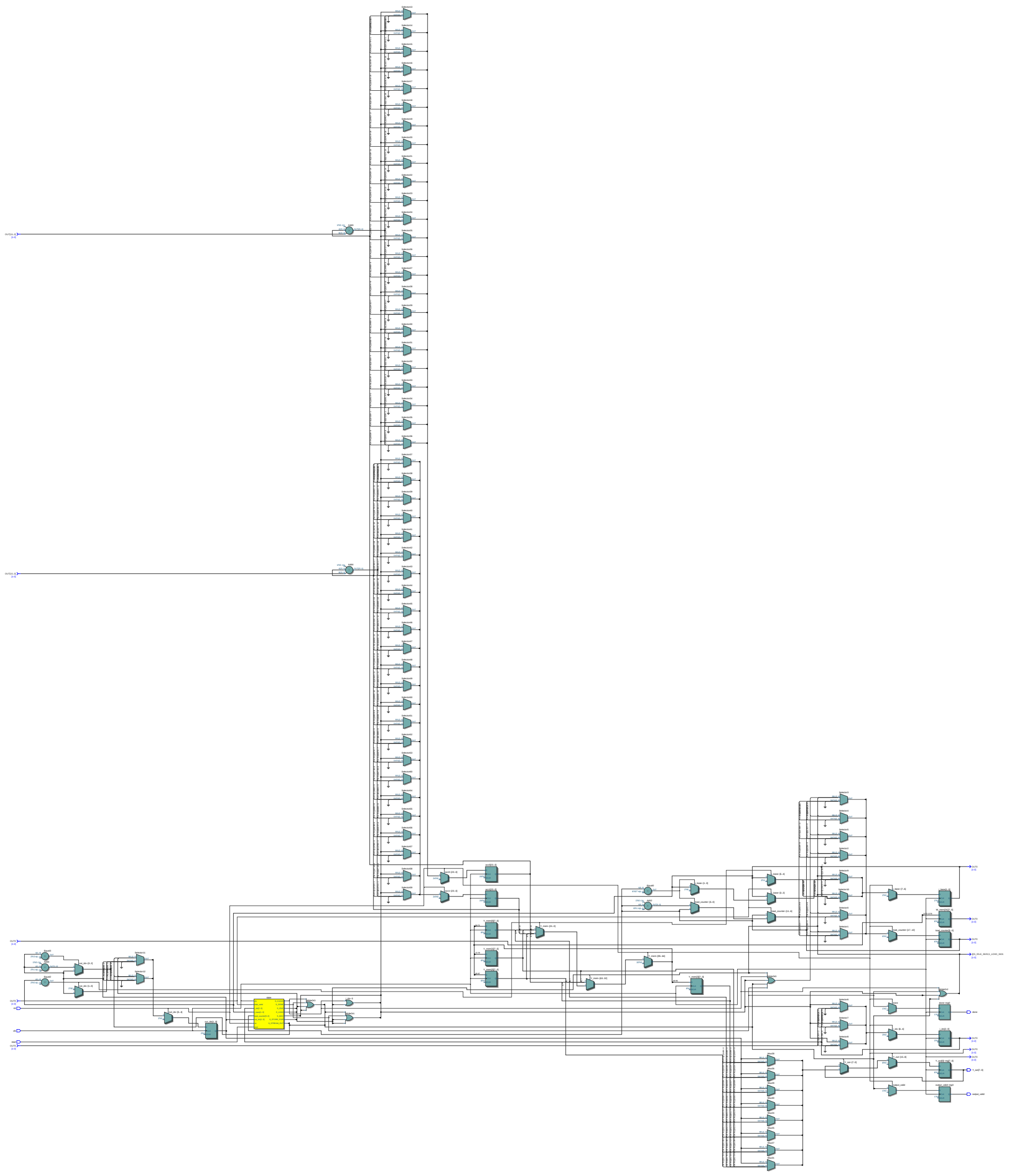
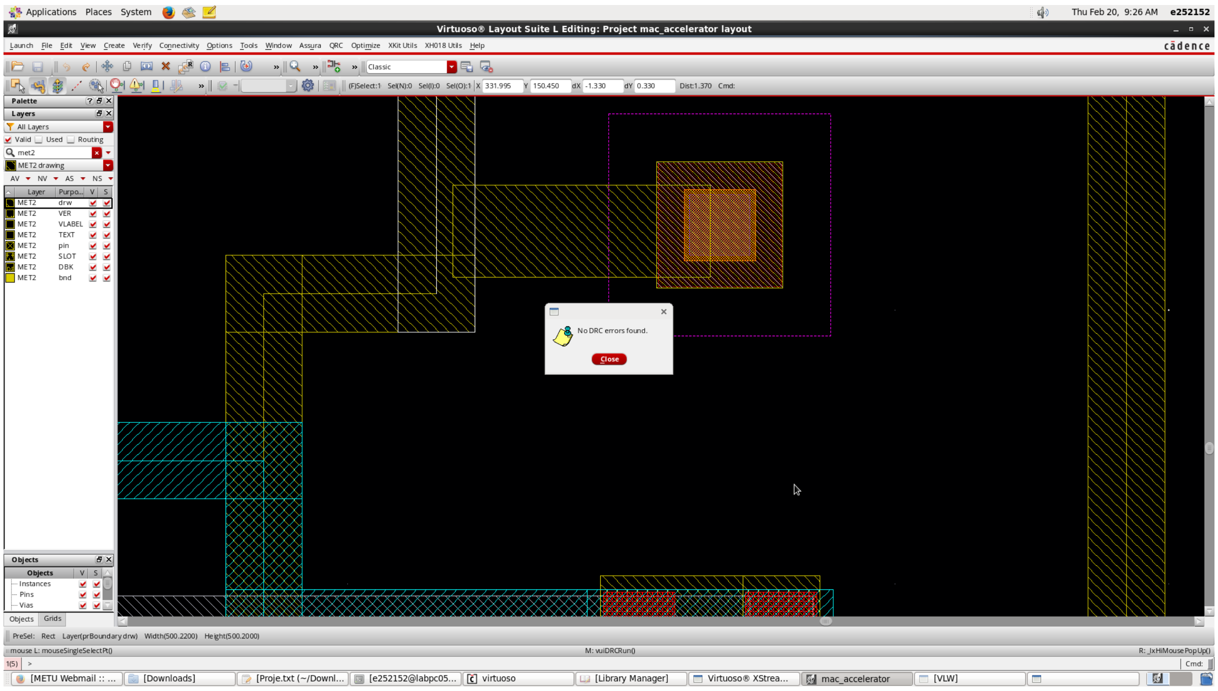


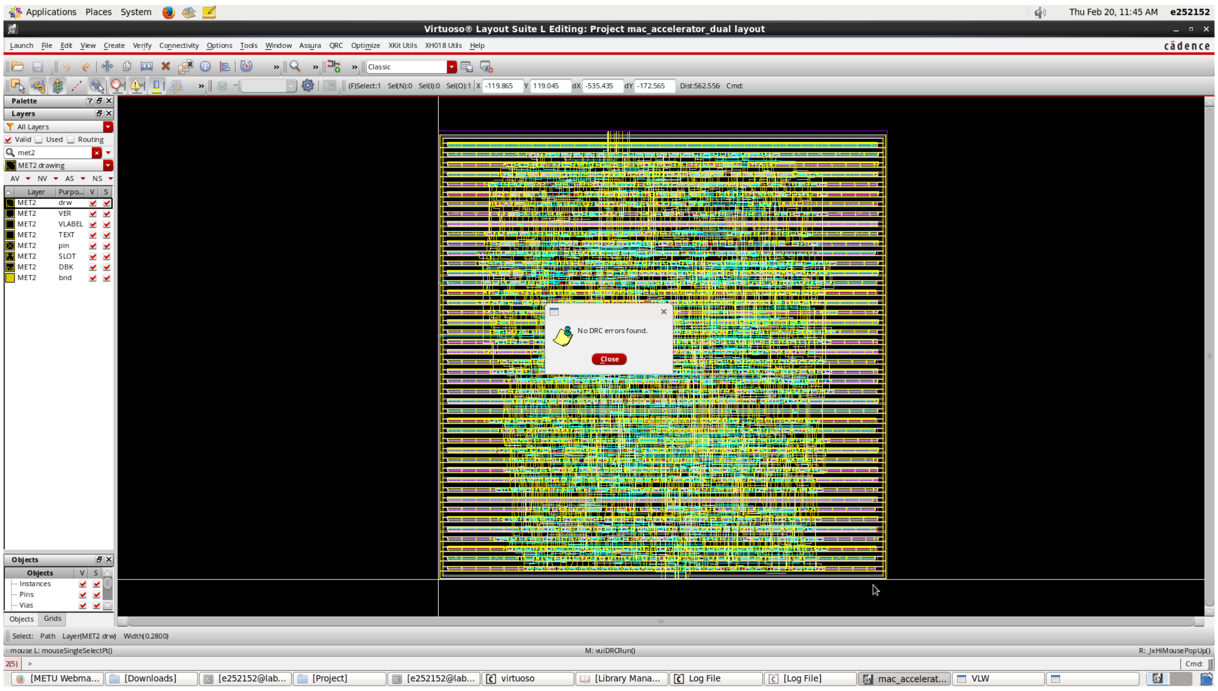
Figure 7: Quartus RTL Viewer - Dual-MAC Accelerator (Part 2/2: Detailed Logic).

D Appendix D: Physical Verification Results


Virtuoso® Layout Suite L Editing: Project mac_accelerator layout
Close

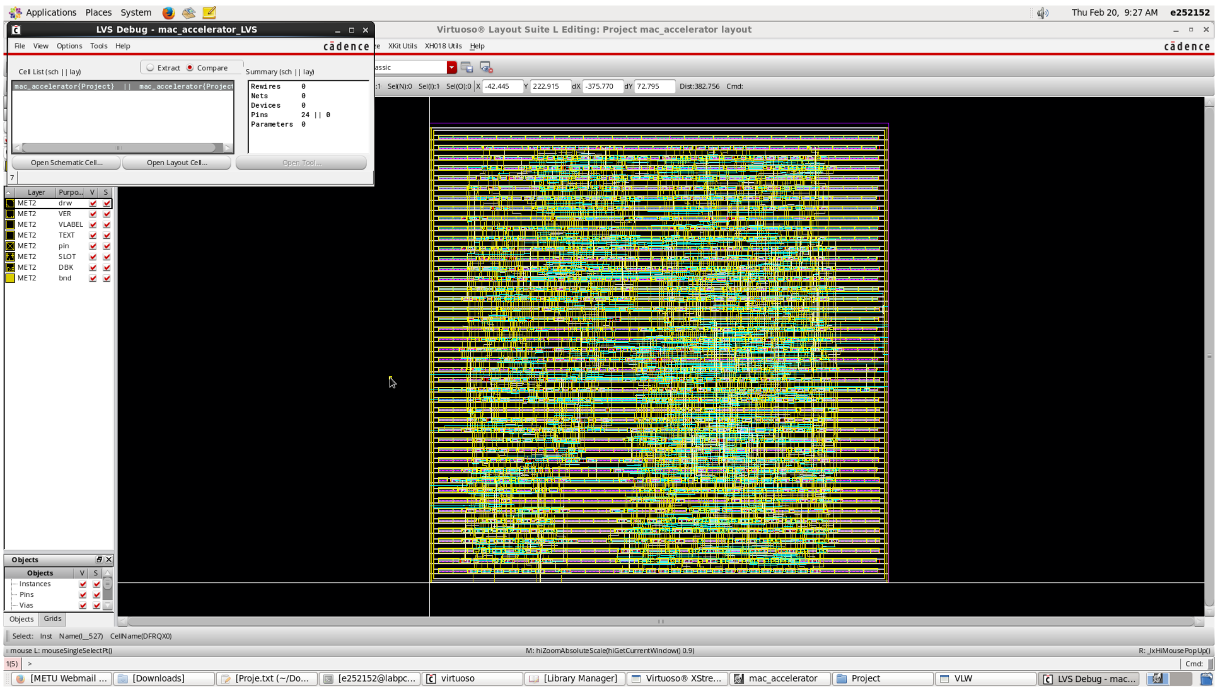
No DRC errors found.

Figure 8: No DRC error for Single-MAC


Virtuoso® Layout Suite L Editing: Project mac_accelerator_dual layout
Close

No DRC errors found.

Figure 9: No DRC error for Dual-MAC


LVS Debug - mac_accelerator_LVS
Virtuoso® Layout Suite L Editing: Project mac_accelerator layout
Extract
Compare
Open Schematic Cell...
Open Layout Cell...
Open Tool...

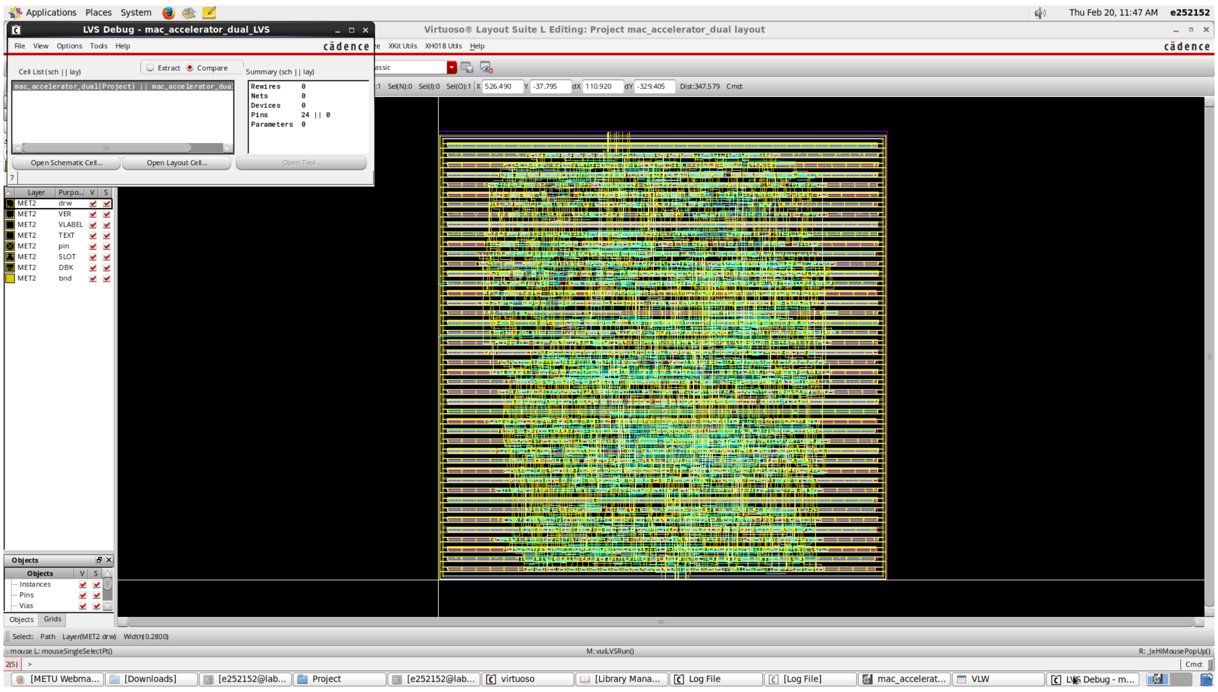
Cell List (sch || lay)

```
mac_accelerator{Project}  ||  mac_accelerator{Project
```

Summary (sch || lay)

```
Rewires     0
Nets        0
Devices     0
Pins        24 || 0
Parameters  0
```

Figure 10: No LVS error (except pins) for Single-MAC


LVS Debug - mac_accelerator_dual_LVS
Virtuoso® Layout Suite L Editing: Project mac_accelerator_dual layout
Extract
Compare
Open Schematic Cell...
Open Layout Cell...
Open Tool...

Cell List (sch || lay)

```
mac_accelerator_dual{Project} || mac_accelerator_dual
```

Summary (sch || lay)

```
Rewires     0
Nets        0
Devices     0
Pins        24 || 0
Parameters  0
```

Figure 11: No LVS error (except pins) for Dual-MAC