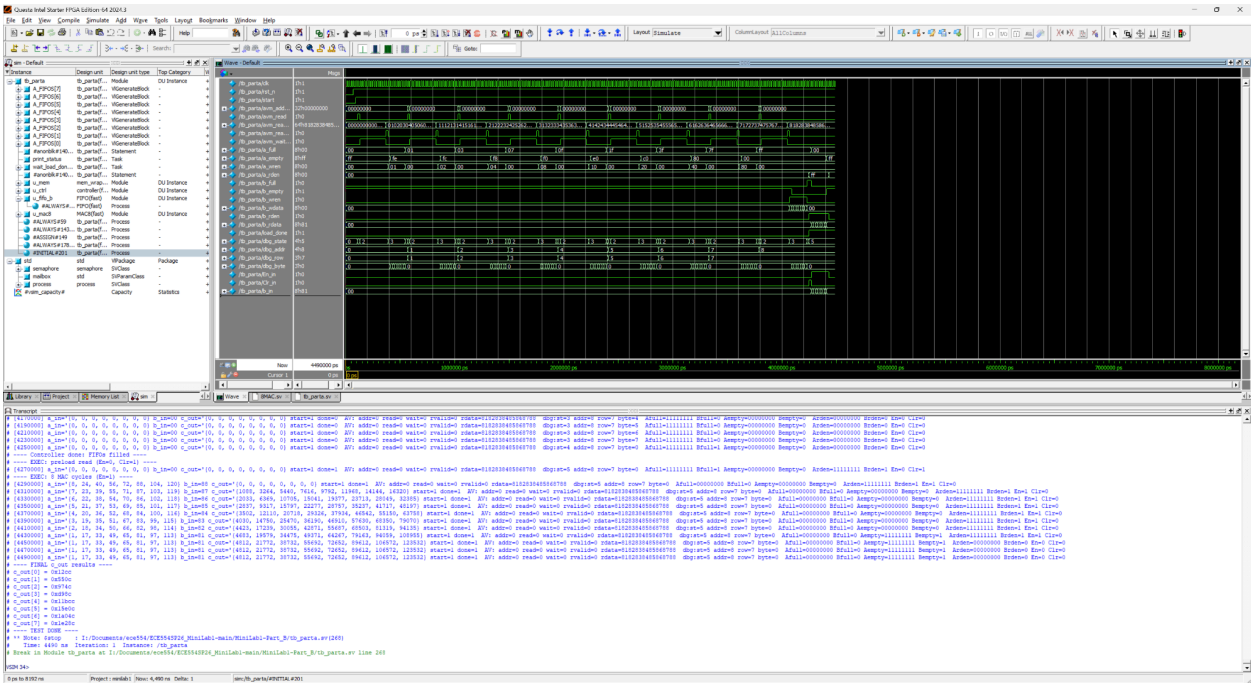
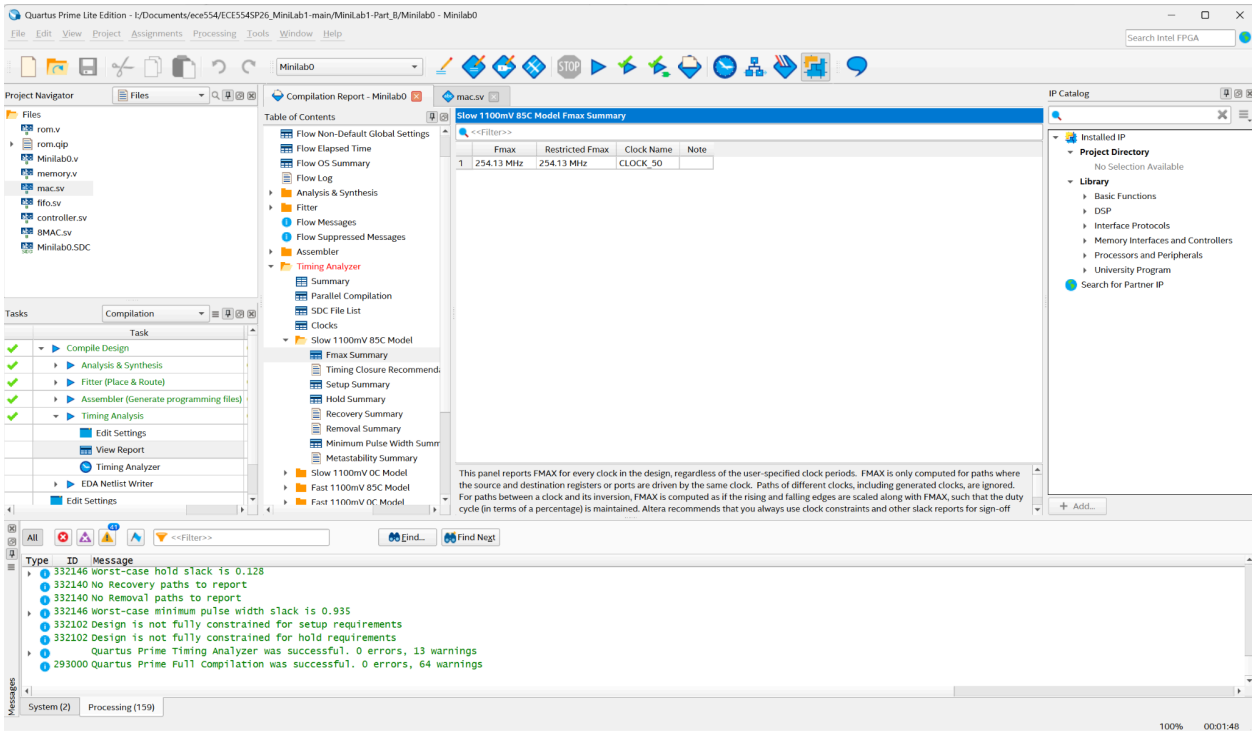
Simulation Log and WaveForm:



The design was tested using a testbench that instantiates the memory wrapper (Avalon-MM slave), controller (Avalon-MM master), FIFO banks, and MAC8 array to verify the complete matrix-vector multiplication datapath. The testbench monitors all critical signals and controller debug states throughout the load and execution phases. Testing proceeds in two phases: first waiting for the controller to fill all FIFOs with data from memory via Avalon-MM reads (with timeout protection), then executing 8 MAC computation cycles while tracking the propagation of enable signals and observing the final accumulated outputs in the c_out registers. The simulation waveform and logs confirmed that all modules were working correctly and the data flowed properly through the system.

Fixing timing constraints:



To meet the 200 MHz timing requirement, we modified the MAC module to implement a two-stage pipeline that separates the multiplication and accumulation operations. In the original design, the multiplication and addition occurred in a single combinational path before the accumulator register, creating a critical timing path that violated the 5 ns clock period constraint. The pipelined version registers the multiplication result in Stage 1 (along with delayed control signals En_d1 and Clr_d1), then performs the accumulation in Stage 2 using the registered product, effectively breaking the long combinational path into two shorter stages that each meet timing.

Our Top Level Module Design:

Our top level module reads 9 words from memory through the Avalon-MM style interface, where addresses 0–7 correspond to the 8 matrix rows and address 8 corresponds to the vector B. Each word is 64 bits (8 bytes), and one byte is loaded per cycle, with the most significant byte loaded first. Each matrix row is written into its own A_FIFO[row], while the B_FIFO holds only the vector values. After the B FIFO has been loaded, the MAC8 array begins execution by streaming values out of the B FIFO and popping values from each A FIFO lane when that lane's enable signal arrives. Once the pipeline finishes and all values have propagated through, the results are held and displayed on the board depending on the switch settings.

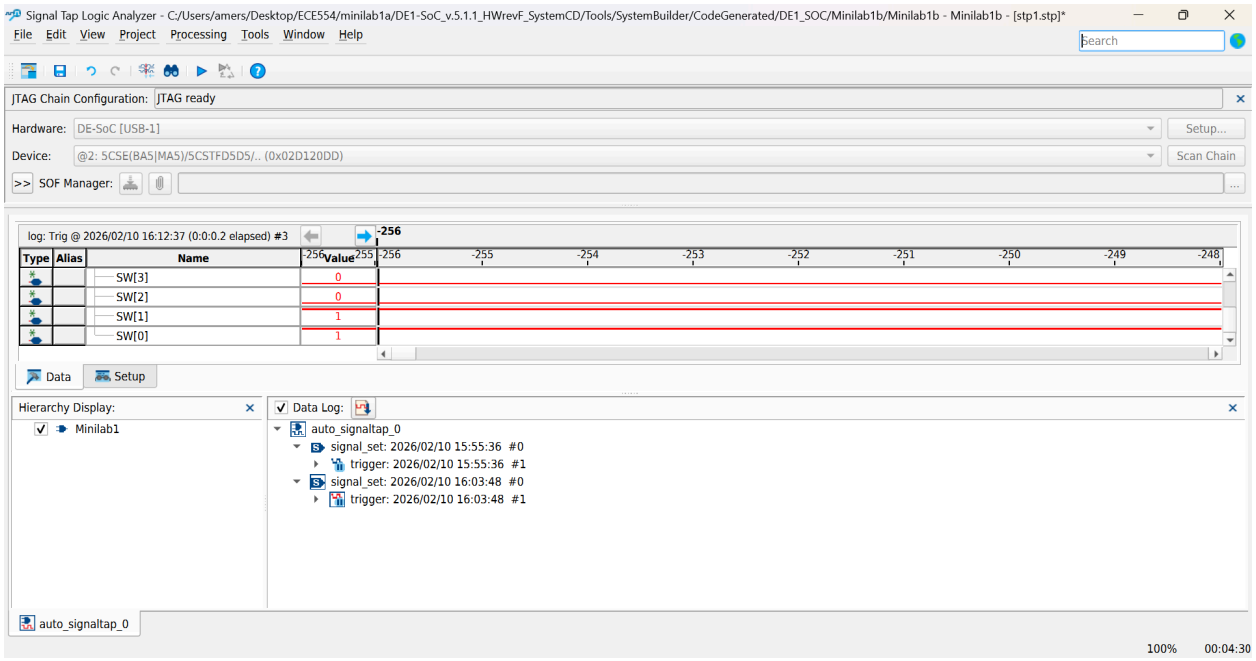
The switches control the display and determine which lane is shown. SW[0] acts as the display enable, and SW[3:1] selects which MAC lane (0–7) is displayed. We also mapped the LEDs to the state machine so that we can easily observe system behavior. LEDR[2:0] displays the current state (memory request, memory wait, unpacking, executing, drain, and done). LEDR[9] indicates when all FIFOs are full, and LEDR[8] indicates when the B FIFO is empty (typically after execution finishes). LEDR[7] was added as a simple debug signal to show when the pipeline is actively processing data.

Our state machine operates using six main states: S_MEM_REQ, S_MEM_WAIT, S_UNPACK, S_EXEC, S_DRAIN, and S_DONE. The first state issues a single-cycle read request by asserting **avm_read**, which starts the memory access. Because memory has latency, the FSM transitions to a waiting state until **readdatavalid** is asserted. Once the data is valid, the design enters the unpack state, where the 64-bit word is written into the appropriate FIFO over 8 cycles. Words 0–7 are written into their corresponding A FIFO, and the final word (address 8) is written into the B FIFO. After the vector has been loaded, execution begins. During execution, one B byte is popped per cycle while each A FIFO lane is read when its pipelined enable signal becomes active. When the B FIFO becomes empty, the FSM transitions to the drain state to allow the pipeline to fully flush and ensure all results settle. Finally, the system enters the done state, where results are held for display, and the only way to restart the process is by asserting reset.

That concludes the details of our top-level module, how our state machine runs in a sequential manner, what each switch/button does, and what the LEDs mean as the MAC is running. Currently, our top level has a timing error that we are unsure how to solve. The lanes are displaying their incorrect values properly, but our execution stage seems to be one byte off. In order to complete the lab, we have decided to move on in case we are not able to debug on time and we will explain our issues during our demo.

Using Signal Tap for Debugging:

In order to test our top level and track our signals, we kept track of our state machine signals and to track when they go high or low. After programming to the board, we were able to see our data and our accumulator stages. Here we show the switches responsively responding to the board input via auto-run analysis.



Difficulties Faced During This Lab:

Amer: Timing was the biggest challenge in this lab. To meet the 200 MHz constraint, we pipelined the multiplication and addition to occur in parallel, but this made the design more cycle-sensitive — data arriving even one cycle early or late would produce incorrect results. Coordinating this with memory latency and the top-level state machine added further complexity. These issues taught us to think more carefully about pipeline-friendly state machine design and writing better testbenches for future labs.

Anirudh: We ran into an issue where the MAC outputs were slightly off from expected values, which turned out to be a one-cycle timing mismatch between the FIFO and the FSM. We fixed it by adding an extra enable cycle before transitioning states. SignalTap also had a learning curve — figuring out the right trigger conditions and signals to probe took some experimentation.

Chance: We had trouble getting the 8MAC module to work correctly at first — the pipelined enable and clear signals weren't propagating properly across the MAC array, which caused incorrect accumulation results. After reviewing the shift register logic and testing with the testbench, we were able to fix the pipeline staging and get all 8 lanes producing correct outputs.

Munasib: Meeting the 200 MHz timing constraint in Part 1b was challenging, as the initial design had setup violations flagged by the Timing Analyzer. We had to review the critical paths

and restructure parts of the logic to reduce delay. Learning to read the timing reports and use the SDC file to set the correct clock period was a useful takeaway from the process.