3.3.1: MAC Integration

a. Describe the module's functionality, including how to control the accumulator state. Would you ever need to put a non-zero value in? Why?

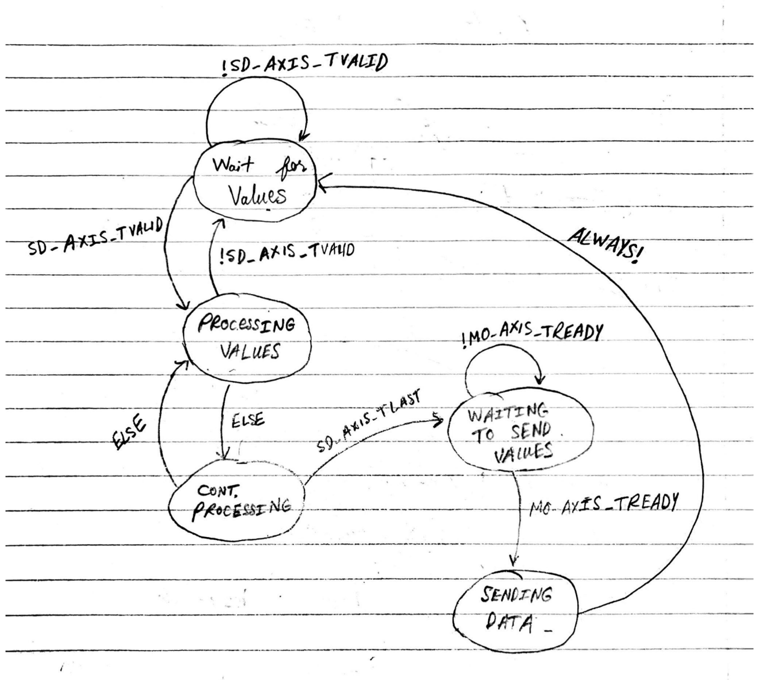


Figure 1. Moore-Type State Diagram for MAC.

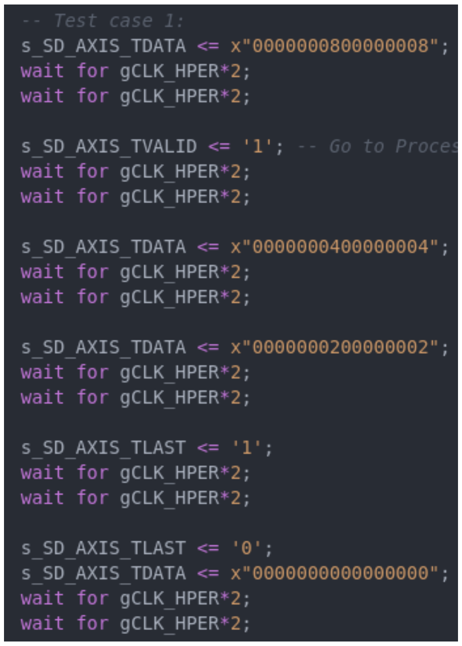
Instead of using a non-zero value, we created an additional “CONT_PROCESSING” state to finish processing values. Additionally, we set input data values (named ***A*** and ***B***) in the “WAIT_FOR_VALUES” state, perform the element-wise multiplication operation (called ***MULT***) in the “PROCESSING_VALUES” state, and set the accumulated value (put into a signal called ***mult_add*** which is then piped into a register to store the accumulated value for the next clock cycle) in the “CONT_PROCESSING” state to deal with signal propagation delays in process statements.

b. Identify a few sample inputs and corresponding outputs that you expect for this module.

The thought process for each of these test cases is to load in a set of N different A and B values (inputs into the MAC unit) which should perform the following operation (and then set ***MO_AXIS_TDATA*** as the output):

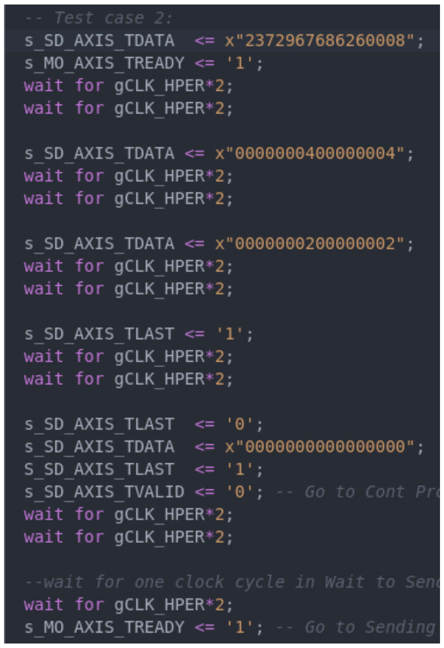
$$\mathbf{A_0B_0 + A_1B_1 + A_2B_2 + \ldots + A_NB_N}$$

Test cases as implemented in VHDL are shown directly below, but a more in-depth explanation can be found later in the document when the waveforms are analyzed.



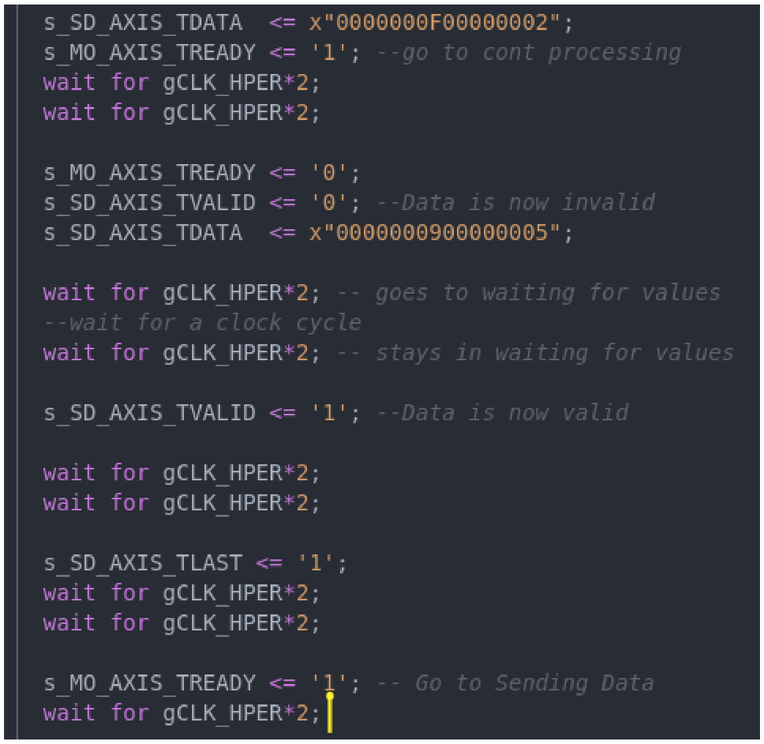
```
-- Test case 1:
s_SD_AXIS_TDATA <= x"0000000800000008";
wait for gCLK_HPER*2;
wait for gCLK_HPER*2;

s_SD_AXIS_TVALID <= '1'; -- Go to Proces
wait for gCLK_HPER*2;
wait for gCLK_HPER*2;

s_SD_AXIS_TDATA <= x"0000000400000004";
wait for gCLK_HPER*2;
wait for gCLK_HPER*2;

s_SD_AXIS_TDATA <= x"0000000200000002";
wait for gCLK_HPER*2;
wait for gCLK_HPER*2;

s_SD_AXIS_TLAST <= '1';
wait for gCLK_HPER*2;
wait for gCLK_HPER*2;

s_SD_AXIS_TLAST <= '0';
s_SD_AXIS_TDATA <= x"0000000000000000";
wait for gCLK_HPER*2;
wait for gCLK_HPER*2;
```

Figure 1a. Test Case 1.



```
-- Test case 2:
s_SD_AXIS_TDATA  <= x"2372967686260008";
s_MO_AXIS_TREADY <= '1';
wait for gCLK_HPER*2;
wait for gCLK_HPER*2;

s_SD_AXIS_TDATA <= x"0000000400000004";
wait for gCLK_HPER*2;
wait for gCLK_HPER*2;

s_SD_AXIS_TDATA <= x"0000000200000002";
wait for gCLK_HPER*2;
wait for gCLK_HPER*2;

s_SD_AXIS_TLAST <= '1';
wait for gCLK_HPER*2;
wait for gCLK_HPER*2;

s_SD_AXIS_TLAST  <= '0';
s_SD_AXIS_TDATA  <= x"0000000000000000";
s_SD_AXIS_TLAST  <= '1';
s_SD_AXIS_TVALID <= '0'; -- Go to Cont Pro
wait for gCLK_HPER*2;
wait for gCLK_HPER*2;

--wait for one clock cycle in Wait to Send
wait for gCLK_HPER*2;
s_MO_AXIS_TREADY <= '1'; -- Go to Sending
```

Figure 1b. Test Case 2.



```
s_SD_AXIS_TDATA  <= x"0000000F00000002";
s_MO_AXIS_TREADY <= '1'; --go to cont processing
wait for gCLK_HPER*2;
wait for gCLK_HPER*2;

s_MO_AXIS_TREADY <= '0';
s_SD_AXIS_TVALID <= '0'; --Data is now invalid
s_SD_AXIS_TDATA  <= x"0000000900000005";

wait for gCLK_HPER*2; -- goes to waiting for values
--wait for a clock cycle
wait for gCLK_HPER*2; -- stays in waiting for values

s_SD_AXIS_TVALID <= '1'; --Data is now valid

wait for gCLK_HPER*2;
wait for gCLK_HPER*2;

s_SD_AXIS_TLAST <= '1';
wait for gCLK_HPER*2;
wait for gCLK_HPER*2;

s_MO_AXIS_TREADY <= '1'; -- Go to Sending Data
wait for gCLK_HPER*2;
```

Figure 1c. Test Case 3.

**c. Define the module interface (i.e., draw the symbol box around the basic design provided by
the textbook).**

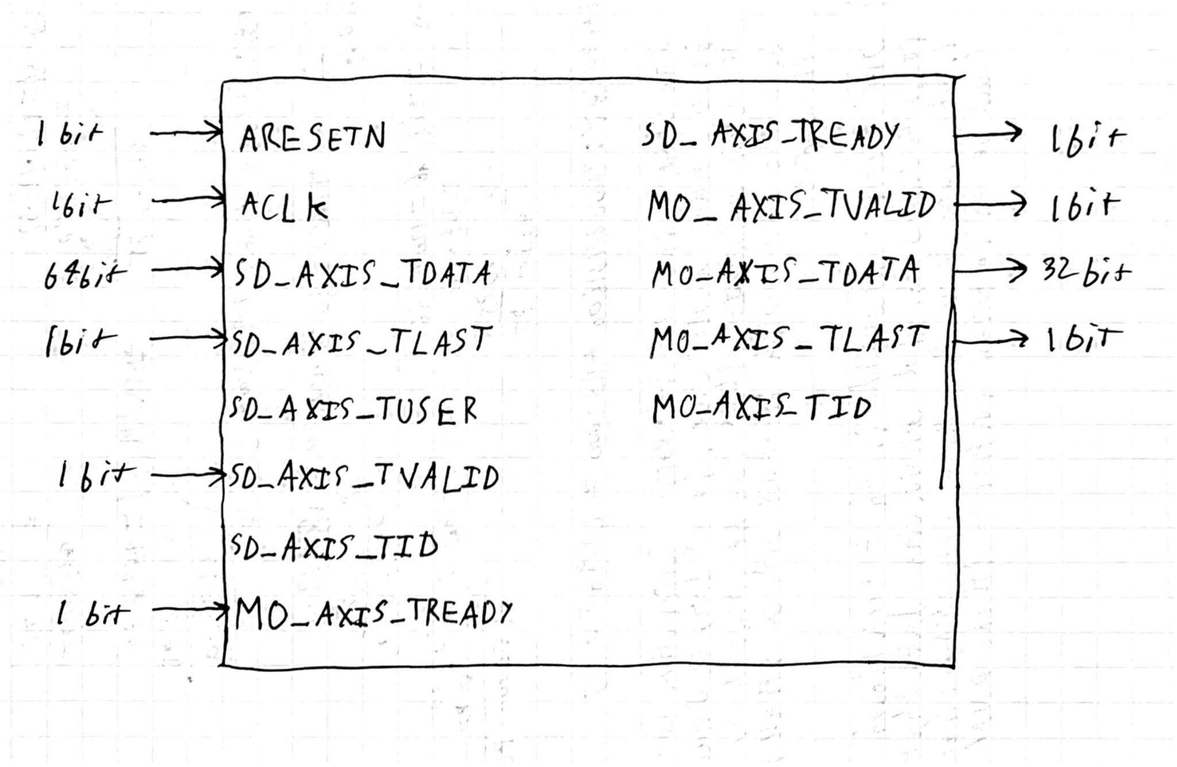


Figure 2. Module Interface Diagram.

**d. Prior to authoring the HDL of your implementation, create a schematic diagram of you design
with signal names, widths, and ports defined.**

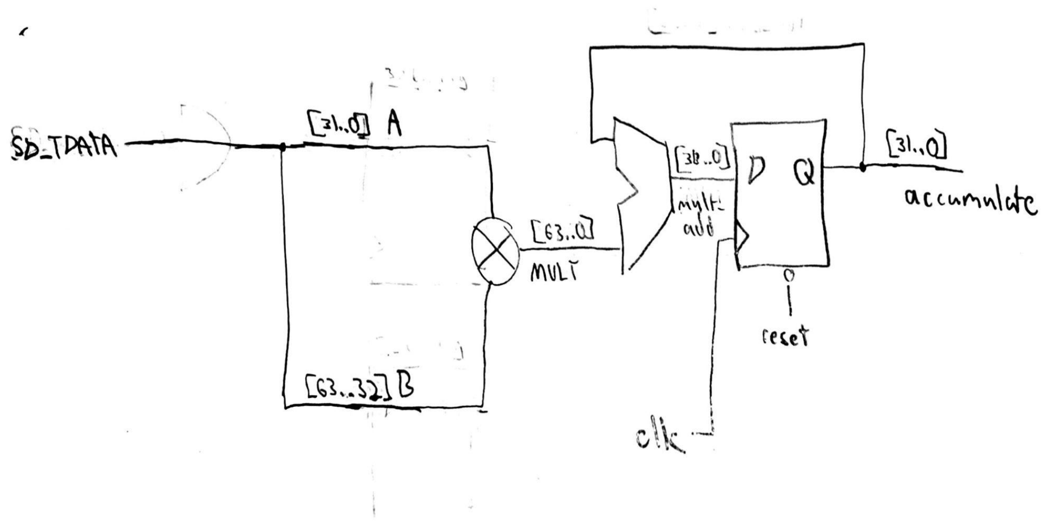


Figure 3. Schematic Diagram of Design.

e. Calculate (and report) all of the required bit-widths you will need based on the specific layer
you implemented. Additionally, describe the appropriate quantization approach for the output
of your accumulator based on Tensorflow Lite's quantization (i.e., 8-bit integers).

Input and weights were both 32 bit fixed points. The multiplied output was 64 bits and the accumulation was 32 bits to avoid overflow.

If we were to implement 8-bit quantization, we would quantize the data immediately after it was read into our system. We would quantize values according to the value's size. In class we learned about creating quantized "levels" that the current values would be estimated to depending on which one they were closest to. This is shown in Figure 4, which is a graph taken from class:

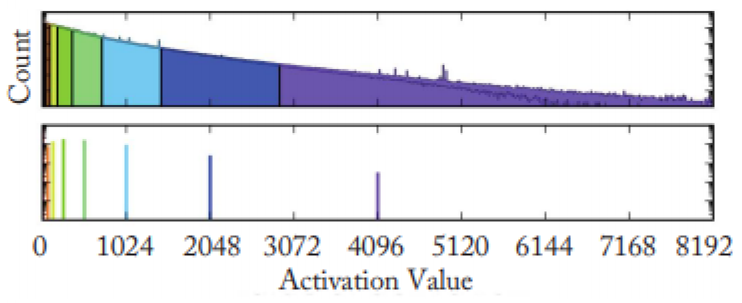


Figure 4. Quantization Levels.

We would have to implement a slt (set less than) function in our system but this would allow us to take each high bit value and estimate which quantized value it is closest to. Lower values should have more quantized levels available for better accuracy.

f. Implement in VHDL using your previously-designed modules. Test the design for edge-case
inputs using whatever approach you prefer (VHDL testbench, do file, etc.), and justify why
your tests were sufficient. Submit your VHDL files and testbench in a folder called Mac.

Test Case 1: Multiply and accumulate multiple values over multiple cycles and then the accumulated value to output. Expect: (8*8)+(4*4)+(2*2)=84

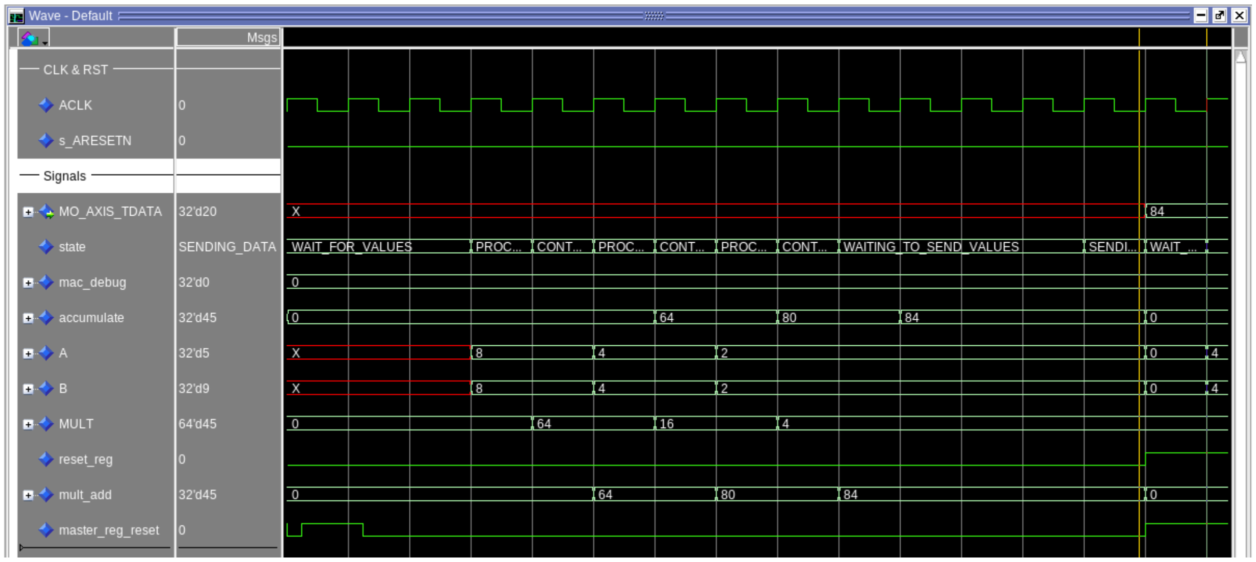


Figure 5a. Test Case 1 Waveform

Test Case 2: For the second test case we wanted to make sure our MAC would go back to our WAIT FOR VALUES state after it sends data. It successfully did this and to be sure the last partial sum accumulated value didn't interfere with the new one we began a new accumulation. For this test case expect: (2250637320 * 594712182) + (4 * 4) + (2 * 2) = 1.33E18 but this value is overflow so our MAC contantenates to 890811332

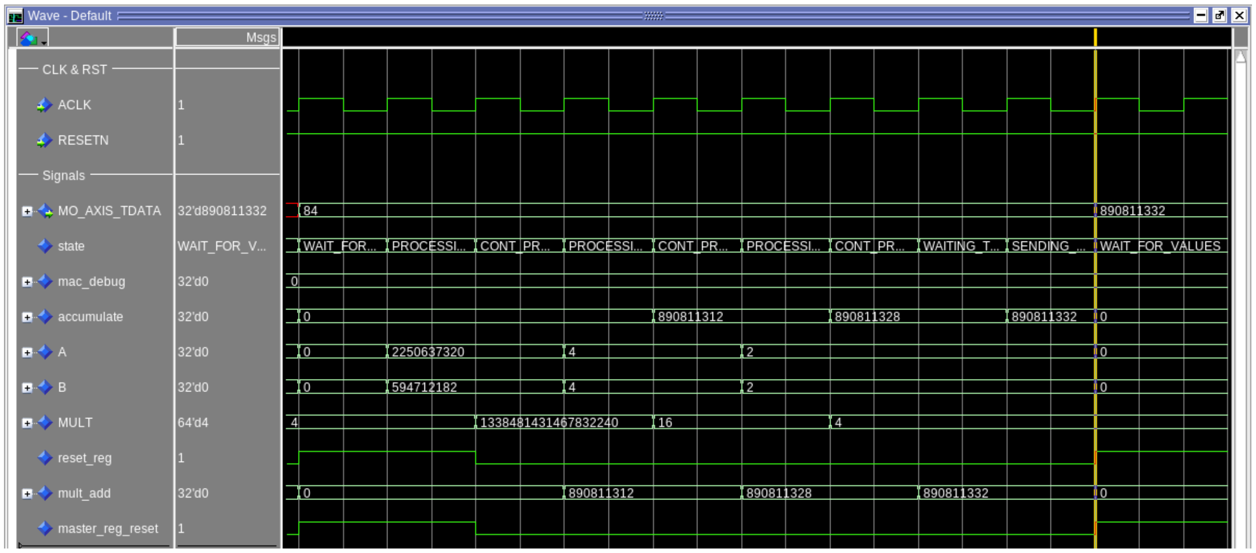


Figure 5a. Test Case 2 Waveform

Test Case 3: For this test case we send in our A & B data values to begin multiplying and accumulating for a new output, but the clock cycle after we input our values we set our VALID signal to invalid. As intended, this “stalls” our MAC so it waits for the VALID signal to become valid again to continue processing. Expect: (5*9)=45 (on mult-add), wait a clock cycle, then see 45 on accumulate. Then we decide to send 45 as an output so wait another 2 clock cycles for MO_AXIS_TDATA to see 45.

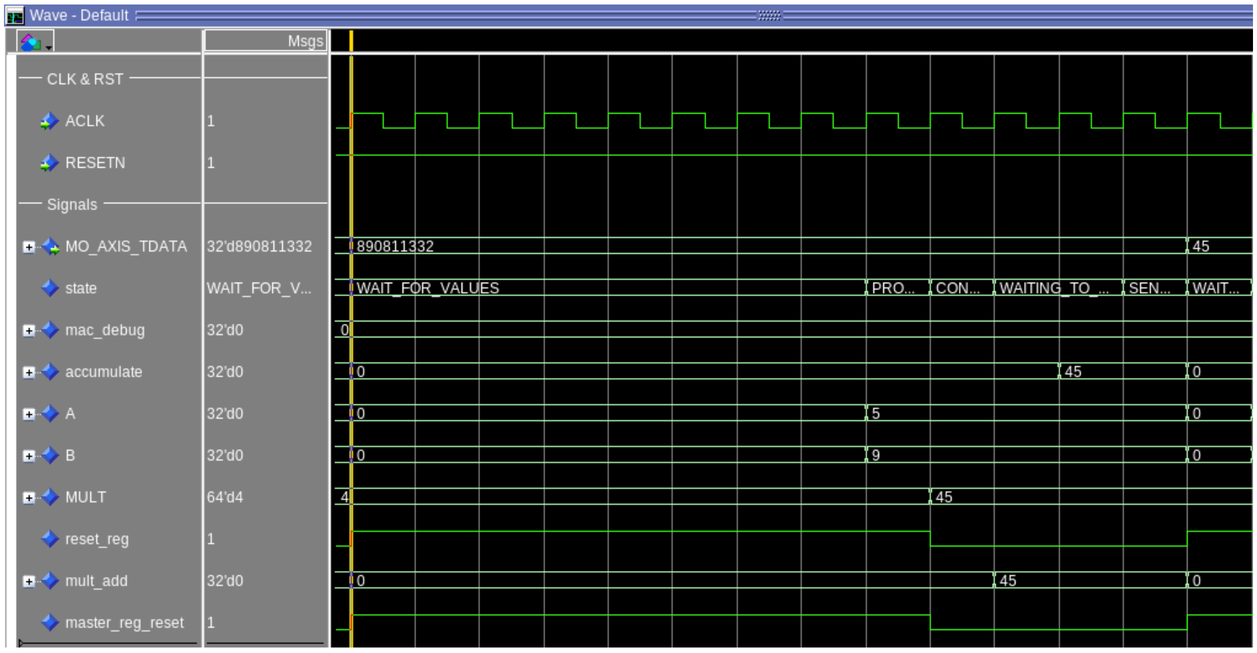


Figure 5c. Test Case 3 Waveform

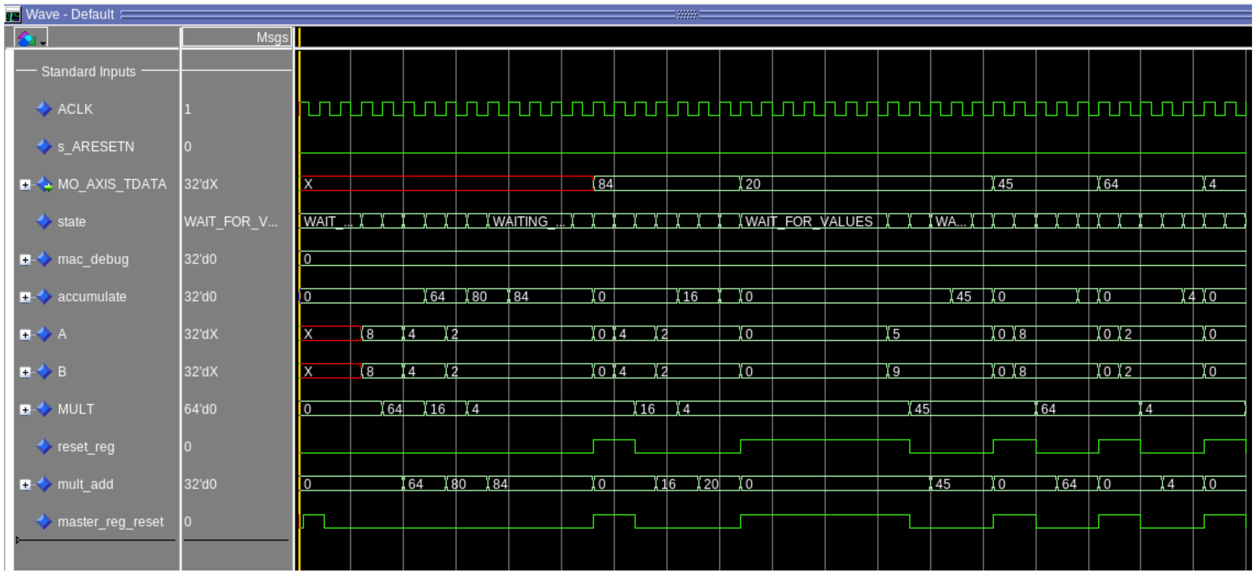


Figure 5d. Waveform For All 3 Test Cases

g. Synthesize your MAC unit and report the maximum frequency, resource utilization, and the
critical path of the design. Separate instructions for this to follow. Calculate the maximum
throughput of your design for a single layer of your choosing. How many Mac operations
occur? How many cycles does each operation take? You may assume that memory and data
movement overheads are negligible and computation is done with a single PE. How would
performance scale with multiple PEs? Is this performance estimate reasonable? Why or why
not?

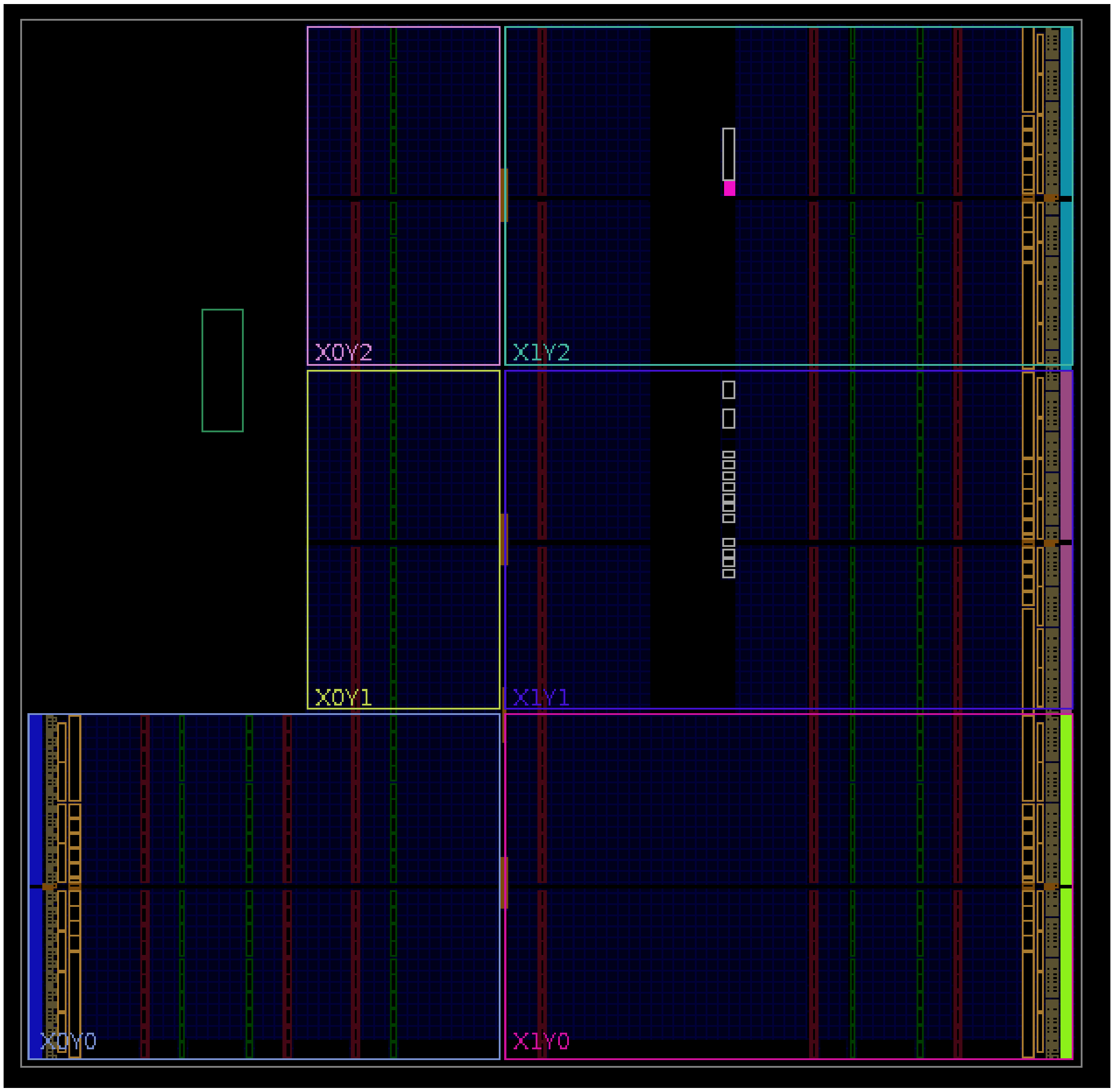
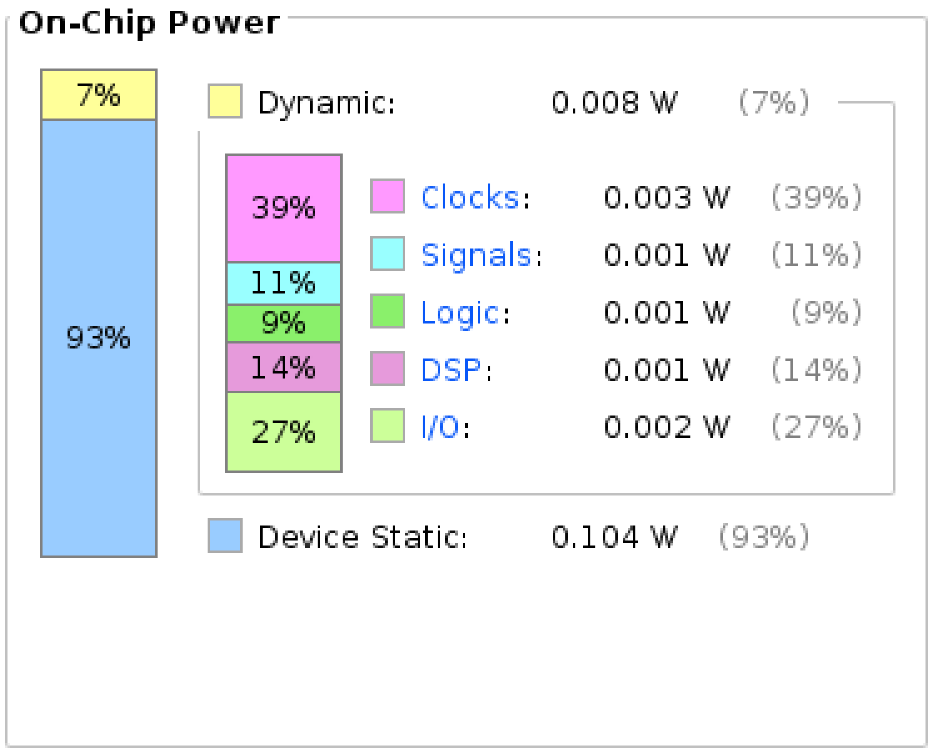


Figure 6. Synthesized MAC unit

Max Clock Freq: 200MHz
Total On-Chip Power: 113mW


On-Chip Power

🟨	Dynamic:	0.008 W	(7%)
🟪	Clocks:	0.003 W	(39%)
🟦	Signals:	0.001 W	(11%)
🟩	Logic:	0.001 W	(9%)
🟪	DSP:	0.001 W	(14%)
🟩	I/O:	0.002 W	(27%)
🟦	Device Static:	0.104 W	(93%)

Figure 7. Resource Utilization

Critical Path: 5 ns

Throughput: If we were to calculate Layer 2 of our DNN using this MAC, we need to calculate how many MACs Layer 2 needs and how many cycles each MAC takes. Layer 2 has a 5x5x32 Filter, so we have to calculate 800 inputs multiplied by 800 weights. Each multiply and accumulate takes 2 cycles, and the rest of our processing takes 3 cycles. So each output element from the MAC takes 1603 (800*2+3) cycles to calculate. We have to calculate 100352 output elements (56*56*32 output size), so the total number of cycles we need to go through to compute this layer is 160,864,256 (100352*1603). Our critical path is 5ns, so if we would only use this MAC to compute the layer, it would take 804ms.

804ms for Layer 2

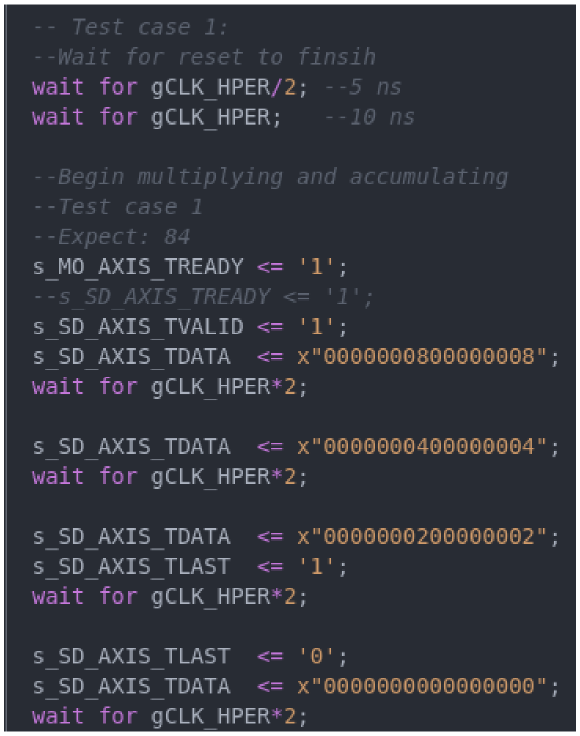
If we have multiple PE's and we are neglecting parallelism hazards/costs, multiple MACs would allow us to cut down our timing by N # of MAC units we have. So if we had 4 MAC units running in parallel, our computation time would be 201ms for Layer 2.

3.4: Manual Pipelining

a. Identify and report "natural" spots where you would consider inserting pipeline registers
in your design. Describe your overall approach, including a figure. What are the maximum
number of stages you could achieve?

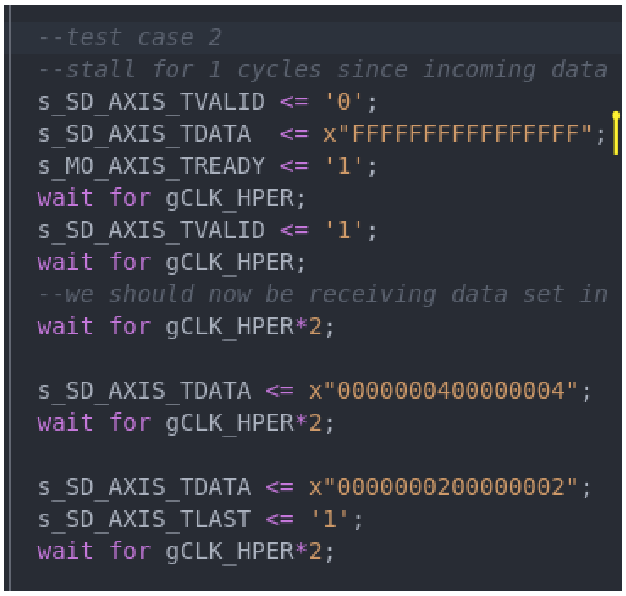
There are two "natural" spots that come to mind when thinking about pipelining the MAC unit. The first involves the multiplication process of any two values, and the second involves the accumulation of that previous pipeline stage's multiplied value into an accumulate register (which will be output based on the AXI protocol).

b. Implement your maximally pipelined design in VHDL. Test the design for edge-case inputs
using whatever approach you prefer (VHDL testbench, do file, etc.), and justify why your tests were sufficient. Submit your VHDL files and testbench in a folder called MacPipe.



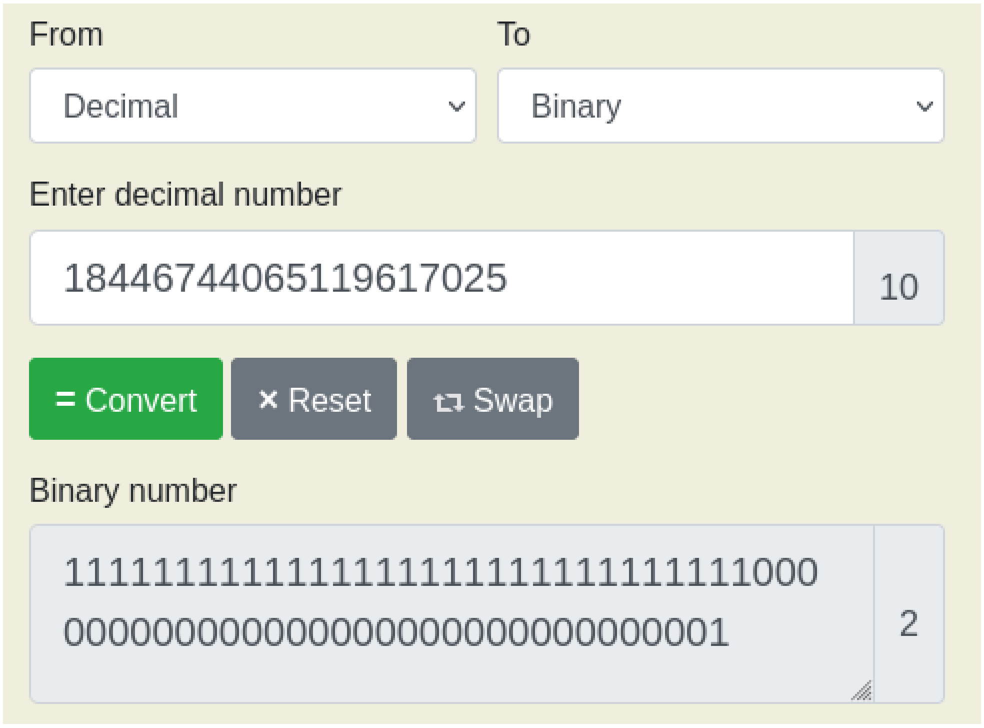
```
-- Test case 1:
--Wait for reset to finsih
wait for gCLK_HPER/2; --5 ns
wait for gCLK_HPER;   --10 ns

--Begin multiplying and accumulating
--Test case 1
--Expect: 84
s_MO_AXIS_TREADY <= '1';
--s_SD_AXIS_TREADY <= '1';
s_SD_AXIS_TVALID <= '1';
s_SD_AXIS_TDATA  <= x"0000000800000008";
wait for gCLK_HPER*2;

s_SD_AXIS_TDATA  <= x"0000000400000004";
wait for gCLK_HPER*2;

s_SD_AXIS_TDATA  <= x"0000000200000002";
s_SD_AXIS_TLAST  <= '1';
wait for gCLK_HPER*2;

s_SD_AXIS_TLAST  <= '0';
s_SD_AXIS_TDATA  <= x"0000000000000000";
wait for gCLK_HPER*2;
```

Figure 8: Pipe Mac Test Case 1



```
--test case 2
--stall for 1 cycles since incoming data
s_SD_AXIS_TVALID <= '0';
s_SD_AXIS_TDATA  <= x"FFFFFFFFFFFFFFFF";
s_MO_AXIS_TREADY <= '1';
wait for gCLK_HPER;
s_SD_AXIS_TVALID <= '1';
wait for gCLK_HPER;
--we should now be receiving data set in
wait for gCLK_HPER*2;

s_SD_AXIS_TDATA <= x"0000000400000004";
wait for gCLK_HPER*2;

s_SD_AXIS_TDATA <= x"0000000200000002";
s_SD_AXIS_TLAST <= '1';
wait for gCLK_HPER*2;
```

Figure 9: Pipe Mac Test Case 2


From
Decimal
To
Binary

Enter decimal number

18446744065119617025 (10)

Binary number

1111111111111111111111111111111000
000000000000000000000000000001 (2)

Figure 10: Concatenation Proof

Test Case 1: Multiply and accumulate multiple values over multiple cycles and then the accumulated value to output. Expect: (8*8)+(4*4)+(2*2)=84

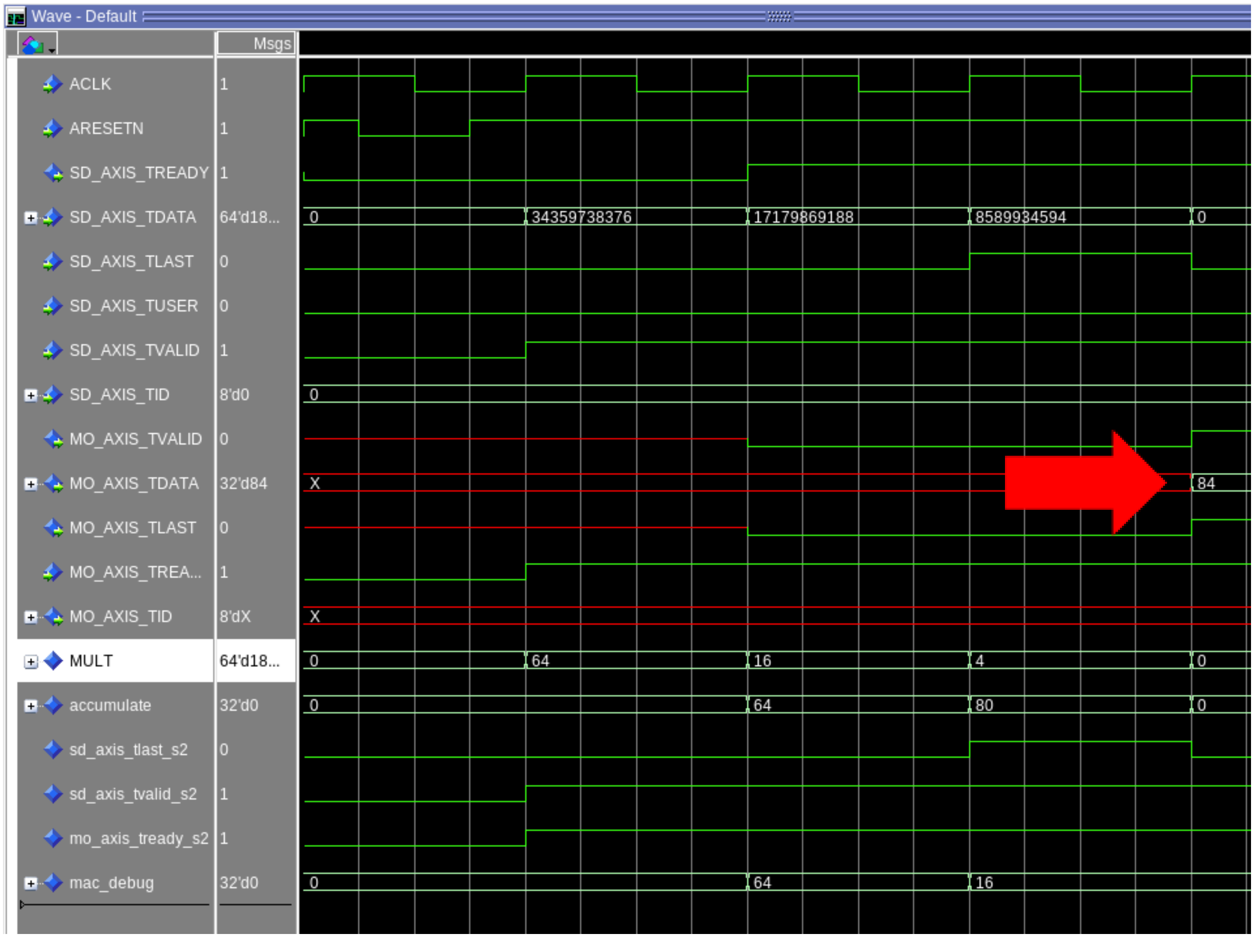


Figure 11: Test Case 1 Simulation

Test Case 2: For the second test case we wanted to make sure our MAC would stall if we were not ready for data. We also wanted to make sure our MAC concatenated an overflow value correctly. It successfully did this and to be sure the last partial sum accumulated value didn't interfere with the new one we began a new accumulation. For this test case expect: (FFFFFFFF*FFFFFFFF) + (4 * 4) + (2 * 2) = big number, but this value is overflow so our MAC contantenates the FFFFFFFF*FFFFFFFF to 1 so the ending output is 21.

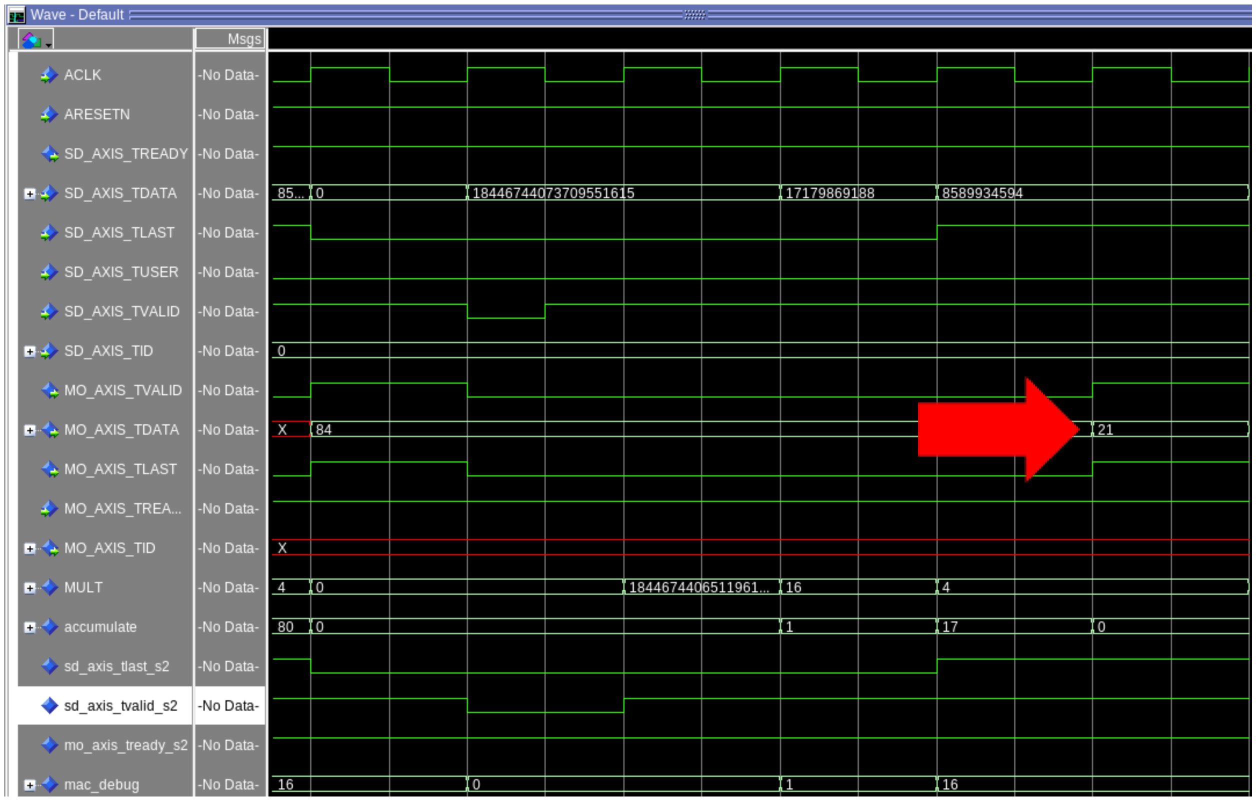


Figure 12: Test Case 2 Simluations

c. Synthesize your pipelined MAC unit and report the maximum frequency. Calculate the maximum throughput of a layer of your choosing as you did with the previous Mac.

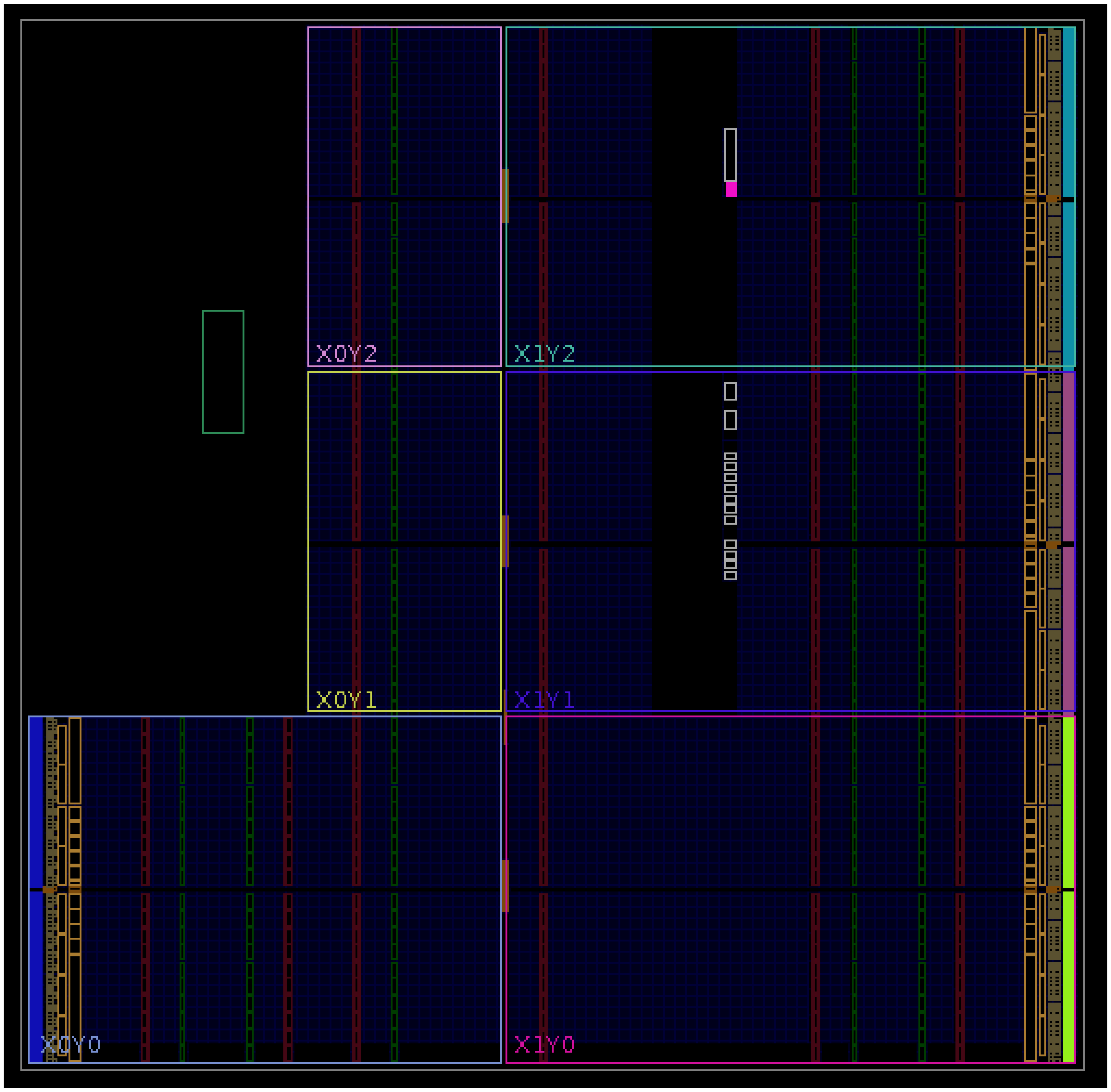
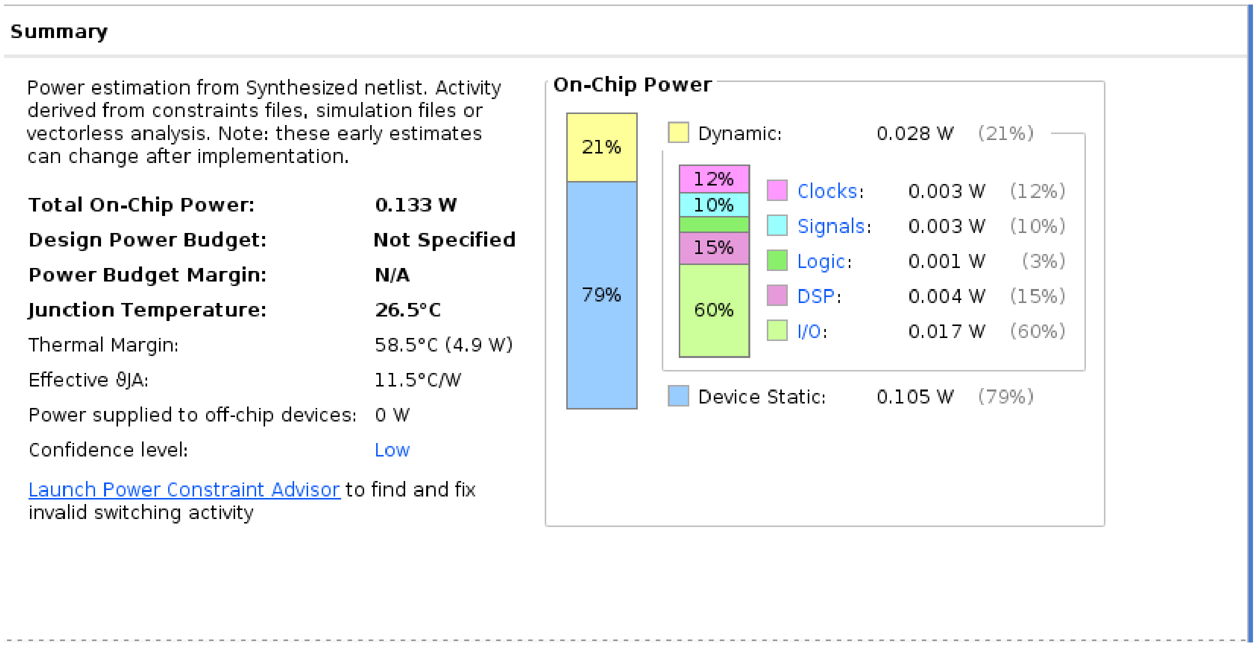


Figure 13: Piped MAC layout

Max Clock Freq: 200MHz
Total On-Chip Power: 133mW

Critical Path: 5 ns


Summary

Power estimation from Synthesized netlist. Activity derived from constraints files, simulation files or vectorless analysis. Note: these early estimates can change after implementation.

Total On-Chip Power:	**0.133 W**
Design Power Budget:	**Not Specified**
Power Budget Margin:	**N/A**
Junction Temperature:	**26.5°C**
Thermal Margin:	58.5°C (4.9 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

On-Chip Power

🟨	Dynamic:	0.028 W	(21%)
🟪	Clocks:	0.003 W	(12%)
🟦	Signals:	0.003 W	(10%)
🟩	Logic:	0.001 W	(3%)
🟪	DSP:	0.004 W	(15%)
🟩	I/O:	0.017 W	(60%)
🟦	Device Static:	0.105 W	(79%)

Figure 14: Piped MAC Power Report

Throughput = 1 instruction per cycle
Throughput: If we were to calculate Layer 2 of our DNN using this MAC, we need to calculate how many MACs Layer 2 needs and how many cycles each MAC takes. Layer 2 has a 5x5x32 Filter, so we have to calculate 800 inputs multiplied by 800 weights. Each multiply and accumulate takes 1 cycle after the initial 2,. So each output element from the MAC takes 802 (800+2) cycles to calculate. We have to calculate 100352 output elements (56*56*32 output size), so the total number of cycles we need to go through to compute this layer is 80,482,304 (100352*8002). Our critical path is 5ns, so if we would only use this MAC to compute the layer, it would take 402ms.

d. Considering the maximum depth of your pipeline, what are the hazards that exist? Under what conditions for your specific DNN layer would these hazards require attention? Show
your calculations and describe your reasoning.

The biggest hazard that we saw is when we have valid data that we want to output but the master is not ready to receive data, we need to stall everything whether we are receiving valid data or not. Another hazard would be if we are receiving valid data but we are not ready to receive data, we would stall the first stage.