Image Digit Classifier

Chinthana Wimalasuriya
856558971

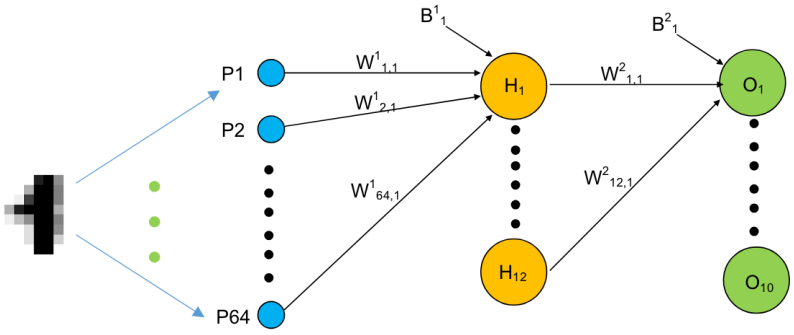
Sunimali Rathnayake
856564916

TABLE OF CONTENTS

1 TASK DESCRIPTION

Handwritten digit recognition is a fundamental problem in the field of pattern recognition and machine learning. In this project, we aim to design and implement an artificial neural network (ANN) capable of accurately identifying handwritten digits. The input data consists of images with dimensions of 8x8 pixels, where each pixel is represented by a value ranging from -0.5 to 0.5. These images serve as the basis for training and testing our neural network model.

The ANN architecture comprises three layers: an input layer, a hidden layer, and an output layer. The input layer consists of 64 nodes, each corresponding to a pixel in the input image. The hidden layer contains 12 nodes, serving as intermediate processing units. Finally, the output layer consists of 10 nodes, with each node representing a digit value from 0 to 9, which are the possible classifications for the handwritten digits.



Milestone 1: Multiplier and Accumulator (MAC) Module Design

Description: Design and implement the MAC module responsible for computing the weighted sum of inputs at each node in the neural network layers.

Milestone 2: Accumulator (ACC) Module Design

Description: Develop the Accumulator (ACC) module to accumulate the results obtained from the MAC module and prepare them for further processing.

Milestone 3: Integrating with sigmoid IP block

Description: Integrate the previously designed MAC and ACC modules with the provided sigmoid IP block, enabling the implementation of the activation function within the neural network.

Milestone 4: Complete the neural network circuit (Bonus Task)

Description: Finalize the neural network circuit by integrating all components, including input, hidden, and output layers, and ensuring the proper flow of data through the network.

2 DESIGN CONSIDERATIONS

In our design process, we acknowledge the critical importance of considering various factors beyond technical specifications to ensure the overall impact of our system aligns with societal needs and values. While addressing technical requirements such as the number of layers, nodes, latency, and hardware usage, we also account for the following considerations:

1. Safety: Our design prioritizes the safety and well-being of users by implementing robust security measures to mitigate potential risks and ensuring reliable operation.
2. Environmental Sustainability: We strive to minimize the environmental footprint of our system by optimizing resource usage, energy efficiency, and waste reduction throughout its lifecycle, from manufacturing to disposal.
3. Global Impact: Our design considers the interconnectedness of global systems, aiming to contribute positively to global challenges such as climate change and social equity.
4. Economic Efficiency: We optimize cost-effectiveness and resource allocation in our design to ensure affordability and accessibility without compromising quality or safety.

Considering these factors, our proposed architecture, featuring a 16 to 1 multiply-add tree combined with a 4-step accumulator, offers both technical efficiency and alignment with broader societal goals. This architecture accommodates the processing needs of up to 64 nodes in a single pass through the accumulator, facilitating scalability and adaptability to diverse network configurations. Additionally, the modular design allows for customization to meet specific application requirements while minimizing environmental impact and maximizing societal benefit.

3 MILESTONE 01: MULTIPLIER AND ACCUMULATOR

In this phase of the project, the design focuses on creating a Multiply-Accumulate (MAC) module capable of performing computations according to the formula:

$$y = \sum_{i=1}^{16} w_i x_i$$

Multiplier Module: Responsible for multiplying the respective w and x values.

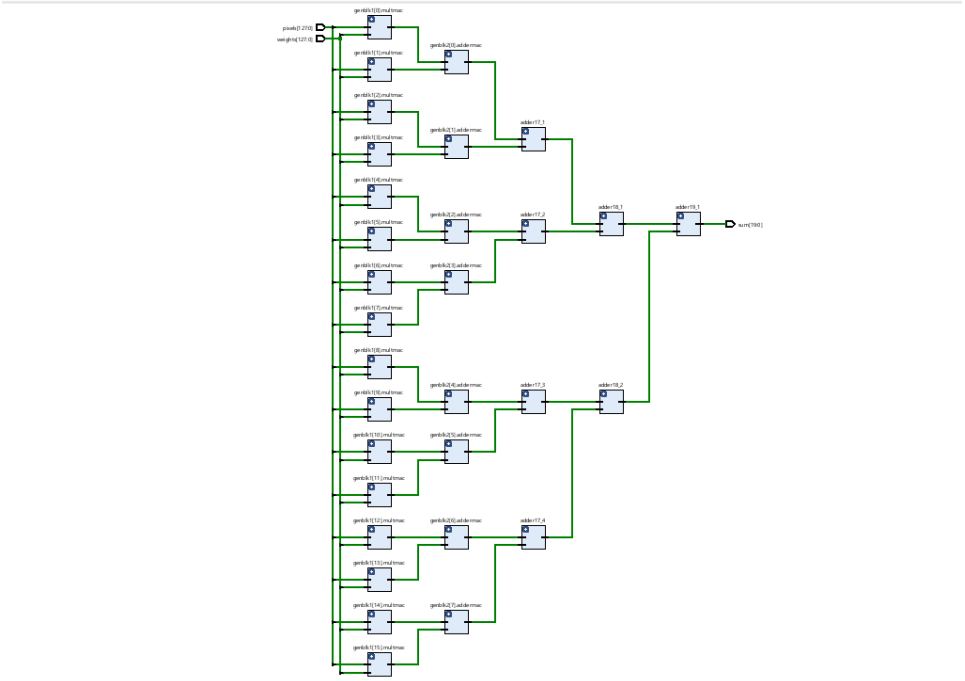
Adder Module: Performs the accumulation of multiplication results over the iterations, resulting in the final output.

3.1 MAC Module Design

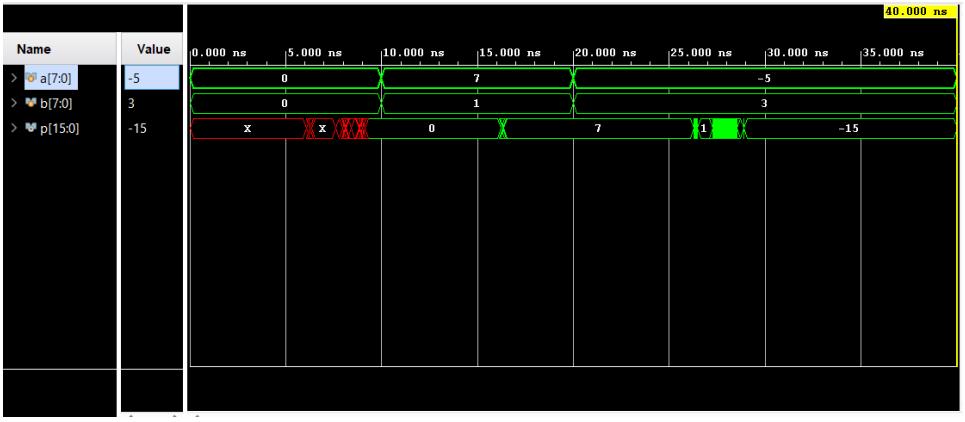
3.1.1 Functionality And Structure

The MAC module contains a row of 16 multipliers whose outputs are then added via a 4-level adder tree. The module takes a pair of 128-bit vectors, namely data_p and data_w. each of these vectors are then broken down into sets of 16 data values, 8 bits each. They're then paired up, one from data_p and one from data_w and fed into each multiplier. Multipliers produce 16-bit vectors, fed into the adder tree, each level of which adds one bit, and the final adder tree output is 20 bits. To allow working with higher clock frequencies, MAC can be pipelined, which is explained later. With the fully pipelined version, latency is 3 cycles, and the throughput is 1.

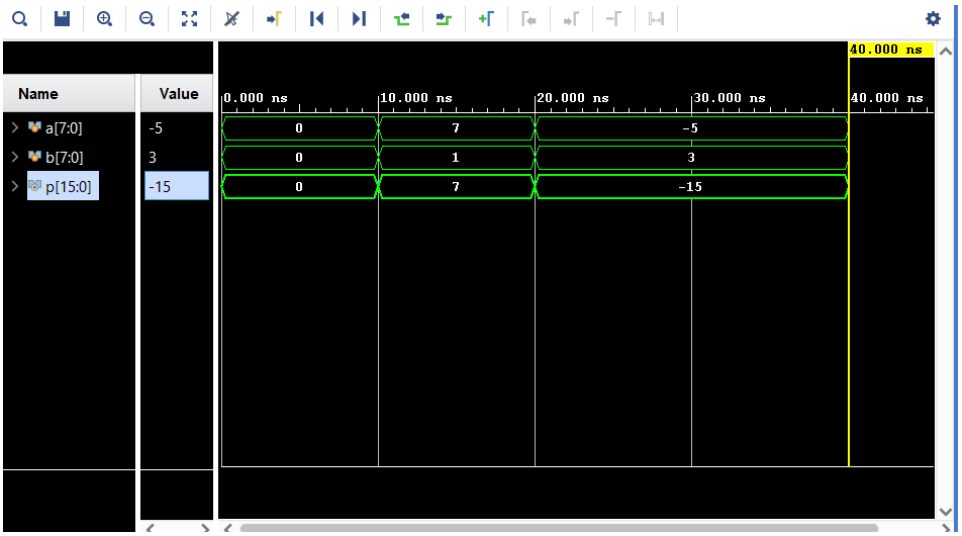
3.1.2 RTL Analysis Schematic



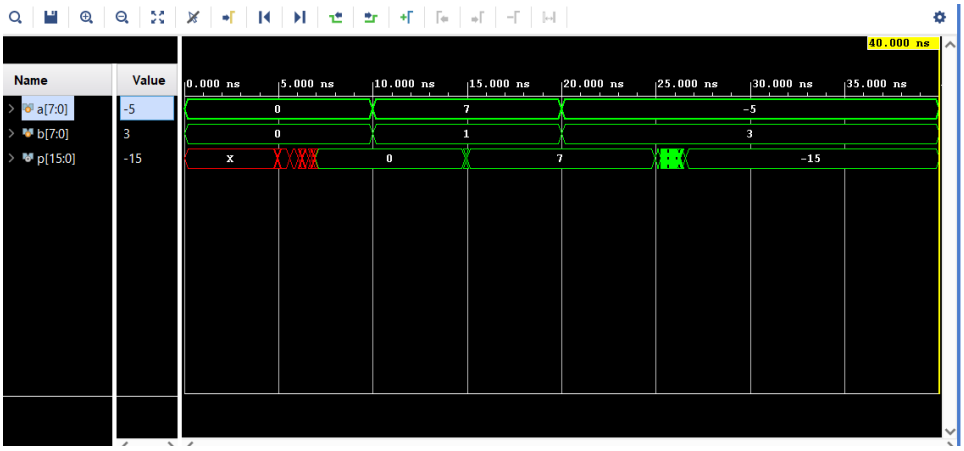
3.1.3 Multiplier Testing – Post Implementation Simulation



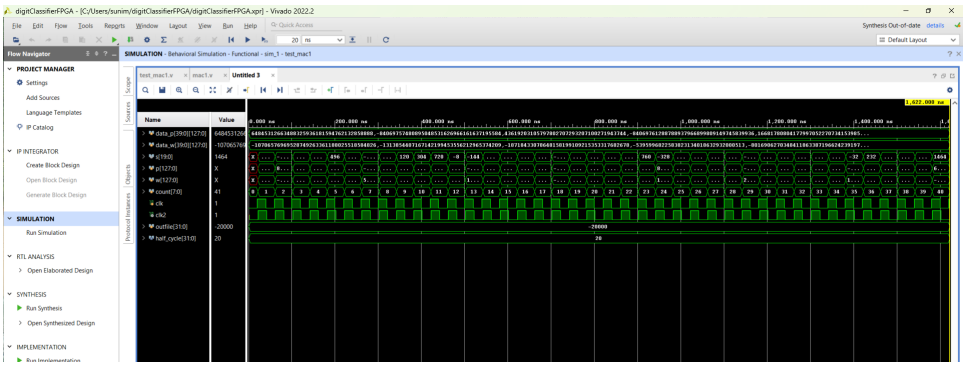
3.1.4 Multiplier Testing – Behavioral Simulation



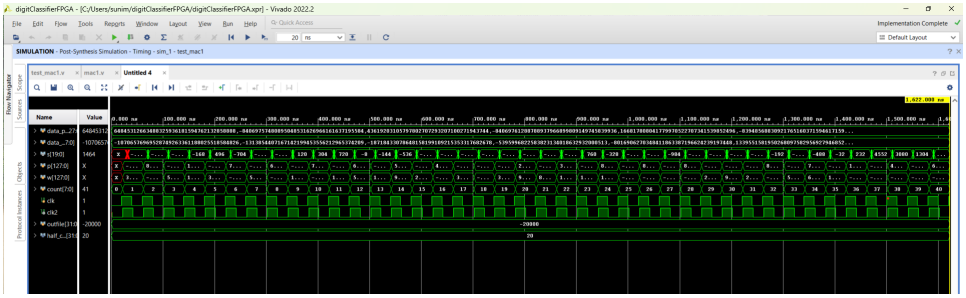
3.1.5 Multiplier Testing – Post Synthesis Simulation



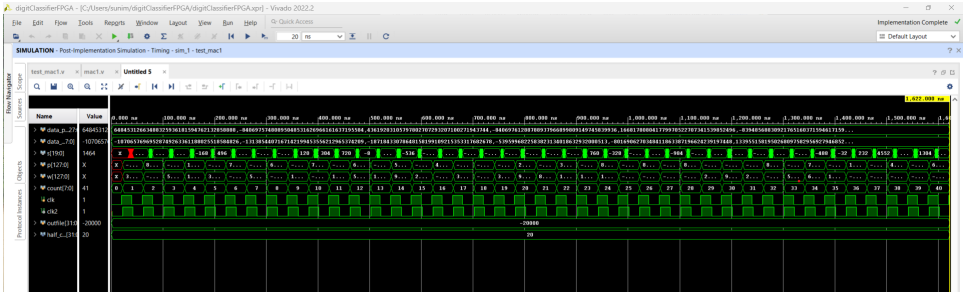
3.1.6 Behavioral Simulation



3.1.7 Post Synthesis Simulation



3.1.8 Post Timing Simulation

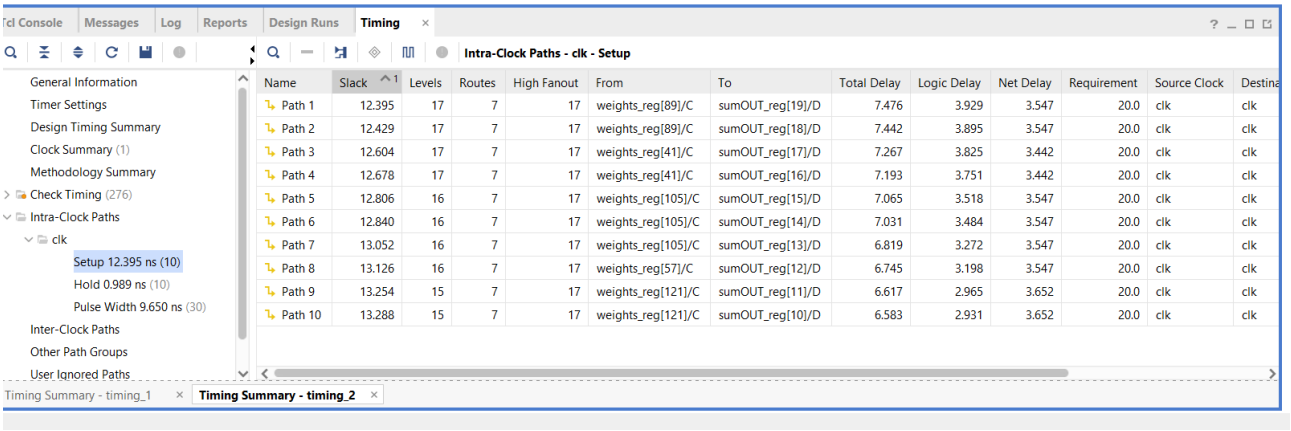


3.2 Pipelining MAC Module

To enhance the frequency and throughput of the MAC module pipelining was introduced. Initially, a two-stage pipeline configuration was implemented, timing analysis was conducted to assess its compliance with project time constraints. Upon evaluation, it was determined that the two-stage pipeline configuration sufficiently met the time constraints, thus demonstrating its feasibility in improving efficiency without violating timing requirements.

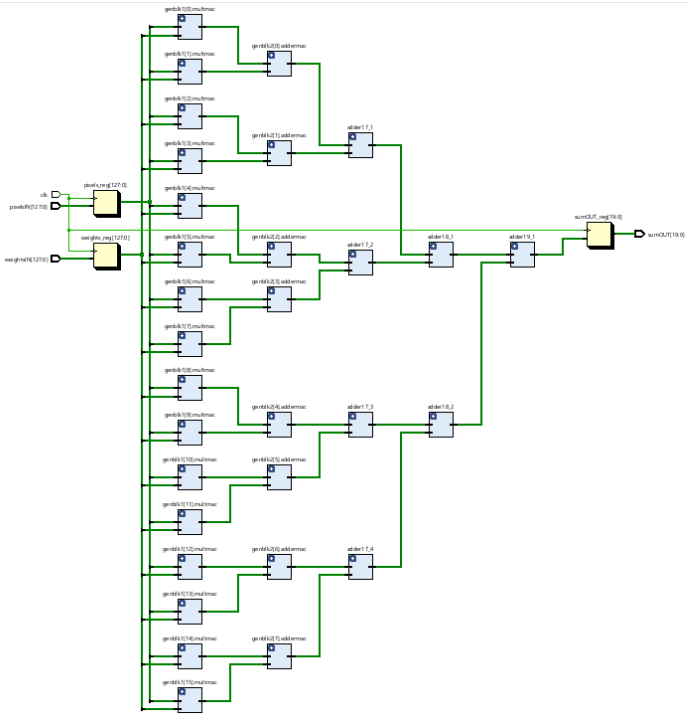
3.2.1 MAC2 Module – Two Pipeline Stages

3.2.1.1 Timing Analysis Report



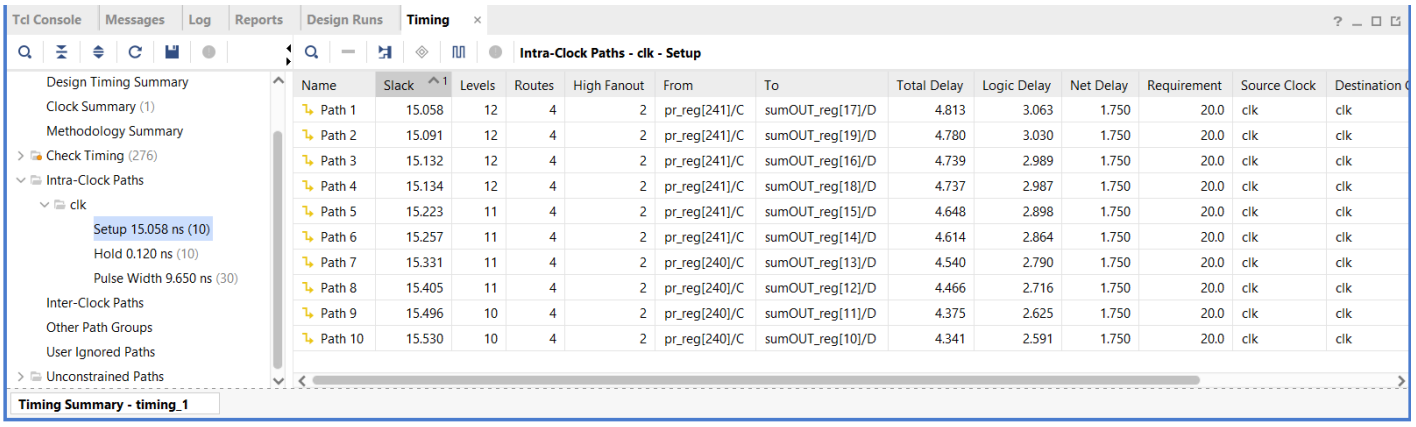
Name	Slack	Levels	Routes	High Fanout	From	To	Total Delay	Logic Delay	Net Delay	Requirement	Source Clock	Destina
Path 1	12.395	17	7	17	weights_reg[89]/C	sumOUT_reg[19]/D	7.476	3.929	3.547	20.0	clk	clk
Path 2	12.429	17	7	17	weights_reg[89]/C	sumOUT_reg[18]/D	7.442	3.895	3.547	20.0	clk	clk
Path 3	12.604	17	7	17	weights_reg[41]/C	sumOUT_reg[17]/D	7.267	3.825	3.442	20.0	clk	clk
Path 4	12.678	17	7	17	weights_reg[41]/C	sumOUT_reg[16]/D	7.193	3.751	3.442	20.0	clk	clk
Path 5	12.806	16	7	17	weights_reg[105]/C	sumOUT_reg[15]/D	7.065	3.518	3.547	20.0	clk	clk
Path 6	12.840	16	7	17	weights_reg[105]/C	sumOUT_reg[14]/D	7.031	3.484	3.547	20.0	clk	clk
Path 7	13.052	16	7	17	weights_reg[105]/C	sumOUT_reg[13]/D	6.819	3.272	3.547	20.0	clk	clk
Path 8	13.126	16	7	17	weights_reg[57]/C	sumOUT_reg[12]/D	6.745	3.198	3.547	20.0	clk	clk
Path 9	13.254	15	7	17	weights_reg[121]/C	sumOUT_reg[11]/D	6.617	2.965	3.652	20.0	clk	clk
Path 10	13.288	15	7	17	weights_reg[121]/C	sumOUT_reg[10]/D	6.583	2.931	3.652	20.0	clk	clk

3.2.1.2 RTL Analysis Schematic Diagram



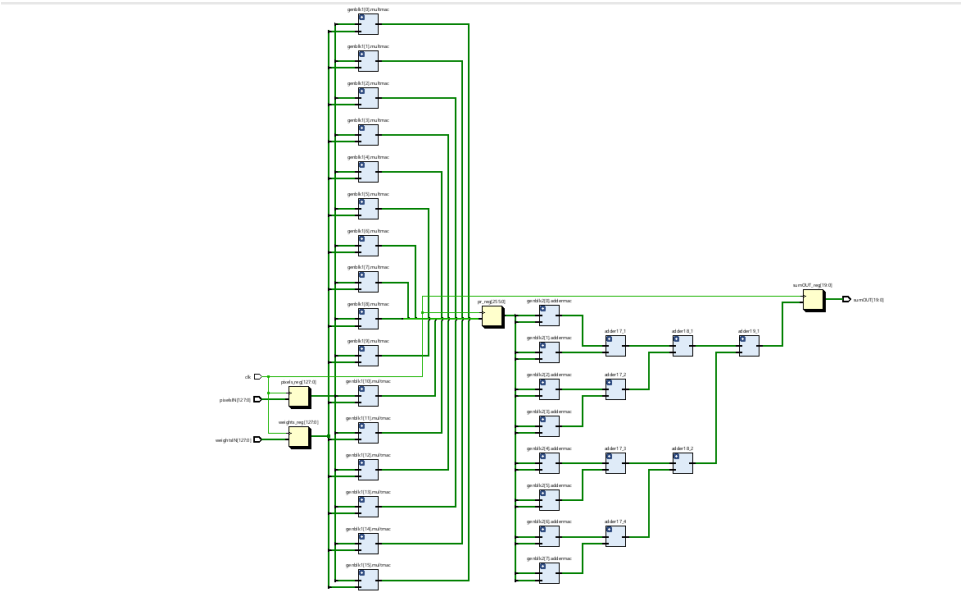
3.2.2 MAC3 Module – Three Pipeline Stages

3.2.2.1 Timing Analysis Report

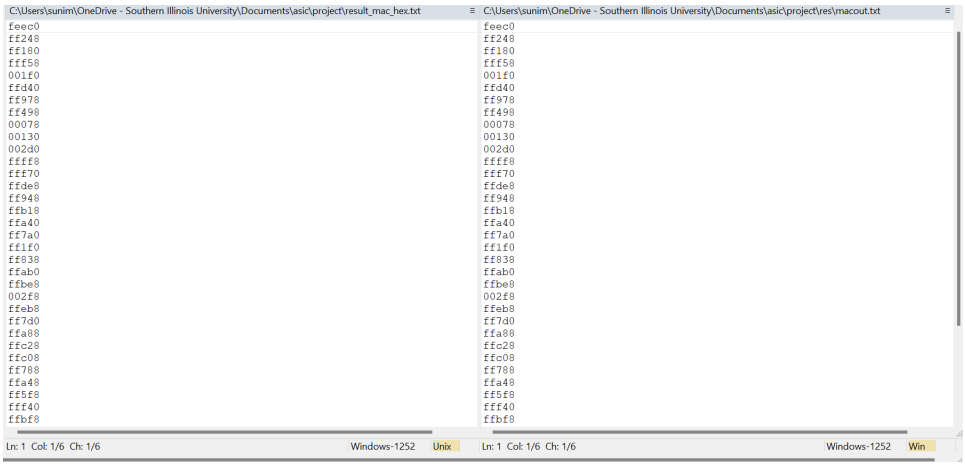


Name	Slack	Levels	Routes	High Fanout	From	To	Total Delay	Logic Delay	Net Delay	Requirement	Source Clock	Destination
Path 1	15.058	12	4	2	pr_reg[241]/C	sumOUT_reg[17]/D	4.813	3.063	1.750	20.0	clk	clk
Path 2	15.091	12	4	2	pr_reg[241]/C	sumOUT_reg[19]/D	4.780	3.030	1.750	20.0	clk	clk
Path 3	15.132	12	4	2	pr_reg[241]/C	sumOUT_reg[16]/D	4.739	2.989	1.750	20.0	clk	clk
Path 4	15.134	12	4	2	pr_reg[241]/C	sumOUT_reg[18]/D	4.737	2.987	1.750	20.0	clk	clk
Path 5	15.223	11	4	2	pr_reg[241]/C	sumOUT_reg[15]/D	4.648	2.898	1.750	20.0	clk	clk
Path 6	15.257	11	4	2	pr_reg[241]/C	sumOUT_reg[14]/D	4.614	2.864	1.750	20.0	clk	clk
Path 7	15.331	11	4	2	pr_reg[240]/C	sumOUT_reg[13]/D	4.540	2.790	1.750	20.0	clk	clk
Path 8	15.405	11	4	2	pr_reg[240]/C	sumOUT_reg[12]/D	4.466	2.716	1.750	20.0	clk	clk
Path 9	15.496	10	4	2	pr_reg[240]/C	sumOUT_reg[11]/D	4.375	2.625	1.750	20.0	clk	clk
Path 10	15.530	10	4	2	pr_reg[240]/C	sumOUT_reg[10]/D	4.341	2.591	1.750	20.0	clk	clk

3.2.2.2 RTL Analysis Schematic Diagram



The simulated MAC module results were compared with the provided results file, and thorough testing was conducted to verify accuracy. No mismatches were found, confirming the reliability of our implementation. This validation process ensured consistency between our simulated data and the provided results, bolstering confidence in our testing procedures and the correctness of the MAC module's computations.


C:\Users\sunim\OneDrive - Southern Illinois University\Documents\asic\project\result_mac_hex.txt
feec0
ff248
ff180
fff58
001f0
ffd40
ff978
ff498
00078
00130
002d0
ffff8
fff70
ffde8
ff948
ffb18
ffa40
ff7a0
ff1f0
ff838
ffab0
ffbe8
002f8
ffeb8
ff7d0
ffa88
ffc28
ffc08
ff788
ffa48
ff5f8
fff40
ffbf8
Ln: 1 Col: 1/6 Ch: 1/6
Windows-1252
Unix
C:\Users\sunim\OneDrive - Southern Illinois University\Documents\asic\project\res\macout.txt
feec0
ff248
ff180
fff58
001f0
ffd40
ff978
ff498
00078
00130
002d0
ffff8
fff70
ffde8
ff948
ffb18
ffa40
ff7a0
ff1f0
ff838
ffab0
ffbe8
002f8
ffeb8
ff7d0
ffa88
ffc28
ffc08
ff788
ffa48
ff5f8
fff40
ffbf8
Ln: 1 Col: 1/6 Ch: 1/6
Windows-1252
Win

4 MILESTONE 02: ACCUMULATOR (ACC) MODULE DESIGN

To compute the output of a hidden node, 64 multiplication results are needed. We'll use the MAC module four times, each handling 16 multiplications. Their results will then be accumulated. Since the MAC output is 20 bits and four MAC results are to be accumulated, the Accumulator will take four clock cycles to produce a result, resulting in a 22-bit output. This approach optimizes the MAC module's efficiency while accurately accumulating the final result.

Main Components

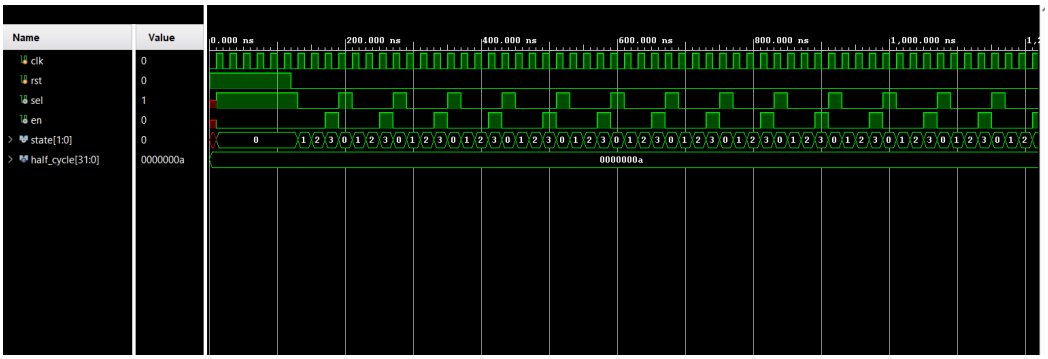
- ACC control module
- Adder module

4.1 ACC Controller Module

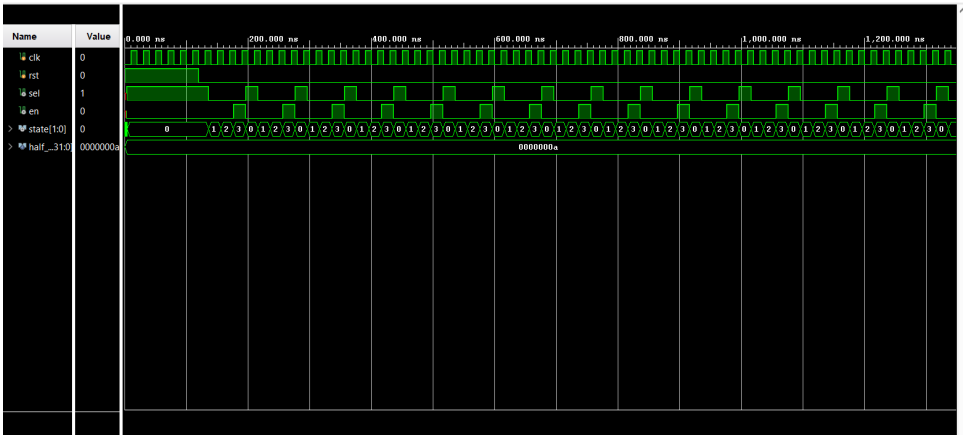
4.1.1 Structure And Functionality

ACC Controller is responsible for governing the functionality of the accumulator, which are mainly setting the initial state and passing the final value to the output when the accumulation cycle is complete. This is actualized via two outputs, *sel* and *en*. *Sel* switches one of the adder inputs between bias and accumulator register, while *en* passes the accumulator register value to the output. The module basically runs through a 4 state FSM, and for the first state, *sel* is asserted high. Then, until the final state, both outputs are asserted low. Finally, in the last cycle, en is asserted high.

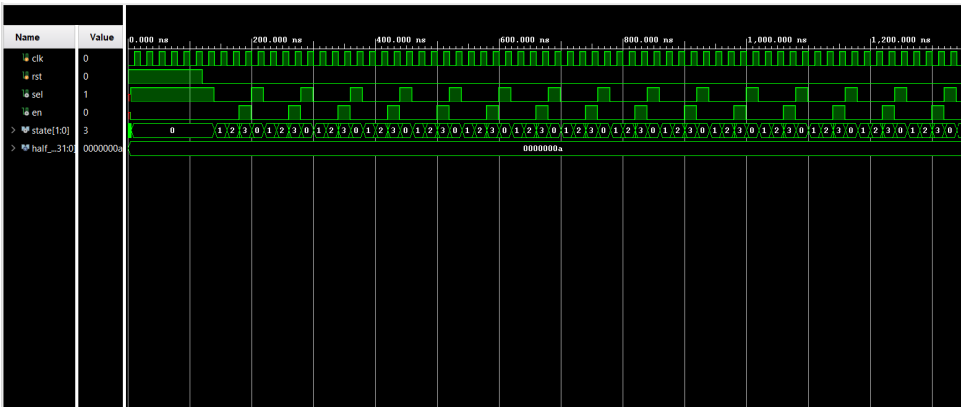
4.1.2 Behavioral Simulation



4.1.3 Post Synthesis Simulation



4.1.4 Post Implementation Simulation

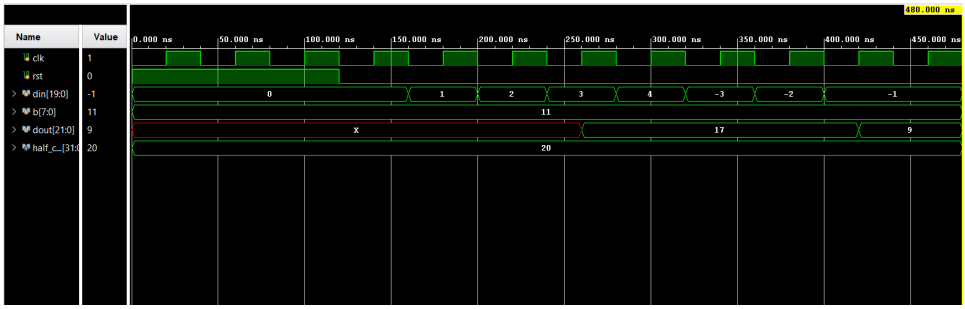


4.2 ACC Module

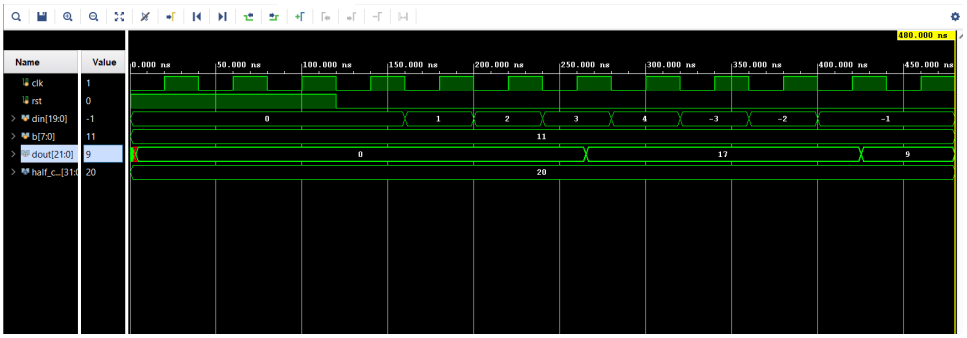
4.2.1 Structure And Functionality

The ACC module takes a series of 4 data values and one bias value, all of which are then summed together to form one final output value. Since this is 4 values of 20 bits each are added together, the final output is 22 bits wide. The throughput is ¼ and latency is 4 cycles.

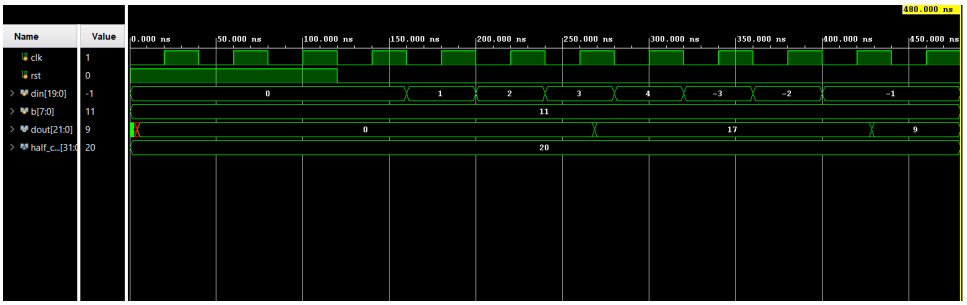
4.2.2 Behavioral Simulation



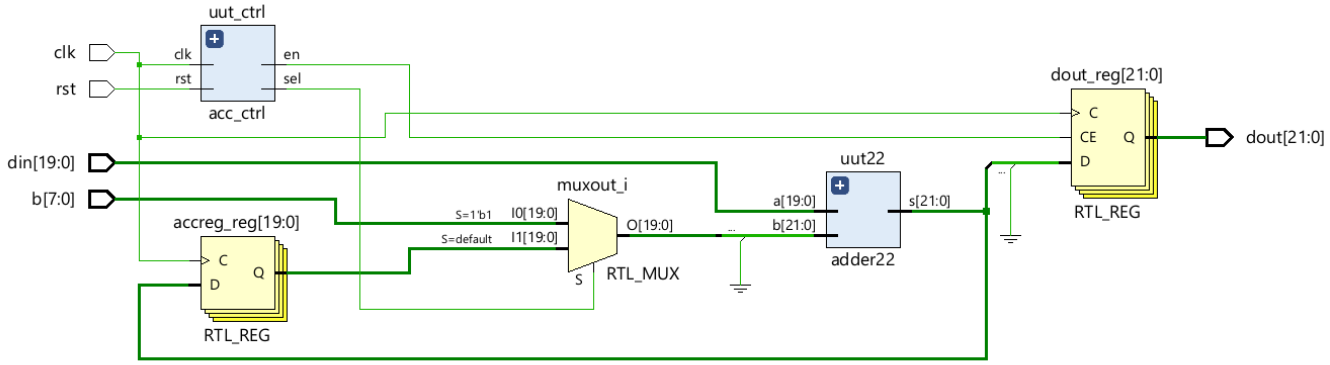
4.2.3 Post Synthesis Simulation



4.2.4 Post Implementation Simulation



4.2.5 RTL Schematic Diagram

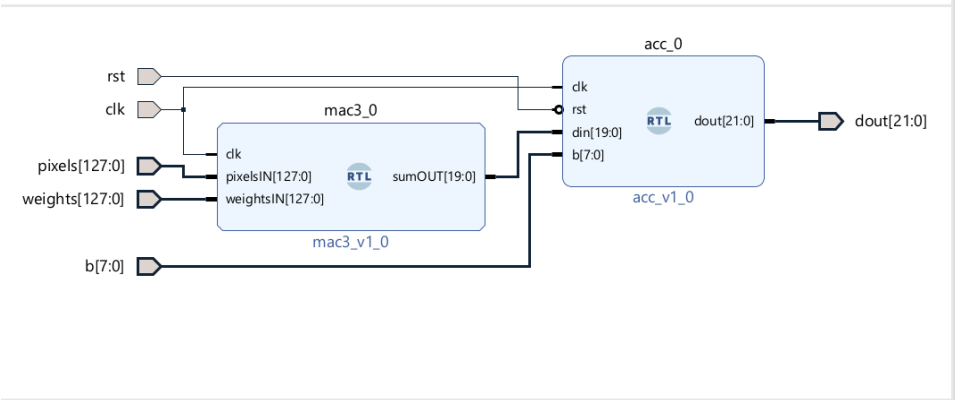


4.3 ACC – MAC3 Module Integration

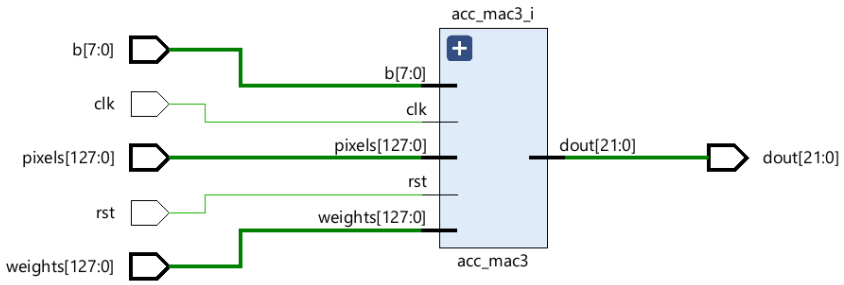
4.3.1 Functionality

This module combines the ACC and MAC into one block, allowing it to take 4 pairs of 128-bit vectors and sum them into one value. This is analogous to a node without an activation function.

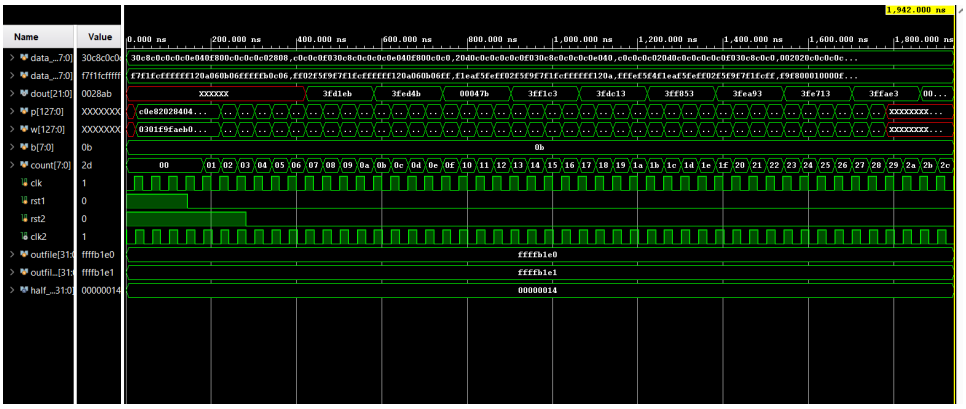
4.3.2 Block Diagram



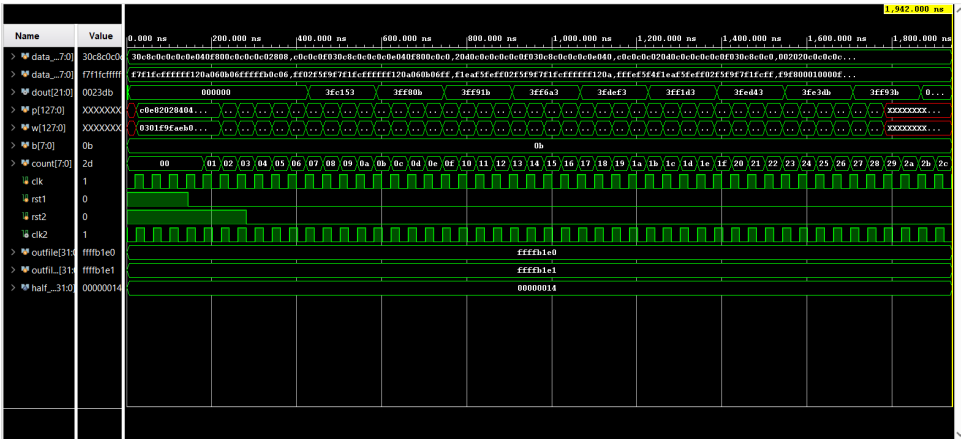
4.3.3 RTL Analysis Schematic



4.3.4 Behavioral Simulation

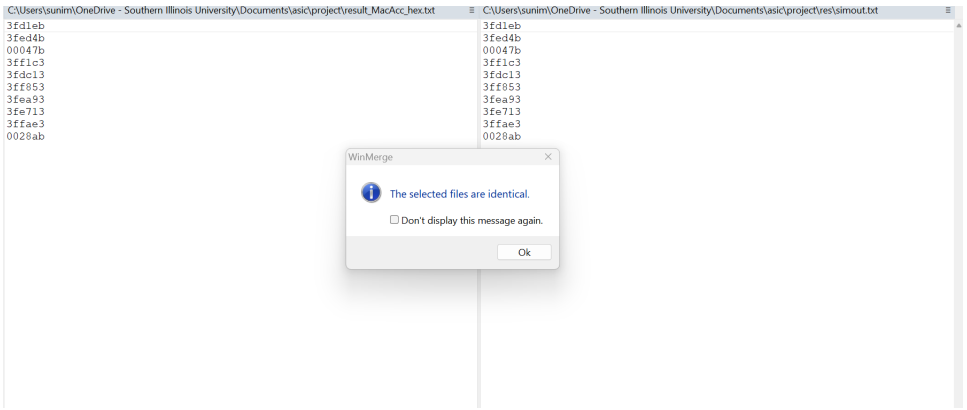


4.3.5 Post Synthesis Simulation



4.3.6 Post Implementation Simulation

The MAC3-ACC module underwent testing, and the results files were compared with the provided results file. No mismatches were identified during this evaluation. This outcome underscores the reliability and accuracy of our testing procedures and confirms the correctness of the MAC3-ACC module's computations.


C:\Users\sunim\OneDrive - Southern Illinois University\Documents\asic\project\result_MacAcc_hex.txt
3fdleb
3fed4b
00047b
3ff1c3
3fdc13
3ff853
3fea93
3fe713
3ffae3
0028ab
C:\Users\sunim\OneDrive - Southern Illinois University\Documents\asic\project\res\simout.txt
3fdleb
3fed4b
00047b
3ff1c3
3fdc13
3ff853
3fea93
3fe713
3ffae3
0028ab
WinMerge
The selected files are identical.
Don't display this message again.
Ok

5 MILESTONE 03: INTEGRATING WITH SIGMOID IP BLOCK

In this design phase, the activation function circuit was integrated using a provided sigmoid IP block. We accomplished this integration by adding the IP block into the user IP repositories within the project.

Implemented sigmoid function using:

- Sigmoid IP block (provided)
- Sigmoid IP interface

5.1 Sigmoid IP Block

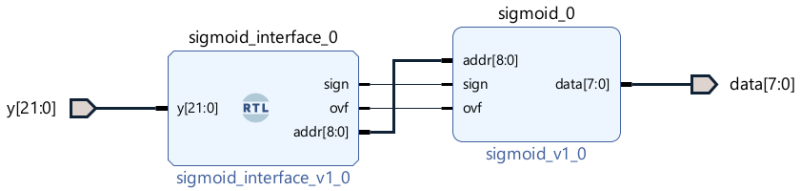
Afterward, we proceeded with behavioral simulation testing, as the sigmoid IP block couldn't be synthesized due to a version mismatch in Vivado.

5.2 Sigmoid Function Integration

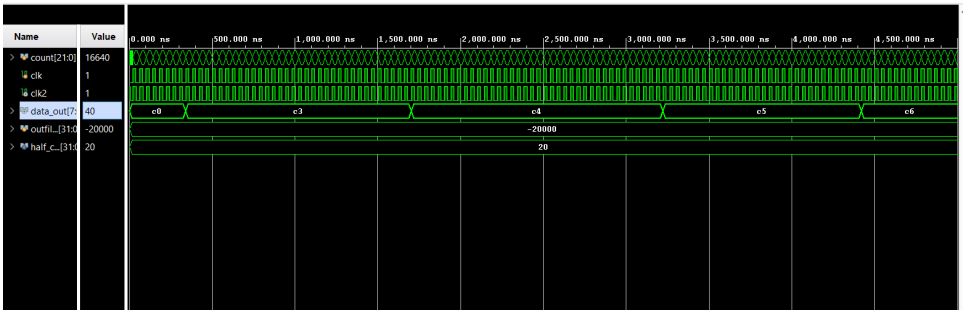
5.2.1 Structure And Functionality

The sigmoid module contains two parts. The provided sigmoid IP, which is essentially a lookup table, and the sigmoid interface. The interface will take a 22-bit number from the accumulator and generate the lookup table address, sign and overflow signals. The overflow signal signifies that the absolute value of the input has reached the end of the defined quantized sigmoid values, and the IP output will get clipped at either -0.5 or 0.5, depending on the sign. The final output from the module will be 8 bits wide.

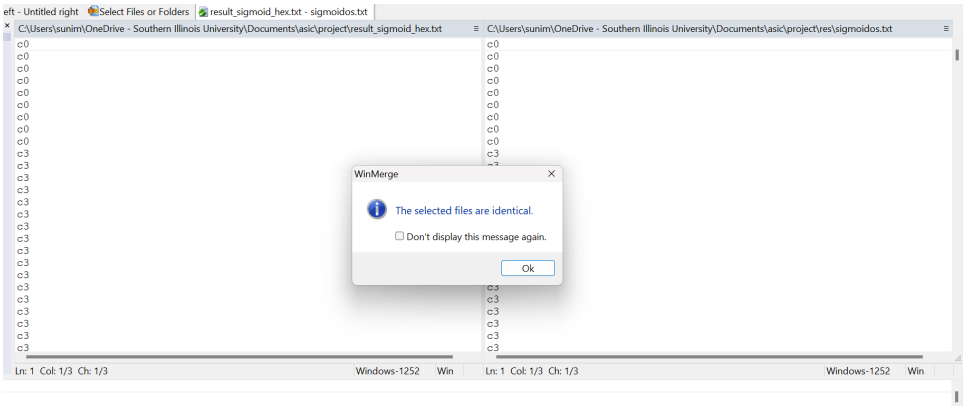
5.2.2 Block Diagram



5.2.3 Behavioral Simulation



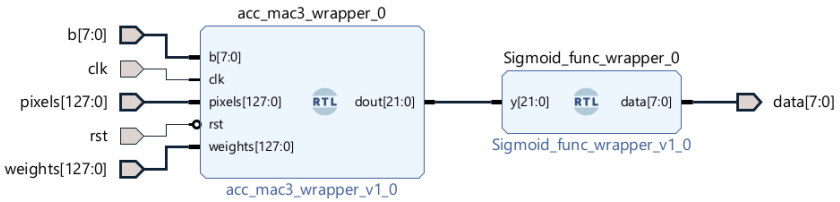
We utilized the test bench file (test_sigmoid.v) to simulate our design and compared the simulation results with the expected output file. There were no mismatches between the simulation results obtained using the testing and the expected output file.



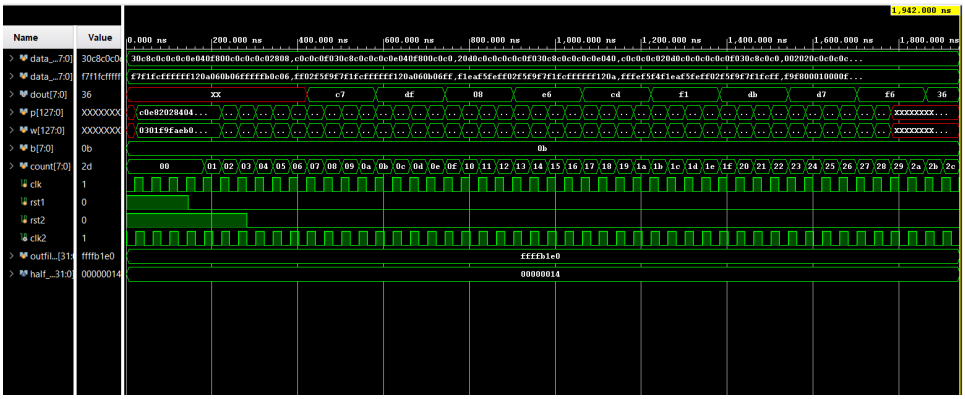
5.3 Node Function Design

In this phase, we integrated the MAC3-ACC module and the sigmoid function together. This circuit takes 64 pixels as input to compute the output of a network node. This is the complete node functionality that includes multiplication, summation, bias, and sigmoid activation. It takes a 128-bit data vector, 128-bit weight vector and an 8-bit bias vector. The output is an 8-bit vector that is the activation output. Overall, the module has a 7-clock cycle latency and a ¼ throughput.

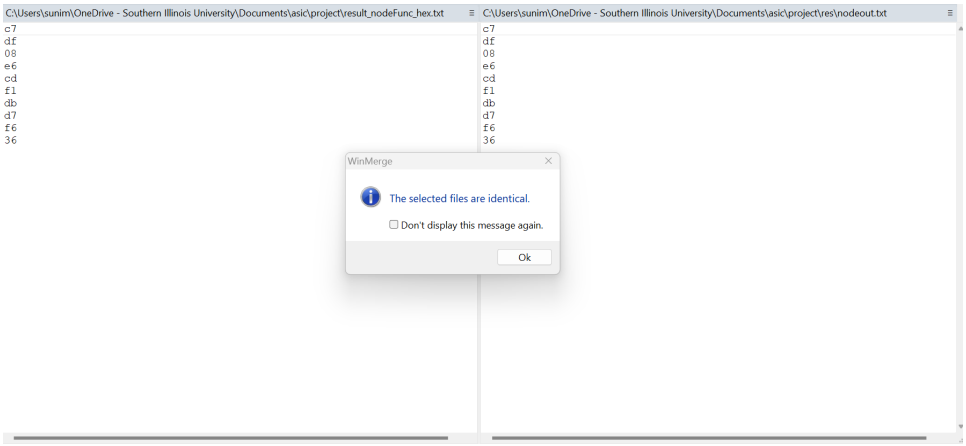
5.3.1 Block Diagram



5.3.2 Behavioral Simulation



We utilized the test bench to verify the synthesized netlist with inputs from weights_hex.txt and digits_hex.txt files. The expected output file matched the generated test result file, confirming the correctness of our implementation.

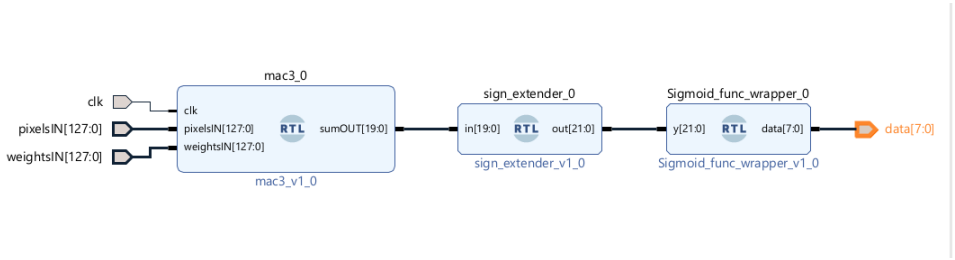


6 MILESTONE 04: BONUS POINTS: COMPLETE THE NEURAL NETWORK CIRCUIT

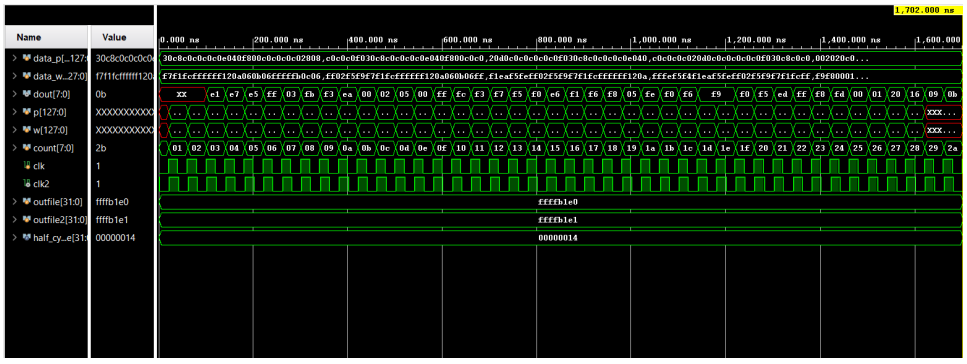
6.1 Outnode Function Design

We created a design for the output node function by integrating the MAC3 and sigmoid function together. This does not include the acc, and only sums the inputs in one cycle. The I/O configuration is identical to the node function, minus the bias. It has a 3-clock cycle latency and a throughput of 1.

6.1.1 Block Diagram

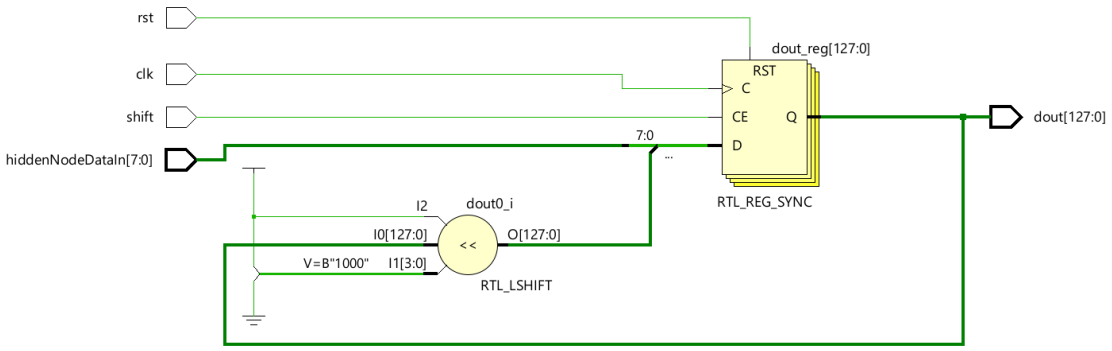


6.1.2 Behavioral Simulation

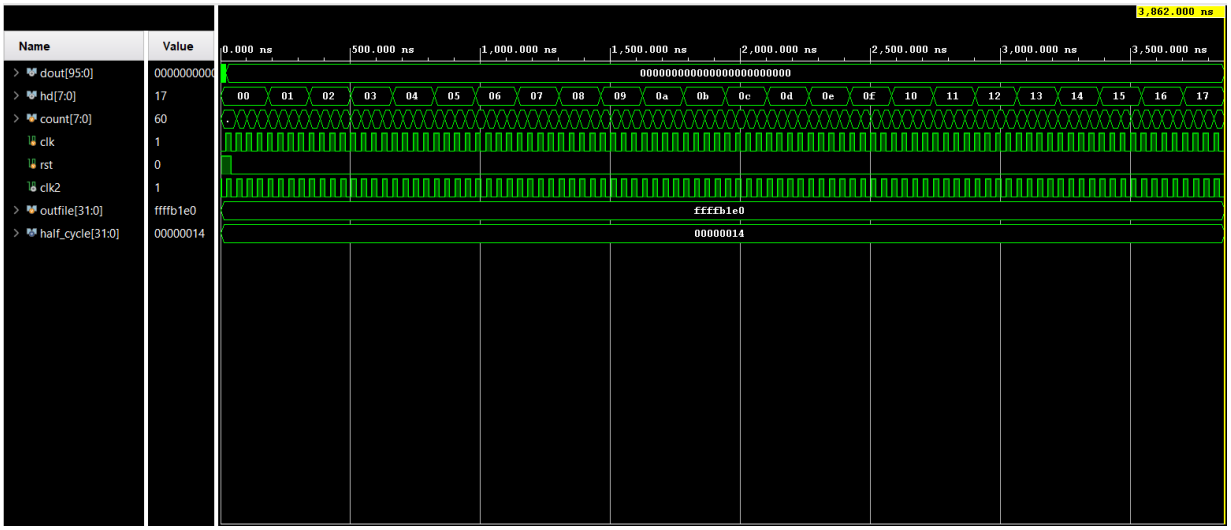


6.2 Hidden Node Register Design

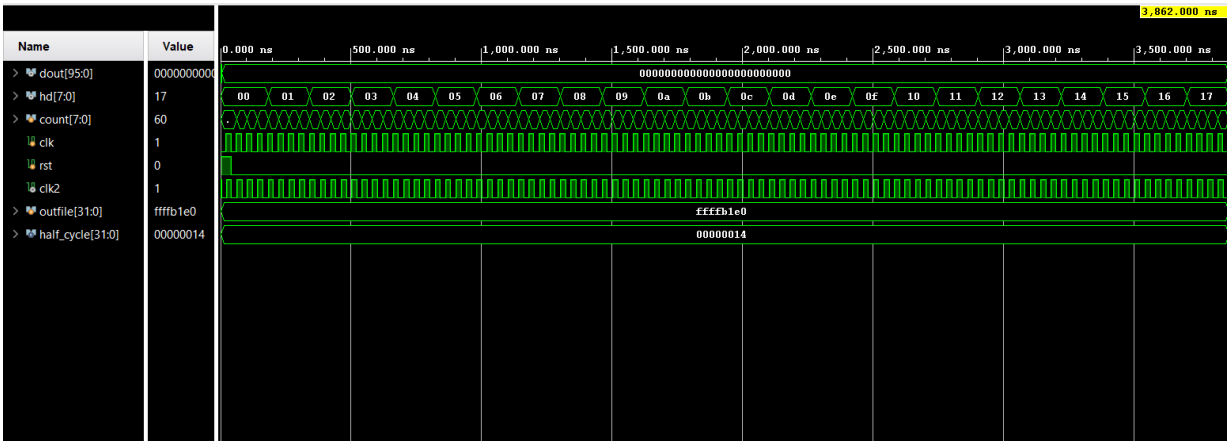
6.2.1 RTL Analysis Schematic



6.2.2 Behavioral Simulation

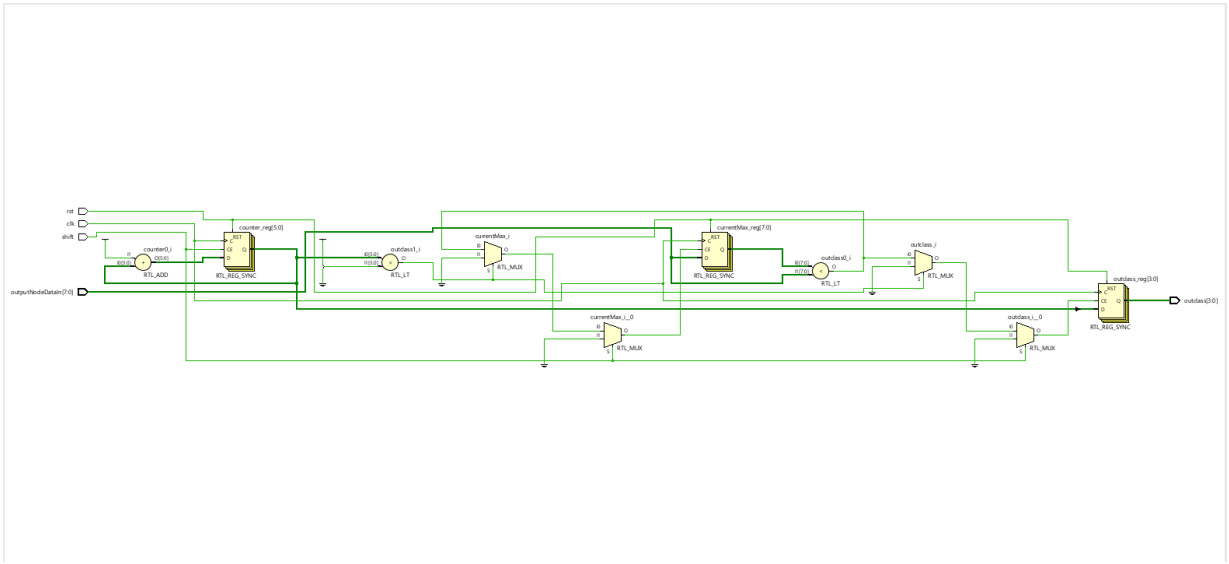


6.2.3 Post Synthesis Simulation

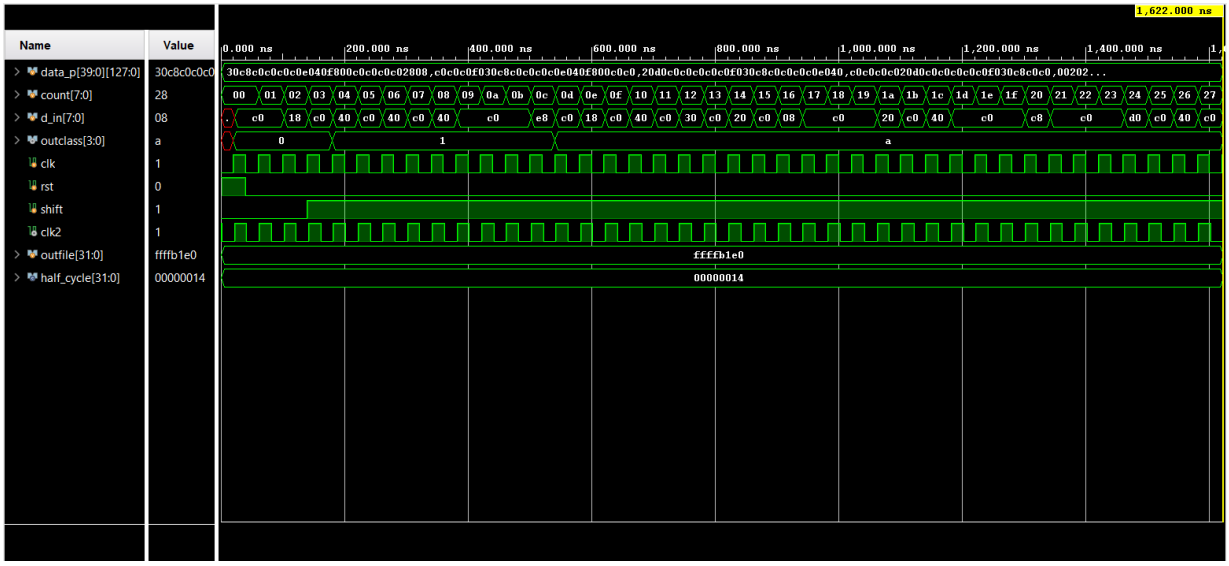


6.3 Output Node Register Design

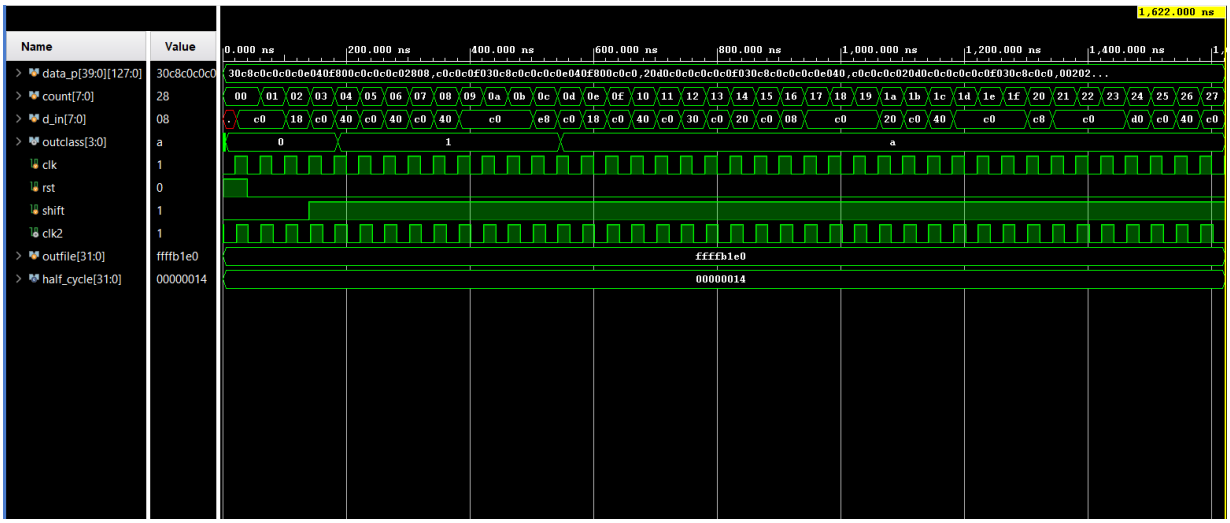
6.3.1 RTL Analysis Schematic



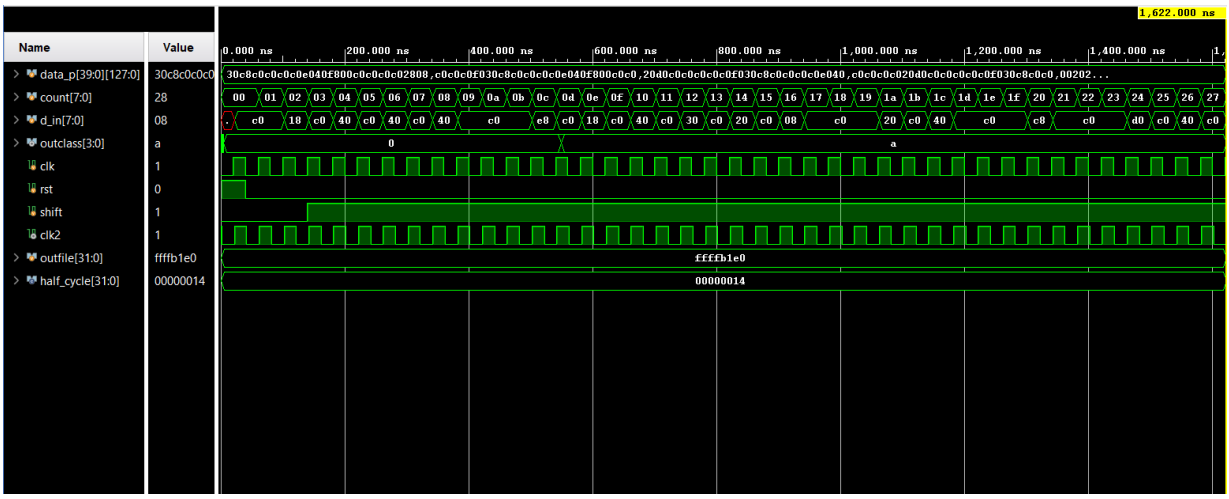
6.3.2 Behavioral Simulation



6.3.3 Post Synthesis Simulation



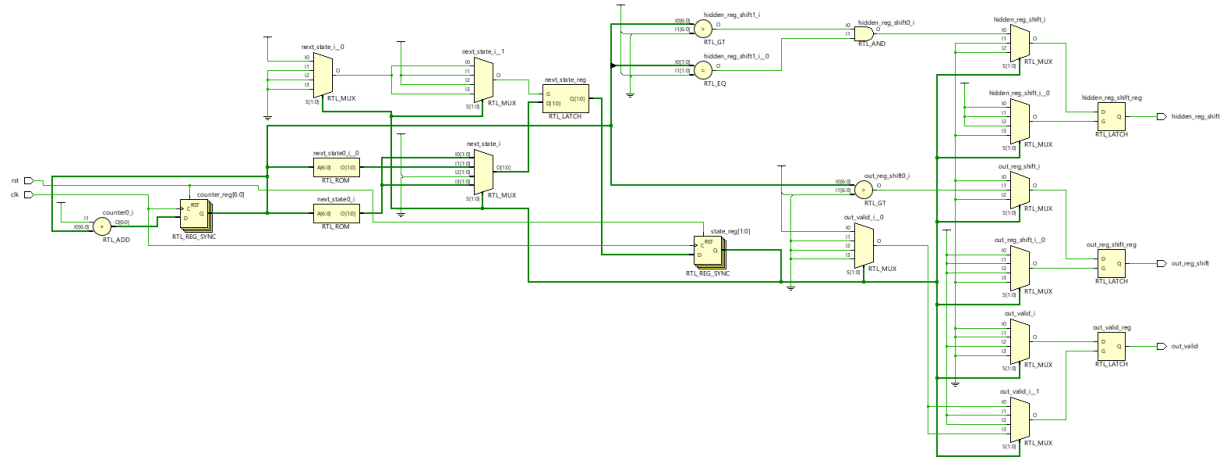
6.3.4 Post Timing Simulation



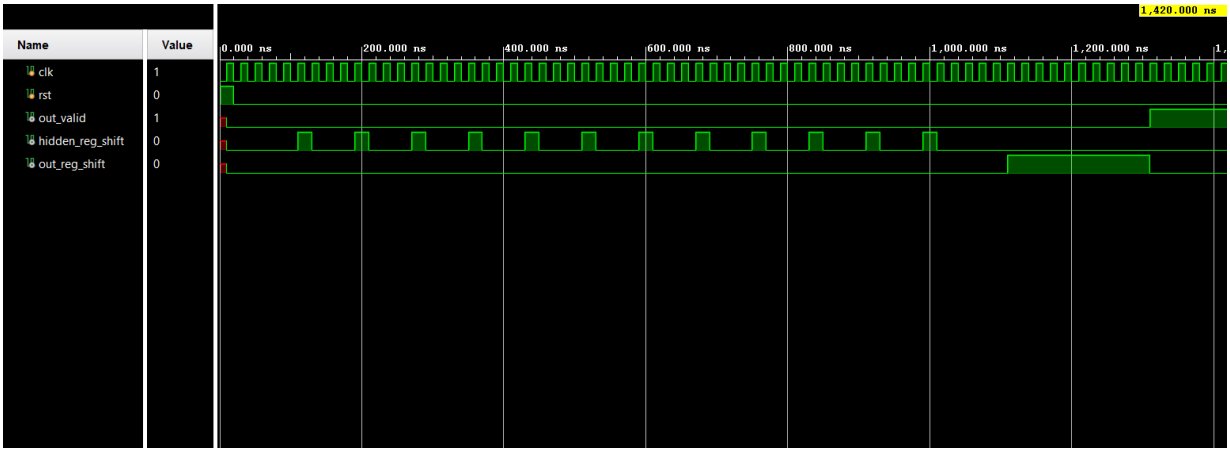
6.4 Controller Design

The complete design is commanded by a controller module which enables and disables the separate stages of the complete system. It has two shift commands that enable the shifting of hidden node register and outnode register. Hidden node register gets shifted every 4 clock cycles in the beginning and once the first layer completes, it gets disabled and it waits for three clock cycles for the mac block to propagate data through the pipeline, and then enables outnode shift for 10 clock cycles. Once that is elapsed, the validity is asserted high to indicate the completion of the process.

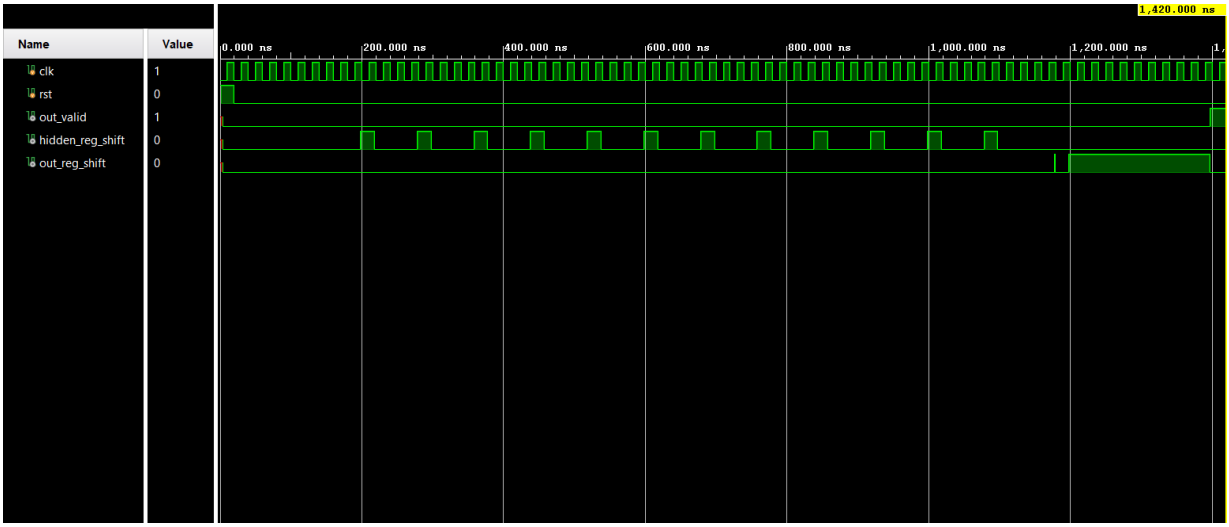
6.4.1 RTL Analysis Schematic



6.4.2 Behavioral Simulation



6.4.3 Post Synthesis Simulation



6.5 Complete System

6.5.1 Structure And Function

The complete system contains the following components chained together as shown in the block diagram

- Node function
- Hidden node register
- Output node function
- Output node register

The node function module will be used to calculate the hidden nodes, and the result of each node will be shifted into the hidden node register, 16 values in total. Once that cycle is completed, the output node function takes data from the hidden node register and calculates the value of each hidden node. These values are then shifted into the output node register and simultaneously scanned for the maximum values. Once the 10 values have been shifted in, the maximum index is available at the output and the valid signal is asserted high.

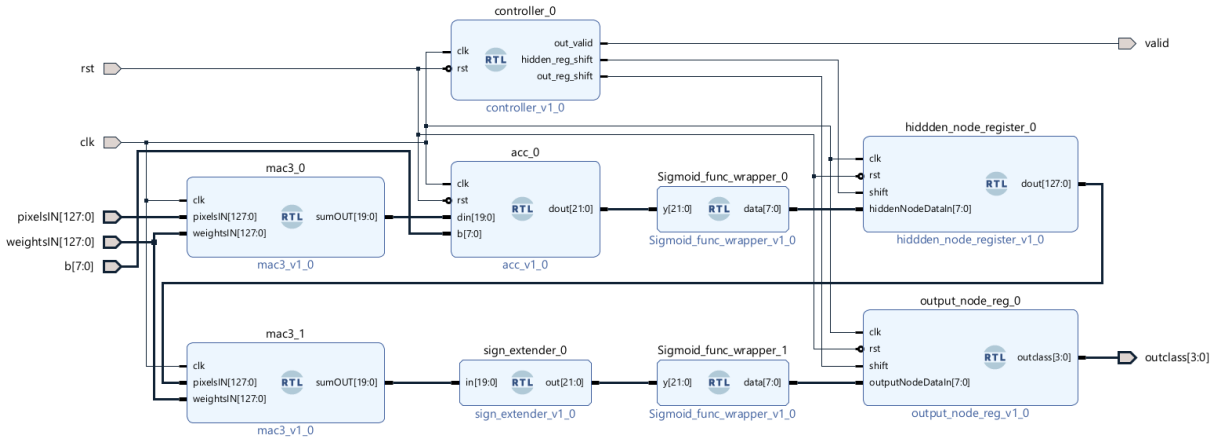
As we already know, the node function has a latency of 7 cycles. 12 passes are required through the module to complete layer one. Since the operation is pipelined, we need 4 * 12 + 3 = 51 clock cycles to finish processing layer one. An additional cycle is allowed for that data to load into the hidden node register. Then the output nodes can be processed at one clock cycle each,

plus the 3 cycles needed for the mac pipeline to get filled. That results in 10 + 3 = 13 clock cycles to process the output layer. Considering the additional cycle for the output register to propagate, the total number of cycles required to process an image is 51 + 1 + 13 + 1 = 66 clock cycles.

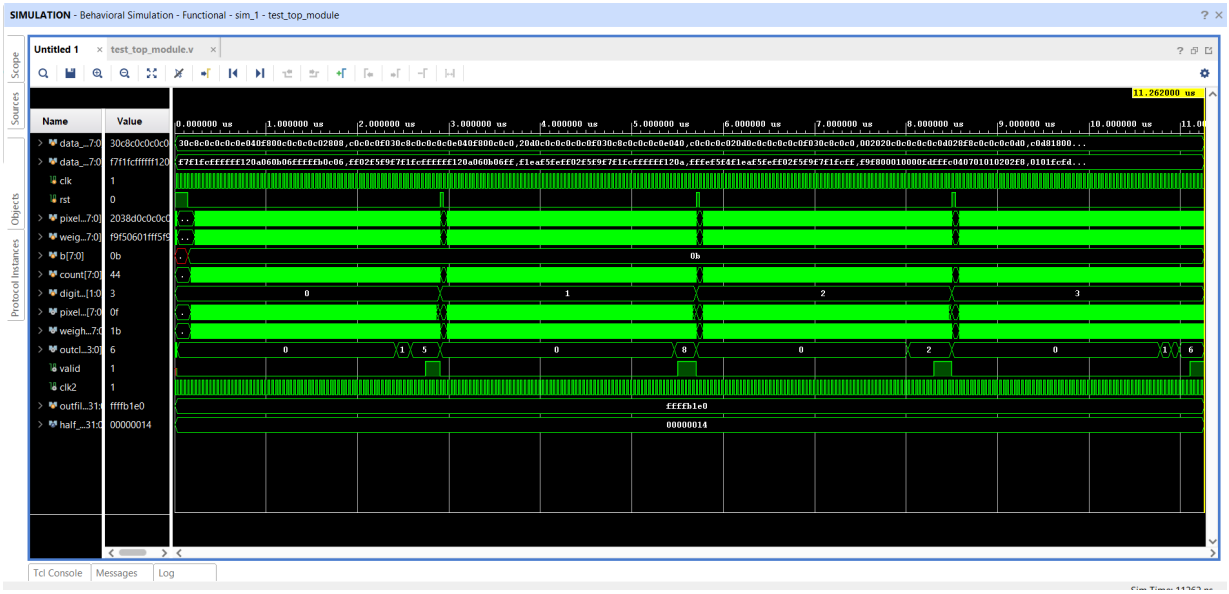
Once an image is processed, the system will hold its state indefinitely. To prepare it for another image, it needs to be reset, clearing all buffers and counter.

As of now, there is no mechanism for addressing the weight or pixel data. It is assumed that the proper data will be at its input at the required time, and it is entirely handled by the testbench. The testbench will appropriately cycle through the pixel data and weight data, feeding them to input ports. The simulation shown in the report is performed using 2 of the provided hex data, and the system identifies it as class 8.

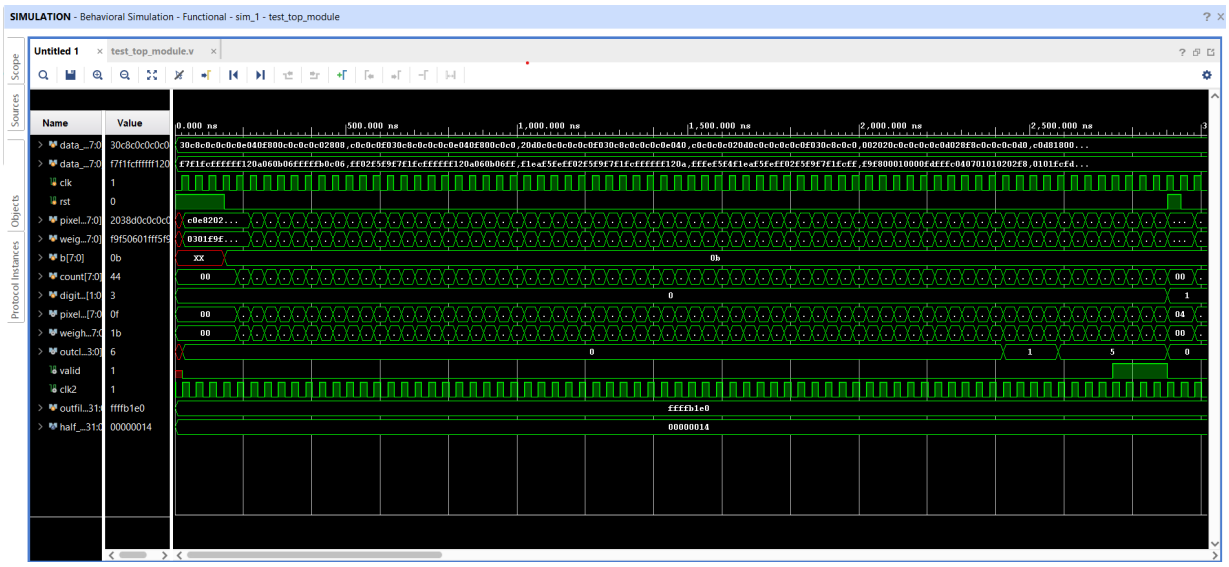
6.5.2 Block Design



6.5.3 Behavioral Simulation – 4 Digits



6.5.4 Behavioral Simulation – 1 Digit



7 CONCLUSION AND IMPROVEMENTS

7.1Shortcomings Of the Current Design

- The main thing that the design lacks is any sort of memory or data fetching mechanism. It depends on the host to feed it data at the correct time and does not provide any feedback. Any misalignment in the data sequencing will seriously affect the accuracy of the system.
- Also, it can only process one node at a time, and therefore lacks parallelism. Since there is no data dependency in fully connected networks in the same layer, this is a huge wasted opportunity.
- Scalability is poor for bigger models. Scaling up the accumulator to handle a longer accumulation cycle will cause the width of the internal lines to explode, resulting in a considerable redesign.
- The precision of the system is poor, since it uses only 8 bits.
- Both node modules will operate at all times, regardless of data validity, causing unnecessary power consumption.

7.2 Possible Improvements

- A memory system with addressing can be implemented such that the controller will sequence through the image by itself. It also eliminates the dependence on external components.
- Multiple instances of the node functions can be spun up and synced together, such that they will process nodes in parallel. This will cut down the execution time by a considerable amount. To facilitate this, the memory modules can be implemented with multiple ports or burst R/W operations.
- Clock gating and module enable/disable logic can be put in place to turn off modules that are performing unnecessary/invalid operations, thereby reducing the power and thermal losses.
- The node functions can be parameterized and modularized, leading to better scalability.
- The parameters can be also extended to port widths, allowing the system to be easily customized to run different models and operate on more precise data.

7.3 Conclusion

The MAC ACC based design is a quite effective approach for tackling the challenge of running a small fully connected network on hardware. It is simple yet takes minimal cycles to process an image and takes far less hardware than trying to 'brute force' the network. It gives good results and can be extended or modified to accommodate a network with a slight variation. Overall, the task can be considered successfully completed with all the milestones.

7.4 Learning Outcomes

Completing the project has provided us with a number of rather important learning outcomes that will be useful throughout our careers for sure. The design, provided in bits and pieces was a great opportunity to learn how to tackle a complex real-world problem. The hardware design techniques such as pipelining and parallel processing were very informative to be seen in practice. Some of the components had to be designed from ground up, and they were challenging, but at the same time informative and exciting. To conclude, completing this project has been a great learning opportunity.