[redacted]

E315 P7 Final Report

Project Statement

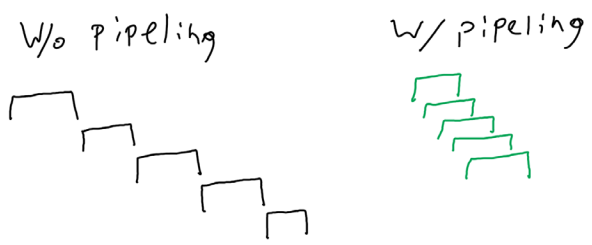
The goal of this project was to implement a faster Dot Product solution in Verilog. The following is a description of what was attempted and accomplished in pursuit of this goal by me, a current undergrad at the Luddy School for Informatics, Computing, and Engineering.

Approaches

The two techniques applied to accelerate the dot product calculation were parallelization and pipelining. In the previous project, a pipelining solution was implemented which accelerated the calculation from 2210 cycles to 230. However, I decided to develop using the given solution with this project which completes the calculation in 237 cycles due to the low difference and robustness of scalability of the solution compared to my own.

As a background, pipelining takes advantage of the property of the floating-point math calculator being able to accept a new calculation every cycle. As such the control logic of dot.sv steps through applying the input to each term in a row of the weights. Decreasing latency up to 7 cycles per additional term in the weight row.

Figure 1: Pipelining Visualization



The next step taken was to parallelize the calculation by splitting the weights between two fmacs. My initial approach was splitting the weights vertically in half due to the theoretically simple input and output control logic needed to coordinate the two. As both instances would be connected and listening to the input at the same time and the output control would multiplex the AXI4 interface output between the first and second instance when transmitting the results.

Figure 2: Split Weights Vertically System Architecture

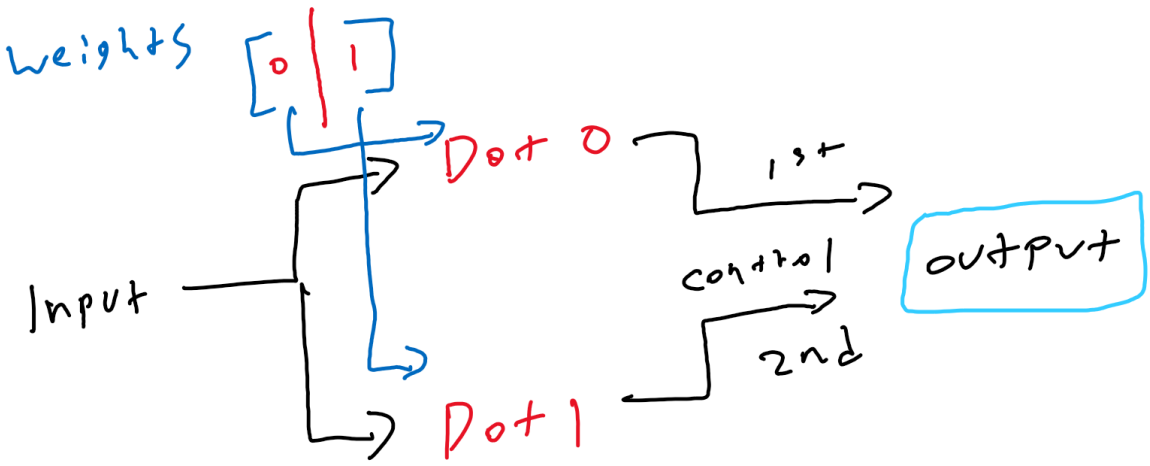
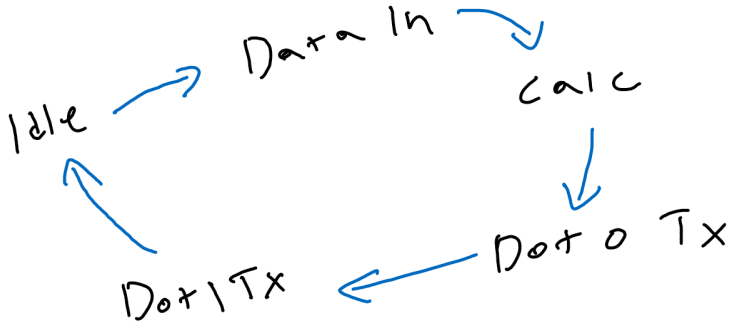


Figure 3: Split Weights Vertically Cycle Diagram



During this attempt, I ran into issues of which I narrowed it down to either the input control or oversight with the multiplex control. I then abandoned this attempt to follow the recommendation of the professor, Andrew Lukefahr of splitting the weights horizontally instead.

This was due to me being unable to find the root cause of the incorrect results from the dot instances.

This approach moved the multiplexing from the output to the input side of the dot instances. While initially I considered splitting horizontally in half, the framework provided split by every other row. After considering the reasoning behind this, it is a better option as it leverages a similar concept to pipelining the input, but in this context between the two instances. This still works as while the order is not sequential, the addition does not care. With this in mind, the state machine after receiving an input ready from the dots would signal the input to start sending data. During this process it alternates the input validity between the two instances of Dot. Once the TLAST is flagged the state machine then connects the TREADY from each instance with the fadd module to then allow the instances to feed to the final output. From there the output of fadd is connected directly to the output interface of the module. The final runtime achieved with this method was 136 cycles although if my own pipelining solution was used the final run time would be 129 cycles.

Figure 4: Split Weights Horizontally System Architecture

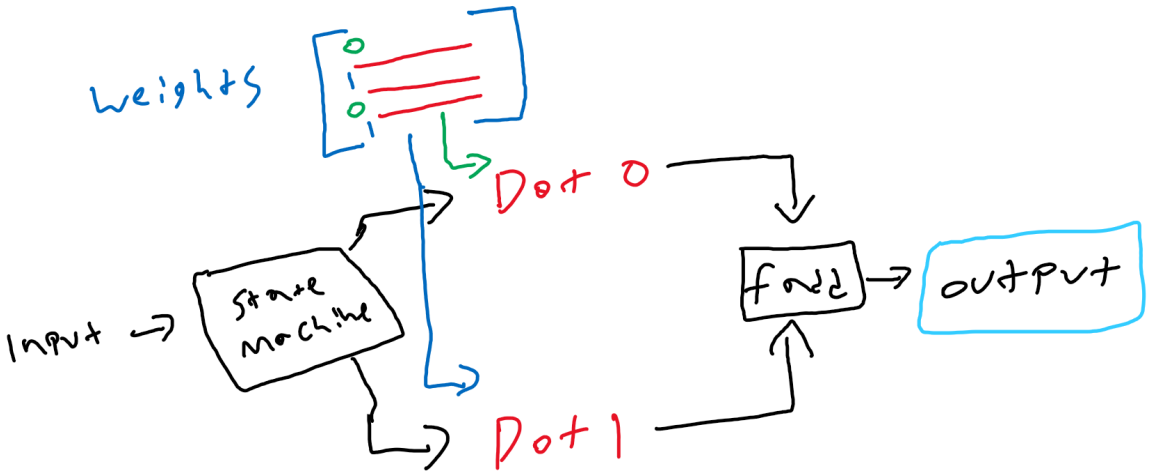
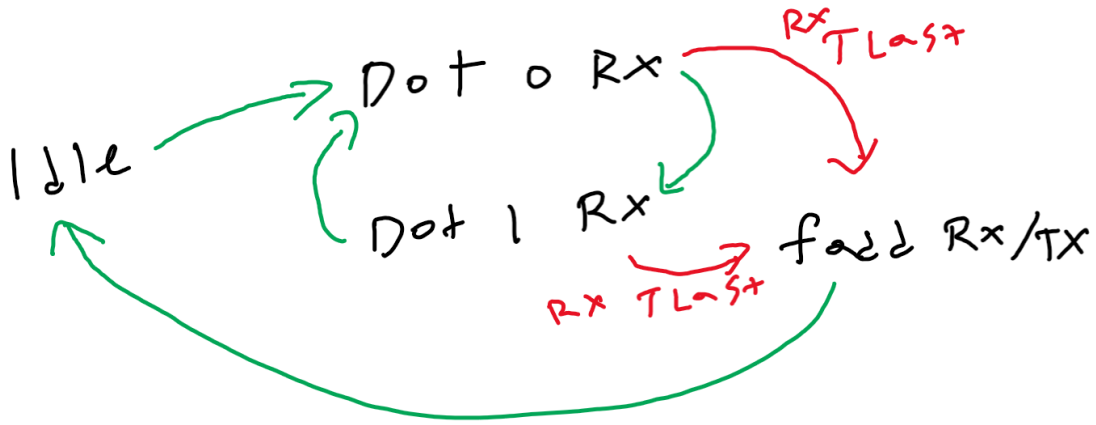


Figure 5: Split Weights Horizontally Cycle Diagram



Discussion

Overall, I believe the reason why my implementation of vertical weight splitting did not work might be due to a misunderstanding of how Vivado handles shared input interfaces. It could also be an oversight with my control logic in providing the input to the instances of dot, resulting in an incorrect calculation. If these issues were addressed, I believe the vertical approach would have at least provided some results. Although, reviewing the architecture it would still lose in speed to the successful approach of horizontal splitting every other row. Due in part to the idea of pipelining the parallelization. As for the difference between my implementation of the dot vs the given. Since the difference between the two is by 7 cycles with and without parallelization. It points to the fact that in my implementation, I assume the 8-cycle latency is absolute and take advantage that the output buffer from the fpm unit will fill up on time. Allowing the output to be relayed back earlier. As such since I do not have a check implemented to see if the buffer is valid this would not be able to handle any errors that may occur or between hardware. Hence the concern of robustness mentioned earlier.

Project Lessons

It can be determined that the learning goals of this project were fulfilled. From this I learned how two methods might be implemented in Vivado even if one was not successful. I also learned how to follow and figure out how to optimize present code in order to accelerate tasks. Finally, I learned from the failed implementation that sometimes a solution might not work even if in theory it makes sense. Of which this was an unexpected challenge where it appeared more straightforward to implement vs the horizontal weight splitting yet resulted in more issues due to inexperience with the language.

In the future, I might explore if increasing the dot instances to 4 would accelerate the calculation further. Possibly reducing by around another half as the first horizontal instance reduced the cycle time.