ECE464 Final Project Extra: Deep Neural Network

Name: **Ethan Chao**
Unityid: **echao2**
StudentID: **200241481**

Delay (ns to run provided example): **2059.2ns** Clock period: **3.9ns** # cycles": **528**	Logic Area (um^2): **10470.2919** Memory: N/A	1/(delay.area) ($ns^{-1}.um^{-2}$) 1/(2059.2*10470.2919) = **4.63812748e-8**
Delay (TA provided example. TA to complete)		1/(delay.area) (TA)

Abstract

The purpose of this project is to implement a fixed size single stage of a convolutional neural network with a ReLu activation function. To implement the convolution, a multiplier and accumulator are needed. The multiplier will have two 8-bit inputs and a 16-bit output while the accumulator is 20-bits wide to prevent overflow. The result of the ReLu function will be saturated to 8 bits so any value greater than 127 will be 127 and any value less than 0 will be 0. The approach to this project was to design a pipelined streaming architecture where the design continues to take in inputs and doesn't stop until the last value of the output matrix is outputted. The design was split into two modules being the datapath and the controller, with the convolution occurring within the datapath module. The design was influenced by an online resource that depicted how a convolution pipelined functioned, and from there it was up to interpretation through Verilog. This pipelining approach was able to achieve a clock period of 3.9 ns, a total delay of 2059.2 ns, a cell area of around 10470 um^2, and a performance score of around $4.64 * 10^{-8}$ ns^{-1} um^{-2}.

ECE464 Final Project Extra: Deep Neural Network

Ethan Chao

Abstract

The purpose of this project is to implement a fixed size single stage of a convolutional neural network with a ReLu activation function. To implement the convolution, a multiplier and accumulator are needed. The multiplier will have two 8-bit inputs and a 16-bit output while the accumulator is 20-bits wide to prevent overflow. The result of the ReLu function will be saturated to 8 bits so any value greater than 127 will be 127 and any value less than 0 will be 0. The approach to this project was to design a pipelined streaming architecture where the design continues to take in inputs and doesn't stop until the last value of the output matrix is outputted. The design was split into two modules being the datapath and the controller, with the convolution occurring within the datapath module. This pipelining approach was able to achieve a clock period of 3.9 ns, a total delay of 2059.2 ns, a cell area of around 10470 um2, and a performance score of around $4.64 * 10^{-8}$ ns-1 um-2.

Introduction

The hardware being designed is a fixed size single stage of a convolutional neural network with a ReLu activation function. In a convolutional neural network layer, the weight matrix is smaller than the input matrix, and an output matrix is produced with a size smaller than the input. In a convolutional stage the weight is multiplied by a subrange of the input to produce the feature map while the output of the sum of weights is passed to a ReLu activation function which saturates them to 8-bits. Figure 1 shows how the convolutional layer functions and Figure 2 shows how the ReLu function works.

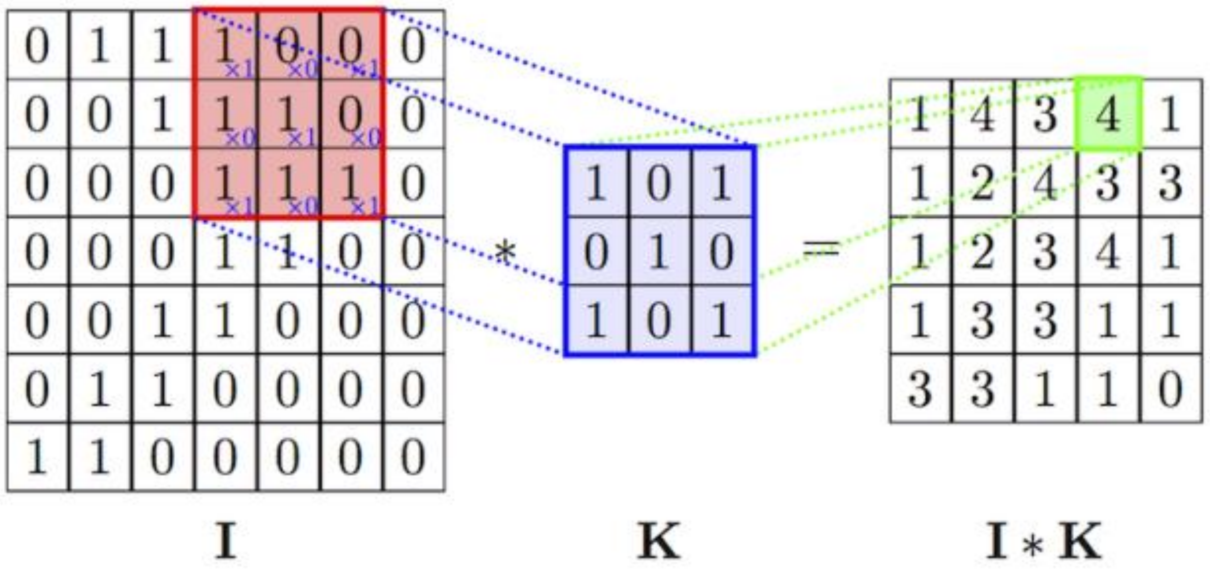

Figure 1: Example Convolution Calculation

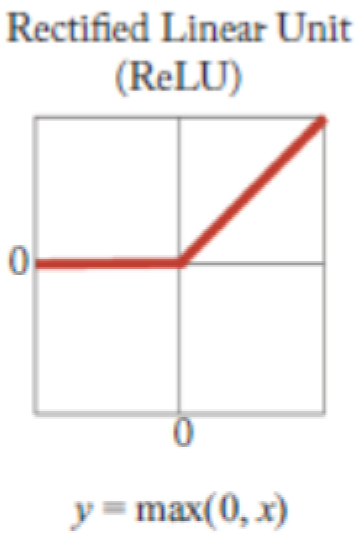

Figure 2: Example ReLu Function

The design featured in this report achieved a clock period of 3.9 ns, 528 total cycles to complete both input files, and a cell area of around 10470 um^2. Using these metrics, the clock delay can be calculated resulting in a total clock delay of 2059.2 ns. The performance can also be calculated resulting in a total performance score of around $4.64 * 10^{-8}$ ns^{-1} um^{-2}. The remainder of this report will discuss details about the design's microarchitecture, interface specification, technical implementation, verification, results achieved, and a conclusion to wrap it up.

Micro-Architecture

The hardware algorithmic approach used in this design features a straightforward circuit path that includes 9 multiply and accumulate (MAC) units with 26 shift registers and a ReLu function at the end to resize it to 8-bits before writing it into the output SRAM. Essentially, the first few states of the controller fill up the weight registers that correspond to each block of the 3x3 weight matrix to store them for continual calculation usage for all the input bits that flow in one after the other. While it fills up a 2-d array for the weights, the input bits also get fed into the data stream continuously (a wire input selects which 8-bits to use first) and starts flowing through the circuit, getting multiplied to the stored weight matrix, accumulating into 20-bit registers to prevent overflow, and moving onto the next MAC unit to be accumulated again until all 9 MAC units and 26 shift registers have been passed through. The ReLu function then resizes it back to 8-bits and the controller finally decides to start writing the output data into the output SRAM (a wire input selects which 8-bits to output first) while also making sure to ignore values that are not valid convolution outputs that appear due to multiplying the very ends of the rows. While these convolution calculations are happening, an internal counter is continuously accumulating and is used to determine which state to change into. It is the sole decider when to start writing output values into the output SRAM, when to stop writing outputs to the output SRAM, when to change to the state that starts the cycle again in the next row, and when to stop and go back to the reset state once no more input data is present and all valid convolution outputs are written to the output SRAM. Figure 3 shows the block diagram for the datapath layout, Figure 4 shows the block diagrams for address and data registers, and Figure 5 shows the diagram for the FSM controller.

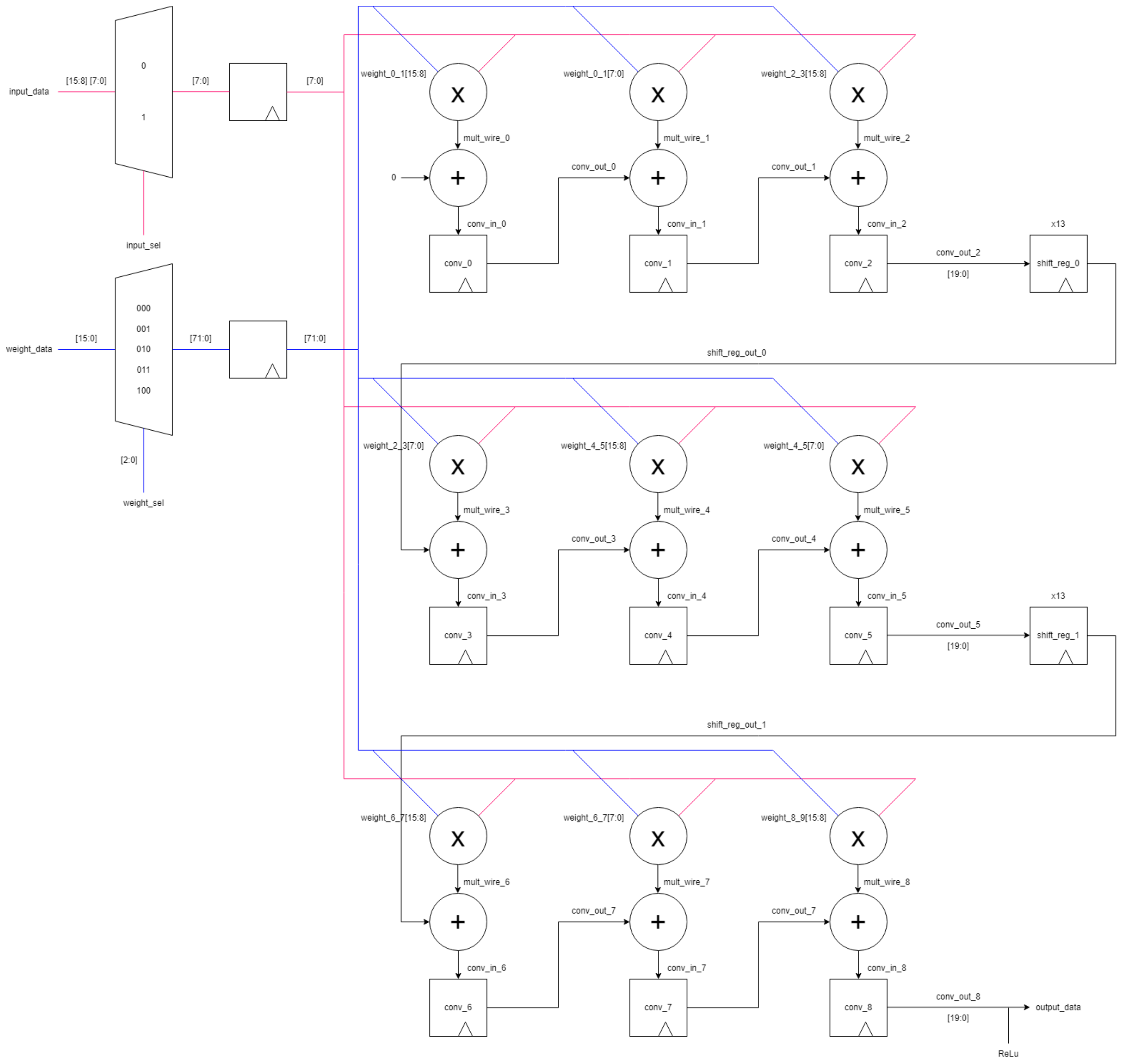

Figure 3: Diagram of Pipeline Datapath

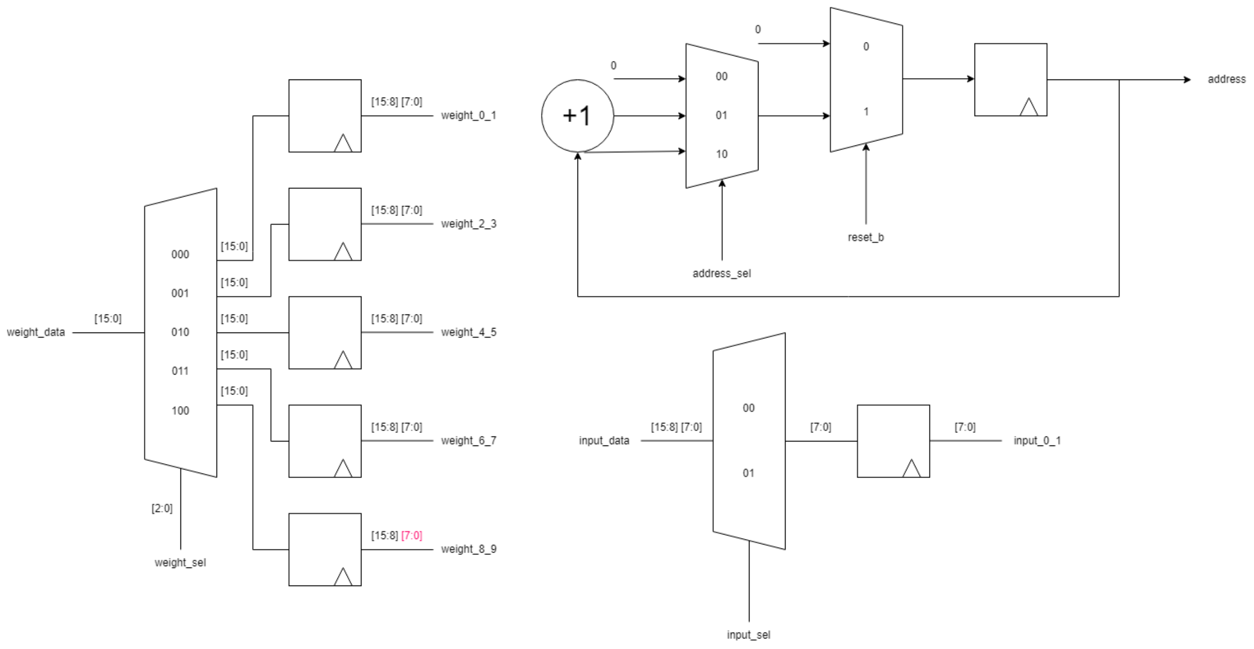

Figure 4: Diagrams of Address/Data Registers

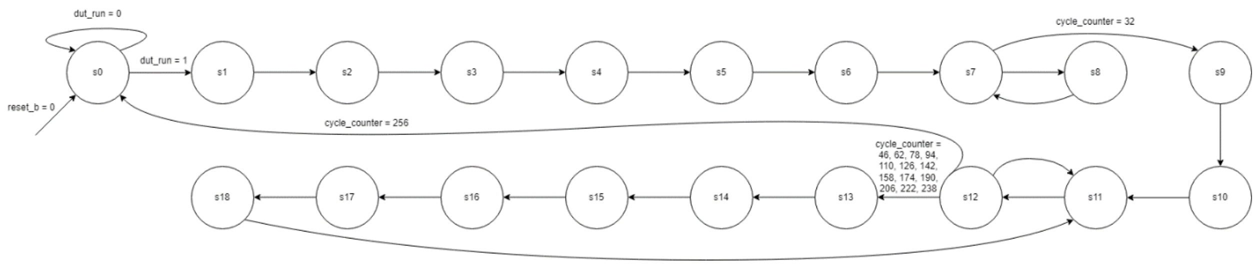

Figure 5: Diagram of FSM Controller

Interface Specification

In this design, the top-level interface is through the module called “MyDesign” which communicates to the datapath and controller with signals and SRAMs provided by the testbench. In short, the top-level interface sends addresses to the input and weight SRAMs and reads in data which is then utilized by the datapath and controller to then be decided by the interface when and where to write the final convolution data to the output SRAM. Table 1 depicts the top-level interface signals used, Figure 5 shows the block diagram for SRAM interfaces, and Figure 6 shows a simplified interface timing diagram.

Table 1: Top-Level Interface Signals

Signal	**Width (bits)**	**Function**
dut_run	[0]	Indicates when to start running through the states

dut_busy	[0]	Indicates whether or not the system is still running
reset_b	[0]	Active low reset
clk	[0]	System clock
input_sram_write_enable	[0]	Determines whether to read or write from the input SRAM
input_sram_read_address	[11:0]	The input address where the design will be reading from
input_sram_read_data	[15:0]	The input data being read into the design
output_sram_write_enable	[0]	Determines whether to read or write from the output SRAM
output_sram_write_address	[11:0]	The output address where the design will be writing to
output_sram_write_data	[15:0]	The output data being written out from the design
weights_sram_write_enable	[0]	Determines whether to read or write from the weight SRAM
weights_sram_read_address	[11:0]	The weight address where the design will be reading from
weights_sram_read_data	[15:0]	The weight data being read into the design
input_sel	[1:0]	Selects which 8-bits from the input data to read into the design
weight_sel	[2:0]	Selects where to store the weight data
output_sel	[1:0]	Selects which 8-bits from the output data to write out of the design
read_address_input_sel	[1:0]	Selects how to manipulate the input read address
write_address_output_sel	[1:0]	Selects how to manipulate the output write address
read_address_weight_sel	[1:0]	Selects how to manipulate the weight read address

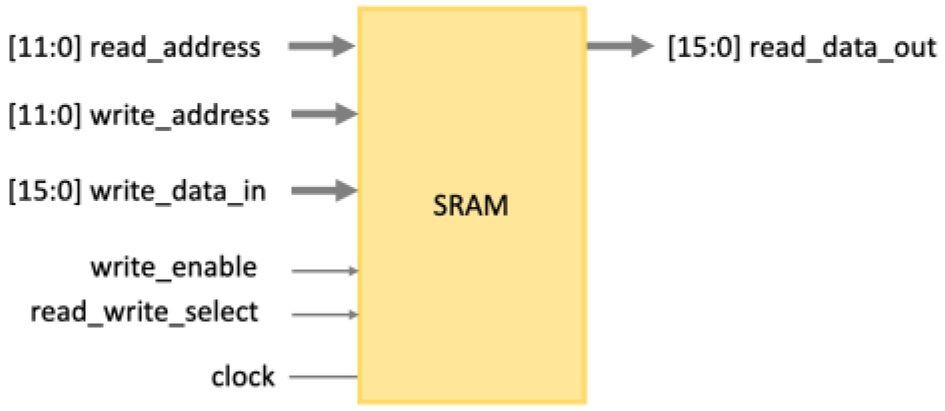

Figure 5: SRAM Interface

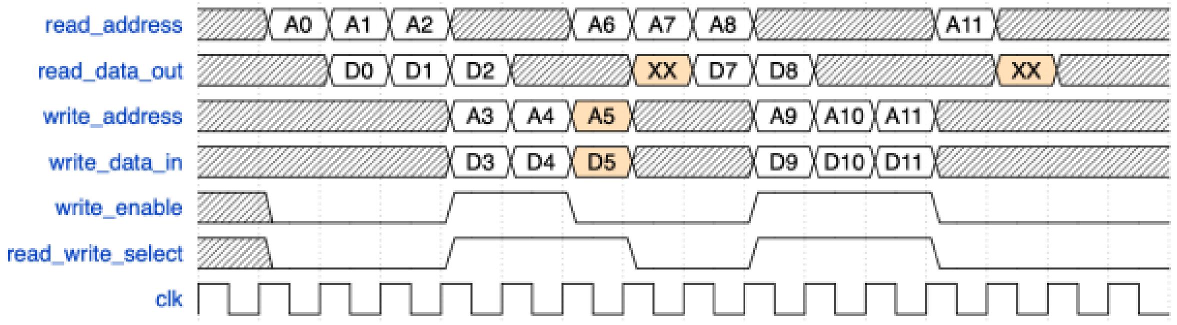

Figure 6: Interface Timing Diagram

Technical Implementation

The design featured in this report has two tiers of hierarchy and can be split into two modules, the datapath and the controller. The datapath contains code relevant to address generation, the output data register, and the pipelined convolution with a ReLu activation function. The address generation section handles the generation of read address for the input and weight SRAMs along with the generation of write address for the output SRAM. The output data register handles when and how the convoluted data is written to the output SRAM. There were originally data registers for the input and weight SRAMs, but the design was shifted to just read the data directly into the convolution. The pipelined convolution takes in 16x16 8-bit inputs and convolves it with a 3x3 8-bit weight matrix to result in a 14x14 8-bit output. Since this is a pipelined convolution, it contains nine multiply and accumulate units and since this is a 16x16 input matrix, there would be 26 shift registers required to shift valid convolution data down to the 3rd row of the mapped weight matrix. At the end of the convolution engine, there is a ReLu function that continuously resizes it back into 8 bits that then gets fed to the output data register to be stored and written when decided by the controller. The controller is there to control the dataflow through the datapath, and this is implemented by a finite state machine with a total of 19 states. In this case, a simple method of using a cycle counter was utilized to determine how the states flowed to one another.

Verification

There were essentially two stages used to ensure that the design I have created would function as intended along with properly synthesizing into a standard cell ASIC. The first stage of the verification process was to simulate the Verilog code using the provided test benches and sample data files in ModelSim. 2 different data sets were provided (input_0 and input_1) along with the needed signals to test the Verilog code and compare it with the golden outputs (golden_outputs_464) to see how many results matched. The testbench first performs the convolution on the data in input_0 and immediately does the same to the data in input_1 right after the data in input_0 completes. Separate results are then shown for each data set depicting the number of correct and matching outputs along with the number of clock cycles it took to complete each set. The second stage of the verification process was to run the code through Synopsys Design Vision software to decide whether or not the design is synthesizable. If there were synthesis errors within the design, the software will classify and show what warnings or errors occurred which could then be fixed. After all notable warnings and errors are fixed, optimizing the design by decrementing the clock period is next on the list. Once everything is done, the reports for timing to see if the design through the chosen clock period meets the slack requirements and the reports for cell area and power throughput are available to be viewed.

Results Achieved

The design featured in this report was able to obtain a clock period of 3.9 ns, 528 total cycles to complete both input files, and a cell area of around 10470 um^2. The clock delay is calculated by multiplying the clock period with the total number of cycles which would amount to a total delay of 2059.2 ns. The performance was calculated by taking the inverse of multiplying clock delay and cell area amounting to a score of around $4.64 * 10^{-8}$ ns^{-1} um^{-2}.

Conclusion

As previously stated, the hardware in this design performs a fixed size single stage of a convolutional neural network with a ReLu activation function. The design takes advantage of pipelined MACs for a faster clock cycle although it has a drawback of a modest increase in area as seen with the 26 shift registers. The results are favorable as there is definitely a tradeoff between clock cycles and cell area but there is still a lot of room for further optimization that could potentially reduce the total cell area, clock cycles, and clock period.

The ultimate goal of this project for me was not just to have functional and synthesizable code, but rather to understand and learn the process behind designing from start to finish. The circuitry was designed by hand, coded into RTL, and synthesized to a standard cell ASIC. The hard part from this entire project was the debugging process which is obvious as why there are many verification engineers assigned to one design. Overall, this project was a good learning experience behind the entire process of a design from start to finish and I am satisfied with the results.