Final Project Report: Convolution Engine with Rectified Linear Output

Scott Burnett

Name: Scott Burnett Unityid: srburne2 StudentID: 200246143

Delay (ns to run provided provided example): 1,980 ns Clock period: 5 ns # cycles: 396	Logic Area: (um^2) 5,311 μm^2 Memory Area: 871 μm^2 Total Area: 6,182 μm^2	1/(delay.area) ($ns^{-1}.um^{-2}$) $8.17 * 10^{-8}$ $ns^{-1}um^{-2}$
Delay (TA provided example. TA to complete)		1/(delay.area) (TA)

Abstract

The circuit computes the convolution of a 16x16 input in SRAM with a 3x3 weight matrix in SRAM, producing a 14x14 output in SRAM.

The output of the convolution can be interpreted as a feature map, with each pixel representing how strongly that section of the input image matches the weight kernel's feature pattern.

The design is pipelined to effectively multiply and accumulate 9 times, and repeat this 196 times, while simultaneously loading data to and from SRAM. This is all pipelined to occur in 396 clock cycles. The total power of the design is 501μW, and the total area of the design is 6,182 μm^2.

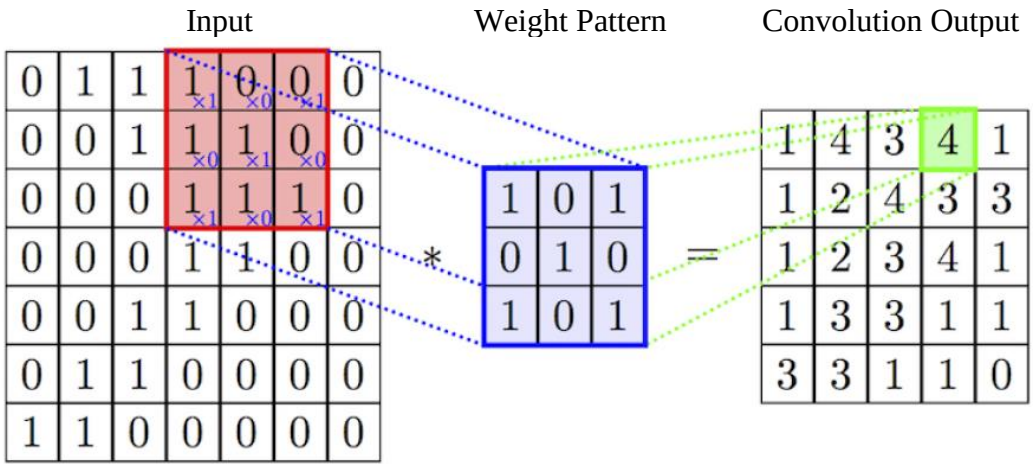
- Each address in SRAM memory stores a 16-bit word, and each word represents two 8-bit cells of the input, weight, or output matrix.
- The circuit implements a RELU clamp on the convolution output, to saturate at 0 and 127.

Convolution Engine with Rectified Linear Output

Scott Burnett

1. Introduction

I designed a convolution engine with a rectified linear output (RELU). In other words, the circuit takes an input image and compares each section of that input image to a pattern image:



The circuit stores the degree to which the input image matches the weight pattern image into an output image.

There are two key inputs to the convolution engine:

1. Input image, a 2D byte array in SRAM
2. Weight pattern, a 2D byte array in SRAM

There is one key output of the convolution engine:

1. The convolution of the input image and the weight pattern. This is a 2D byte array in SRAM

The value in the convolution output represents how much that same section in the input image matches the weight pattern.

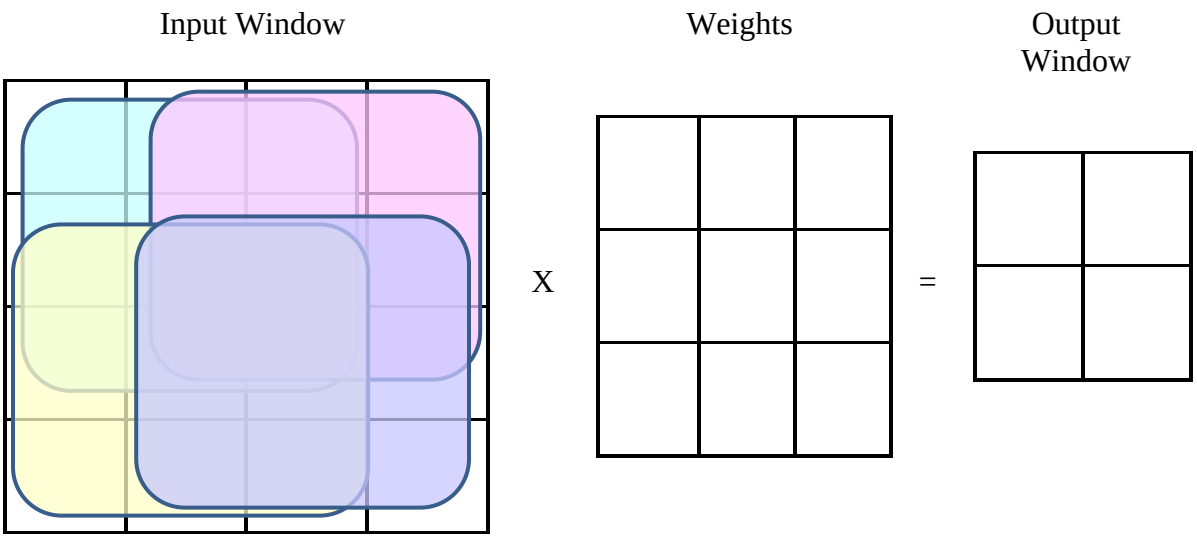
Summary: Results Achieved

The circuit computes the convolution of a 16 by 16 matrix and a 3 by 3 matrix in 396 clock cycles. The circuit can run at a speed up to 200Mhz, at which the calculation takes 1.98μs to complete. The total cell area is 6182 μm^2.

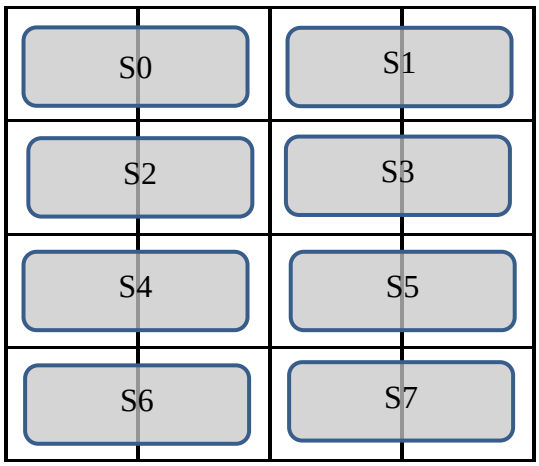
Report Structure

2. Micro-Architecture

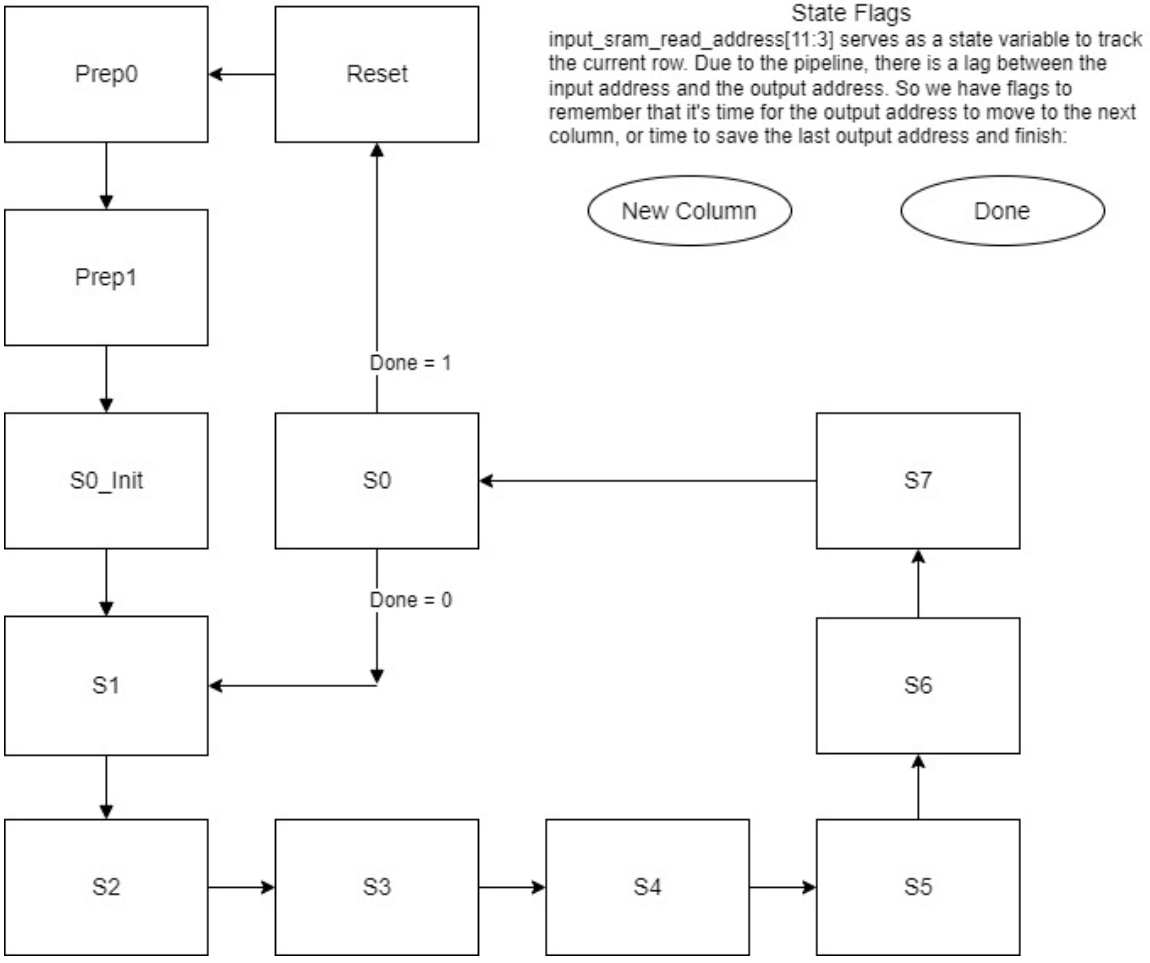
Hardware Algorithm



Reading the Input Window from SRAM throughout 8 states



State Machine



3. Interface Specification

Inputs and Outputs

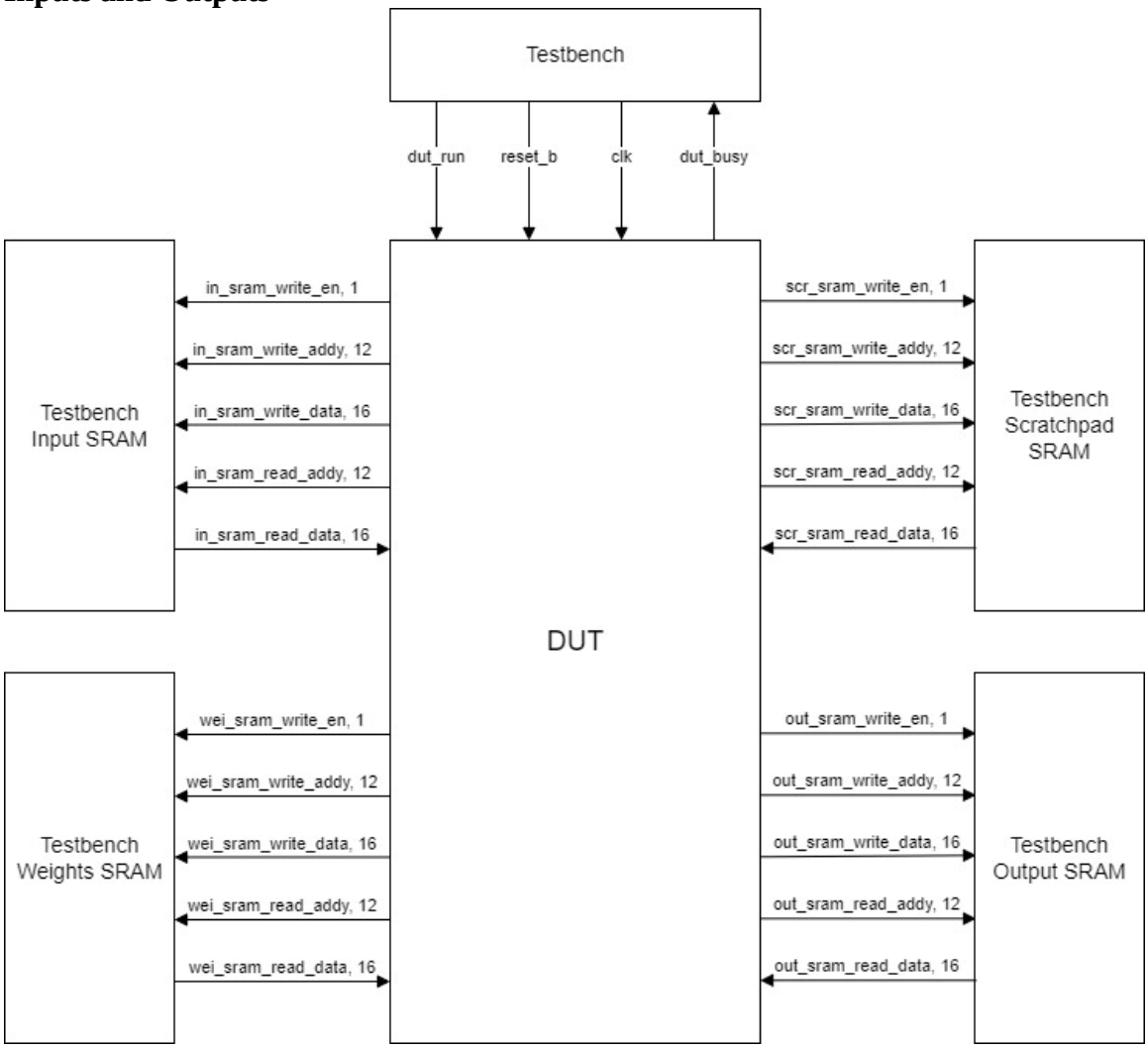


Table of signals, signal widths, and signal functions

Signal Category	Signal Name	Bit Width	Signal Function
Testbench Signals	Dut_run	1	Input. Tells the DUT to start performing the convolution. The input and weights SRAM should be filled with the required input data before raising dut_run high
	Reset_b	1	Input. Active low reset signal
	Clk	1	Input. Clock, rising edge triggered
	Dut_busy	1	Output. DUT raises this signal high when DUT starts computing the convolution. Once DUT has written the convolution result to SRAM, DUT sets this signal low. Low when idle, high when running.
SRAM Write Enable Signals	input_sram_write_enable	1	Output. If high: After the edge of the next clock cycle, write_data will be written to write_addresss If low: In the next clock cycle, the data in read_address during this clock cycle will show up on read_data
	output_sram_write_enable		
	scratchpad_sram_write_enable		
	weights_sram_write_enable		
SRAM Write Address Signals	input_sram_write_addresss	12	Output. The address that the current write_data will be written to, after the next rising edge of the clock
	output_sram_write_addresss		
	scratchpad_sram_write_addresss		
	weights_sram_write_addresss		
SRAM Write Data Signals	input_sram_write_data	16	Output. The data that will write to the current write_address, after the
	output_sram_write_data		
	scratchpad_sram_write_data		

	weights_sram_write_data		next rising edge of the clock
SRAM Read Address Signals	input_sram_read_address	12	Output. The address whose data will show up on read_data during the next clock cycle
	output_sram_read_address		
	scratchpad_sram_read_address		
	weights_sram_read_address		
SRAM Read Data Signals	input_sram_read_data	16	Input. The data located in the read_address of the previous clock cycle
	output_sram_read_data		
	scratchpad_sram_read_data		
	weights_sram_read_data		

Timing Description

1. The testbench raises dut_run high
2. The device will raise dut_busy high on the next clock cycle
3. The device will lower dut_busy once the result is computed and written to SRAM

4. Technical Implementation

High Level Modeling

I wrote a Python script to verify my understanding of the project specifications. The script reads the input .dat files and calculates a handful of the outputs. Through comparing the script outputs to the expected outputs, I confirmed that my understanding of the project specification is correct. Getting to this point took me a few iterations of troubleshooting,

The Python script cleared up a few questions that I had surrounding:

- Organization of the memory words into bytes
- Properly interpreting this data as being signed
- The RELU clamping at 0 and 127

Design Hierarchy

The only hierarchy in the design is the RELU sub-module. This allowed the design to instantiate the RELU module rather than copy-pasting the RELU code block.

Implementation Details: Pipelining

The design implements a multi-step process to be performed on each input from a data stream. Pipelining streamlines this process. Pipelining generates a fresh output of a multi-step process every clock cycle.

First, let's view the pipelining that occurs in a single clock cycle:
Take a 4-step process. The design will compute the following in a single clock cycle:

- Step 0 for a process that will finish at time T+3 (process instance 3)
- Step 1 for a process that will finish at time T+2 (process instance 2)
- Step 2 for a process that will finish at time T+1 (process instance 1)
- Step 3 for a process that will finish at time T+0 (process instance 0)

The table above computed 4 different steps at 1 moment in time for 4 different process instances. Now, let's look through the lens of computing 4 different steps at 4 different times for 1 process instance:

The advantage of looking through the lens of computing at 4 different times for the same process instance is that this directly translates into a state sequence:

- We are pipelining a 4-step process that runs in a larger state loop from [0 to N_STATES)
- Let's say the current state is N

Here is the 4-step process being fulfilled over 4 clock cycles for a single process instance:

Step Number	State where Action Occurs	Clock cycle or time when action occurs	Action to be performed on the clock edge following this state
Step 3 Precondition 0	State N-0	Clock Cycle N-0	I0 is required as a precondition for this action (I0 is a piece of input data from the SRAM)
Step 2 Precondition 1	State N-1	Clock Cycle N-1	I0 <= input_sram_read_data
Step 1 Precondition 2	State N-2	Clock Cycle N-2	SRAM updates input_sram_read_data
Step 0 Precondition 3	State N-3	Clock Cycle N-3	input_sram_read_address <= next_input_sram_read_address

Notice how the state number where the action occurs is the same as the clock cycle or time when the action occurs. In other words:

- Precondition X occurs X clock cycles in advance.
- So program precondition X to occur at State (N – X) % N_STATES

5. Verification

- Description of approach used to verify correctness.

The professor provided a verification approach to compare the output of the DUT to its expected output, for a set of given inputs.

The professor provided a testbench called testbench.sv that applies the appropriate stimulus to the DUT, and reads the DUT outputs to determine whether or not the DUT outputs match the expected outputs.

The teacher was kind to provide a makefile to make it easier to run the testbench in terminal. The makefile automatically runs terminal commands with proper settings to simulate the testbench.

6. Results Achieved

- Throughput, area, power/energy (if applicable), etc.

The circuit computes the convolution of a 16 by 16 matrix and a 3 by 3 matrix in 396 clock cycles. The circuit can run at a speed up to 200Mhz, at which this calculation takes 1.98μs to complete. The total cell area is 6182 μm^2.

The total power of the design is 501μW, which is reported in report_power_final.rpt. This number represents the average power during typical operation under the testbench stimulus.

7. Conclusions

- Summary of project and key results

This was a successful project. The professor provided a foundation that allowed me to understand the task at hand. I was able to brainstorm a hardware algorithm to complete the task. I found it valuable to complete the project plan, and invite feedback from the TA's and other members of the class.

The solution is pipelined to simultaneously preform the following tasks:

- Read from memory
- Preform multiply / accumulates
- Write to memory

The project enhanced my understanding of pipelines. Through following the professor's advice about design before coding, timing diagrams, and separating into data path and controller, I now have a solid grasp of the foundations of RTL design in Verilog.

Project Plan

/10

Name: Scott Burnett
Unityid: srburne2
StudentID: 200246143

Summary Risk Plan:

1. Running out of time
2. Not having a working design (spending too much effort on pipelining rather than getting a basic working design)

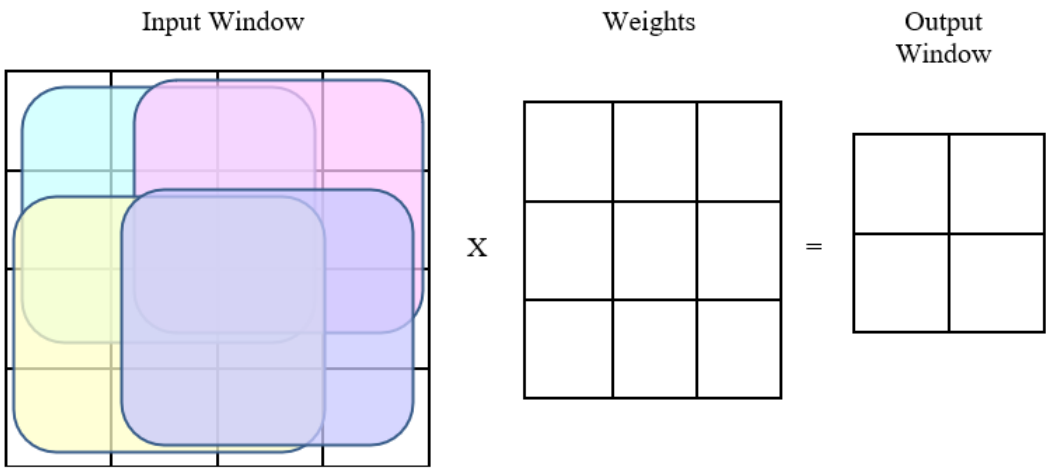
Schedule:
11/14/22 – Complete the Project Plan

11/27/22 – Project Due

Brief Description of Mode of operation, including selected algorithms
Convolution

- Create a 4x4 input sub-window, resulting in a 2x2 output multiply accumulate
- Read the inputs of the 4x4 input sub window in pairs, taking 8 clock cycles
- Requires 6 multipliers
- 8 clock cycles generate 4 outputs

High level sketch. Add details on the following pages if necessary



After calculating the output window for the current input window, the input window will shift down 2 cells to read new input data. Keep in mind that the input window will eventually sweep across the entire 16 by 16 input array. The output window will move in sync with the input window.