Final Project Report First Page. Must match this format (Title)

Name: Jacob Little Unityid: jlittle4 StudentID: 200328953

Delay (ns to run provided example): (4927 + 1903) * 5 ns = 34150 ns Clock period: 5 ns # cycles": 4927 + 1903 = 6830	Logic Area: (um^2) 6748.1540 um^2 Memory: N/A	1/(delay.area) ($ns^{-1}.um^{-2}$) 4.3393×10^{-9} $ns^{-1}.um^{-2}$
Delay (TA provided example. TA to complete)		1/(delay.area) (TA)

Abstract

Abstract should briefly summarize that the hardware function is (remember a future employer might be reading this), what your approach was, and the main results achieved.

This design computes the convolution between an input matrix and a kernel matrix used to stride along the input. In addition, it also pools the convolution result and performs the ReLU function to obtain the output.

The module works by using four multiply accumulations in parallel to calculate all four values necessary for pooling at once to prevent the need to use an additional SRAM for intermediate values.

Matrix Convolution Module for a Neural Network Chip

Jacob Little

1. Introduction

This design computes the convolution between an input matrix of size N that must be a power of 2 and a 3x3 kernel matrix used to stride along the input. It then performs pooling by choosing the maximum value of each 2x2 sub-matrix in the result of the convolution (hereby referred to as the feature map) and finally performs a ReLU function (truncating numbers to a range of 0-127) on the pooled value before writing the output. The module for this design interfaces with three SRAMs. One for input matrices (as well as their size N), one for weights to obtain the kernel for convolution, and finally one for the output matrices. The Pooling and ReLU stages have been swapped from the original project specification because the order of these two operations does not affect the result and it reduces the number of items that need to have the ReLU function applied to them. Another key aspect of this design is that it avoids the use of an additional SRAM to store intermediate values by calculating all four feature map values necessary to perform pooling (and subsequently ReLU) in parallel so that pooling can be performed immediately once all four multiply accumulates have finished. In effect, this means a 4x4 matrix is used to stride along the input.

2. Micro-Architecture

Below is the high-level architectural diagram of the module:

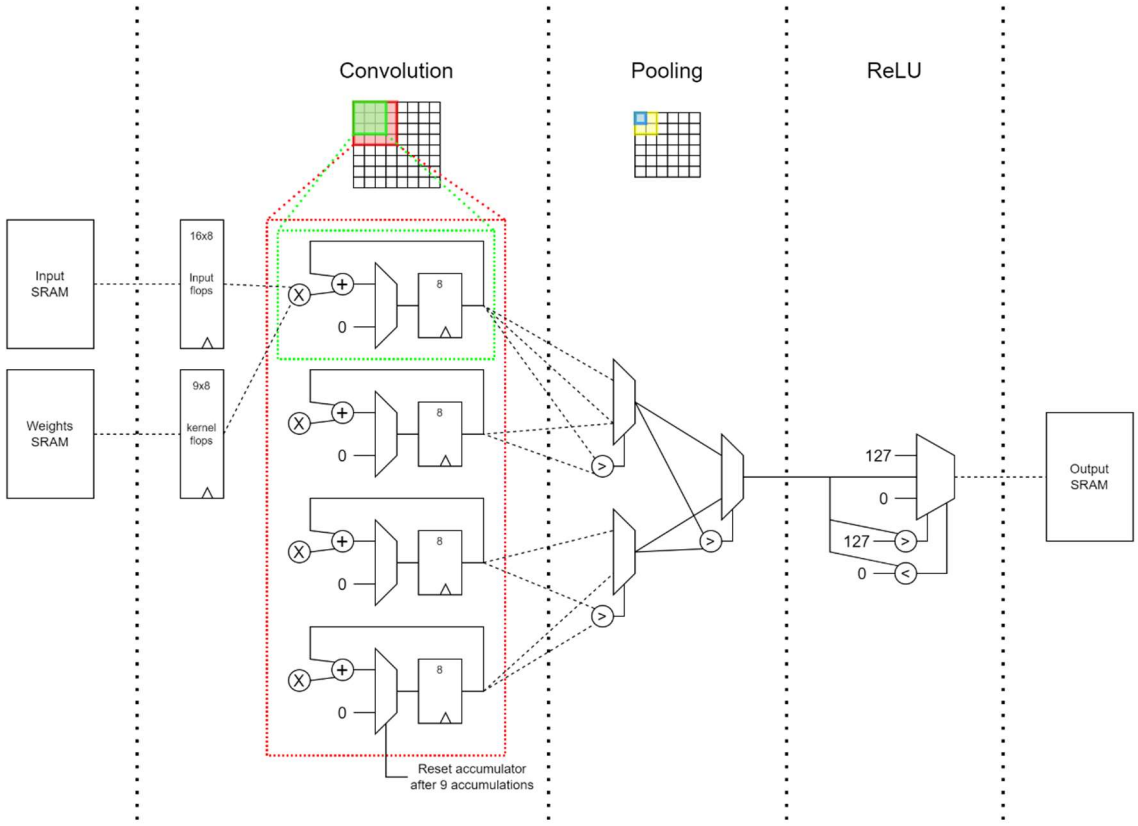

Figure 1: High-Level Architectural Diagram

Values for the input matrix are read from the input SRAM and stored in a 16-word, 8-bit register of flip-flops which represents the stride 4x4 matrix used to calculate the four feature map elements needed for pooling. A 9-word, 8-bit register of flip-flops is also present to store the kernel matrix at the beginning of a set of matrix calculations (this will remain static throughout the run). Four multiply accumulations calculate four feature map elements in parallel, which are then fed into three comparators used to find the maximum of the four (this is what performs pooling) which is immediately followed by two more comparators that enforce ReLU by constraining the result to between 0-127. Once the result is ready, it is written to the output SRAM and the next set 4x4 stride matrix is read from the input SRAM to start the cycle anew.

3. Interface Specification

Separate states are used to read in the kernel and stride input matrix, as well as write to the output matrix. In each state, a counter begins at 0 and counts the number of clock cycles that the state takes to finish. This counter is used to address the input and kernel SRAMS to receive the necessary values from them on the next clock cycle. Addressing the kernel SRAM is very simple since it is only 9 elements long (5 16-bit words) and is only read-in at the beginning of execution. However, addressing the input SRAM is far more difficult because the elements needed at each access vary substantially and are not sequential in the SRAM. To manage this, a register stores a “frame pointer” that points to the first element of the 4x4 stride matrix that needs to be read at each input SRAM access. Offsets from this frame pointer are calculated to address the input SRAM. The output is written to sequentially, so the output address is simply incremented with each value written to it. However, because 8-bit output results need to be stored two at a time in 16-bit words, a register is used to save an output result until a partner result is calculated so both can be written to the output SRAM. Additionally, the last result is written to the output SRAM concatenated with 8 zeroes due to the fact that the output matrix will always have an odd number of elements.

4. Technical Implementation

Below is the finite state machine that controls the module:

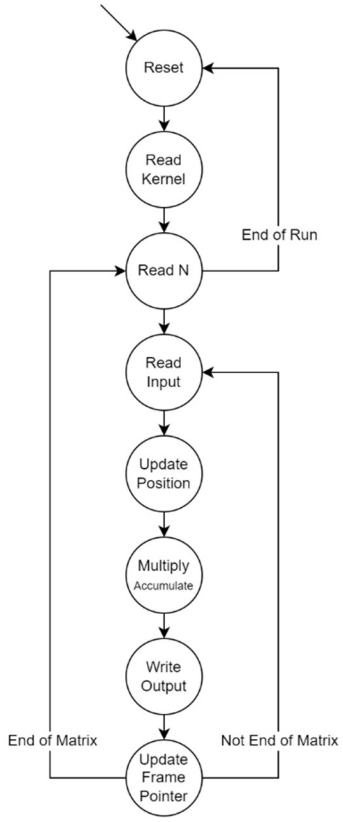

Figure 2: Finite State Machine

As previously stated, some of these states take multiple clock cycles to execute (such as reading values from the SRAMs and multiply accumulation), and the amount of time each state takes is managed by a general-purpose state counter that starts at 0 at the beginning of a new state and increments on each clock edge as long as the state remains the same.

The reset state is the entry point to the state machine and only moves to the next state if the signal to begin execution is received.

The read kernel state reads in the entire kernel matrix into the kernel register.

The read N state reads the size of the matrix from the input SRAM into the N register (in actuality, it reads N right shifted by 1 because that is the only value needed by the implementation which reduces the number of flops needed to store N by one and reduces the logic needed to right shift it later on).

The read input state reads in the input matrix based on the current frame pointer. A key optimization of this module is that while it reads in all 16 values needed for the stride 4x4 matrix on the input when the frame pointer is located at the first column of the input matrix, on subsequent columns, it is able to reuse half of the values by simply shifting them over from the right two columns of the 4x4 stride matrix to the left two columns. This allows it to only have to read in 8 new values instead of 16 on columns other than the first.

The update position state increments two counters that keep track of what row and column of the input matrix we are currently calculating with (in reality, this is actually

the total number of rows and columns of the input matrix right shifted by 1 because we always traverse the input matrix by two elements horizontally and vertically).
The multiply accumulate stage performs the multiply accumulation of four values of the feature map needed for pooling. The output registers of these multiply accumulates are directly connected to digital logic that performs the comparisons and multiplexing necessary to implement pooling and ReLU in a single cycle.
The write output state takes the last value from the digital logic that performs pooling and ReLU after the multiply accumulations have completed and pairs it with another output (either the one that follows it or precedes it) to write both to the output SRAM.

5. Verification

Before implementing the design in Verilog, a high-level design was implemented in C that gave the calculated the output of each stage for verification. This was used extensively when comparing the output of the waveforms produced by Modelsim to compare functionality and find errors in the implementation. Another high-level design was also implemented in C that aimed to verify the correctness of the high-level architectural design by implementing it's general structure and data-path. While this was not used much in developing the Verilog, it was immensely helpful to verify the correctness of architecture before implementing it in Verilog.
The module was designed with simplicity in mind. This is why only one state machine and counter are used. This made it much easier to determine errors in the timing diagram produced by Modelsim and determine that the correctness of previous inputs would likely hold up for future inputs.

6. Results Achieved

While this module leaves some performance on the table by choosing not to overlap states, this is made up for by a reduction in space due to the control being rather simple. The first set of inputs was calculated in 4927 cycles and the second set of inputs was calculated in 1903. For the entire verification run at a clock period of 5 ns, it took 34150 ns to complete both sets of inputs in succession.
The total area of the module is 6748.1540 um^2 with a total of 5895 cells.

7. Conclusions

I believe this design to be substantially efficient. It went through many iterations and design considerations as well as being restructured and re-organized to substantially reduce the complexity. But in the end, it was definitely worth it because the design is very efficient, simple, and includes a lot of clever solutions to difficult problems that seemed near-insurmountable at times. This has been very enlightening to both my experience in RTL and my experience solving interesting and complex problems.