Final Project: FPGA Implementation of LeNet5 Convolutional Neural Network

Bryan Blakeslee

Submission Date: 12-17-18
Reconfigurable Computing
Instructor: Professor Lukowiak
TAs: Andrew Ramsey, Stephanie Soldavini

Introduction

The purpose of this project was to create an implementation of the LeNet5 convolutional neural network in VHDL for instantiation on an FPGA. The network's task was to classify a series of handwritten digits as numbers ranging from zero to nine. The dataset used was the MNIST Handwritten Digit Database. The first step was to architect and train the network's behavioral model in Matlab using the Deep Learning Toolbox. Once trained, the network was synthesized into C code using Matlab Coder. This C code was then run through Vivado's high level synthesis tool to generate the final VHDL model as an IP core. This core was then integrated into a larger interface system, enabling it to interface with a PC over a UART connection. A separate test dataset was then sent to the FPGA for classification. The accuracy was then computed and compared against the behavioral Matlab model.

Background

The first phase of the project consisted of constructing the behavioral model of LeNet 5 in Matlab with the Deep Learning Toolbox. This consisted of several steps. First, the dataset was acquired and unpacked with the assistance of the MNIST Helper library, then stored in a Matlab Datastore object. This dataset consists of a total of 70,000 images of handwritten numbers ranging from zero to nine, split into 60,000 training images and 10,000 testing images, in addition to the ground truth labels. Samples of the training and testing datasets are shown in Figures 1 and 2, respectively.

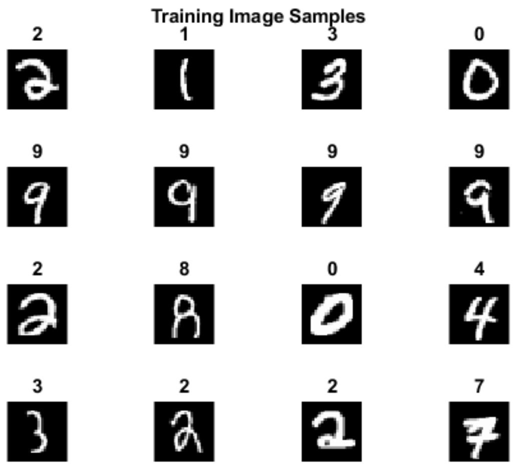


Figure 1: Sample of 16 training images with ground truth labels.

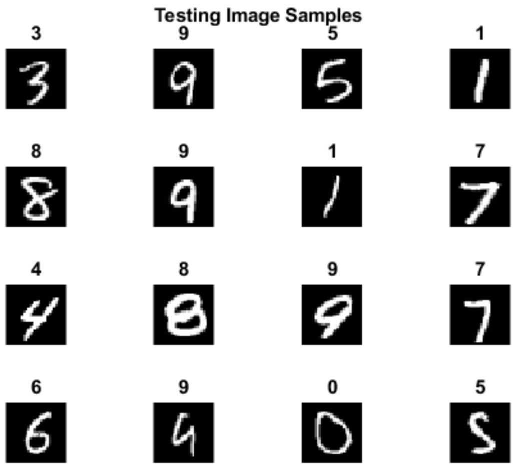


Figure 2: Sample of 16 testing images with ground truth labels.

Next, the network was architected and and trained. Figure 3 shows the architecture of the LeNet5 network.

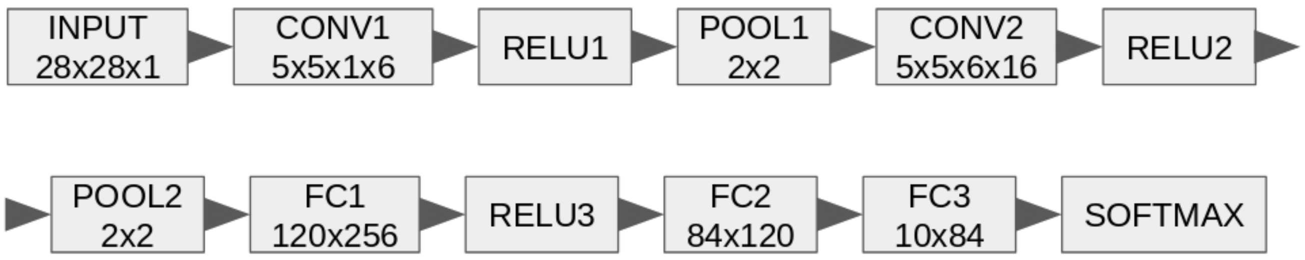


Figure 3: LeNet5 network architecture.

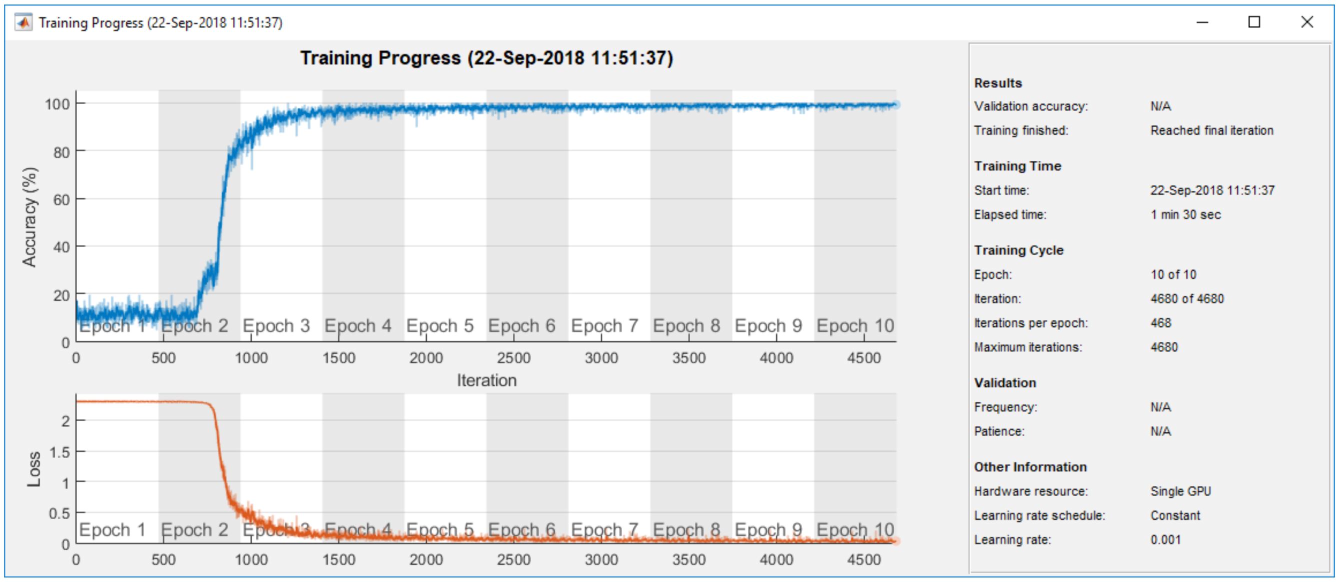
The final set of training hyperparameters are listed in Table 1.

Hyperparameter Name	**Hyperparameter Value**
Optimizer	Stochastic Gradient Descent with Momentum
Number of Training Epochs	10
Learning Rate	0.001
Mini-Batch Size	128

Table 1: Listing of network training hyperparameters.

The optimizer dictates how the chosen cross-entropy loss function is minimized, which guides the network learning. This optimizer is an iterative algorithm that makes small updates along the most negative slope of the loss function, driving it towards a global minimum. The momentum component prevents the optimizer from getting stuck in a local minimum. The number of training epochs is the number of times the entire dataset is applied to the network for training. The learning rate is a scale factor that prevents the optimizer from taking too large of a step, potentially keeping it from achieving a local minimum. The mini-batch size is the number of training samples that the network processes at a time for a cost function update during training.

Once the hyperparameter configuration was complete, the LeNet5 network was trained utilizing a GTX 970 GPU. Several iterations of training were performed in order to appropriately tune the network hyperparameters. The final training accuracy and loss plots are shown as Figure 4.



Results

Validation accuracy:	N/A
Training finished:	Reached final iteration

Training Time

Start time:	22-Sep-2018 11:51:37
Elapsed time:	1 min 30 sec

Training Cycle

Epoch:	10 of 10
Iteration:	4680 of 4680
Iterations per epoch:	468
Maximum iterations:	4680

Validation

Frequency:	N/A
Patience:	N/A

Other Information

Hardware resource:	Single GPU
Learning rate schedule:	Constant
Learning rate:	0.001

Figure 4: Accuracy (top) and loss (bottom) plots.

Once training was complete, the test set was applied to the network and the accuracy computed. The final test accuracy was found to be 98.89%. Sample output of the network classification is shown as Figures 5 and 6.

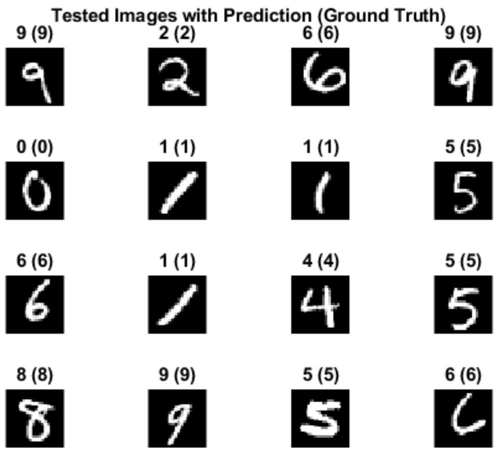


Figure 5: Randomly selected output from the network, parenthetical number is ground truth.

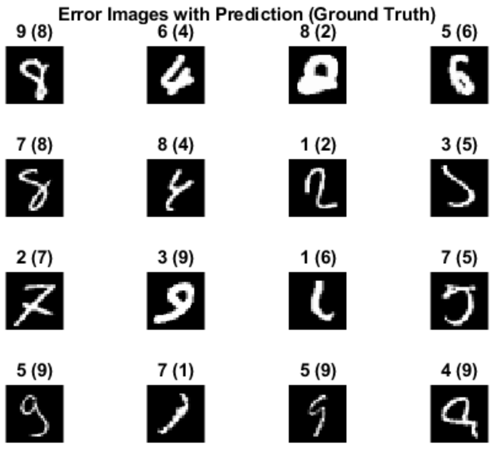


Figure 6: Randomly selected output from incorrectly classified digits, parenthetical number is ground truth.

The next step was to prepare the model for conversion to VHDL. Since the Matlab Deep Learning Toolbox only supports GPU code generation, and not direct C code or HDL generation, a separate pair of synthesis files were created. The consisted of two files, the synthesis testbench and the synthesis model. The synthesis testbench was created by merging the dataset loading code and the network training code. The network weights were then extracted from the final model and declared as global variables. This was done to allow the synthesis model to access these matrices without having to pass them in as function parameters, resulting in a simpler interface. The synthesis model was then evaluated on each sample of the testing dataset and the number of correct and incorrect classifications were recorded. These values were then reported to the user.

The synthesis model consisted of an iterative, CPU-based implementation that only used Matlab functions that were able to be processed by the Matlab Coder system. The synthesis model was implemented with a similar layer structure as the behavioral model. The 2D convolution layers were modeled as a set of nested for loops to implement the shift. The relevant portion of the input was extracted, then point multiplied with the filter. It was then summed together to implement convolution. The Rectified Linear Unit (ReLU) layer was implemented using the max function of zero and the prior layer input. This effectively set all negative numbers to zero. The max pooling layer was also implemented as a set of nested for loops. However, instead of multiplying and summing, the relevant section of the input was extracted and the maximum value obtained. The fully connected layers require special consideration. This is because they were implemented in two different ways. The FC1 layer was actually implemented as convolution with a filter the same size as the input, while the FC2 and FC3 layers were implemented as traditional vector multiplies.

Once the synthesis model was validated as having the same accuracy as the Deep Learning Toolbox model, CPU-based C code was then generated. This was done by specifying the synthesis model as the top level function, with no numeric conversion, causing the generation of code using floating point numbers. The inputs to the model were then defined. The image input to the synthesis function was configured as a 32x32 floating point matrix, while the global variables were configured as single precision matrices. These dimensions are given as Table 2.

Network Matrix	Dimensions
weightsConv1	5x5x1x6
biasConv1	1x1x6
weightsConv2	5x5x6x16
biasConv2	1x1x16
weightsFC1	5x5x16x120
biasFC1	1x1x120
weightsFC2	84x120
biasFC2	84x1
weightsFC3	10x84
biasFC3	10x1

Table 2: Global network weight dimensions.

Once the interface was defined, the actual code generator was configured. In order to reduce the chance of HLS synthesis having issues with more sophisticated C constructs, the generator was configured to generate very minimal C code. Under the Speed tab, the “Saturate on integer overflow” and “Support only purely-integer numbers” options were disabled. This was done to preserve the rollover behavior of the C datatypes and prevent Matlab from inserting any additional saturation or quantization code that may would synthesize correctly. Under the Memory tab, variable-sized arrays were disabled, to prevent any calls to malloc, which is also unsupported for HLS synthesis. The array layout was also set to row-major order to make memory accesses more sequential. Array dimensions were not preserved, allowing arrays to be flattened and reduce the number of for loops that were generated. Under the Debugging tab, run-time error checks were disabled. The code was then generated.

Once generated, the C code was then examined. It was found that while Matlab adhered to the floating point restriction for the input data and the global constants, it generated the internal activation map variables as double precision values. These were manually changed to floats. It was also found that the network coefficients were stored in a separate file as globals, but were transferred into another global location through the use of a set of memcpy calls. These calls were removed and the coefficients manually moved to be local variables inside the synthesis function.

Once these modifications were complete, the Vivado HLS synthesis tool was configured to synthesize VHDL code with a 20ns clock constraint for the chip utilized by the Nexys4 DDR board. This code was then packaged into an IP Core, which was then loaded into Vivado’s IP manager for use in a custom datapath implementation.

The datapath implementation is shown as Figure 7.

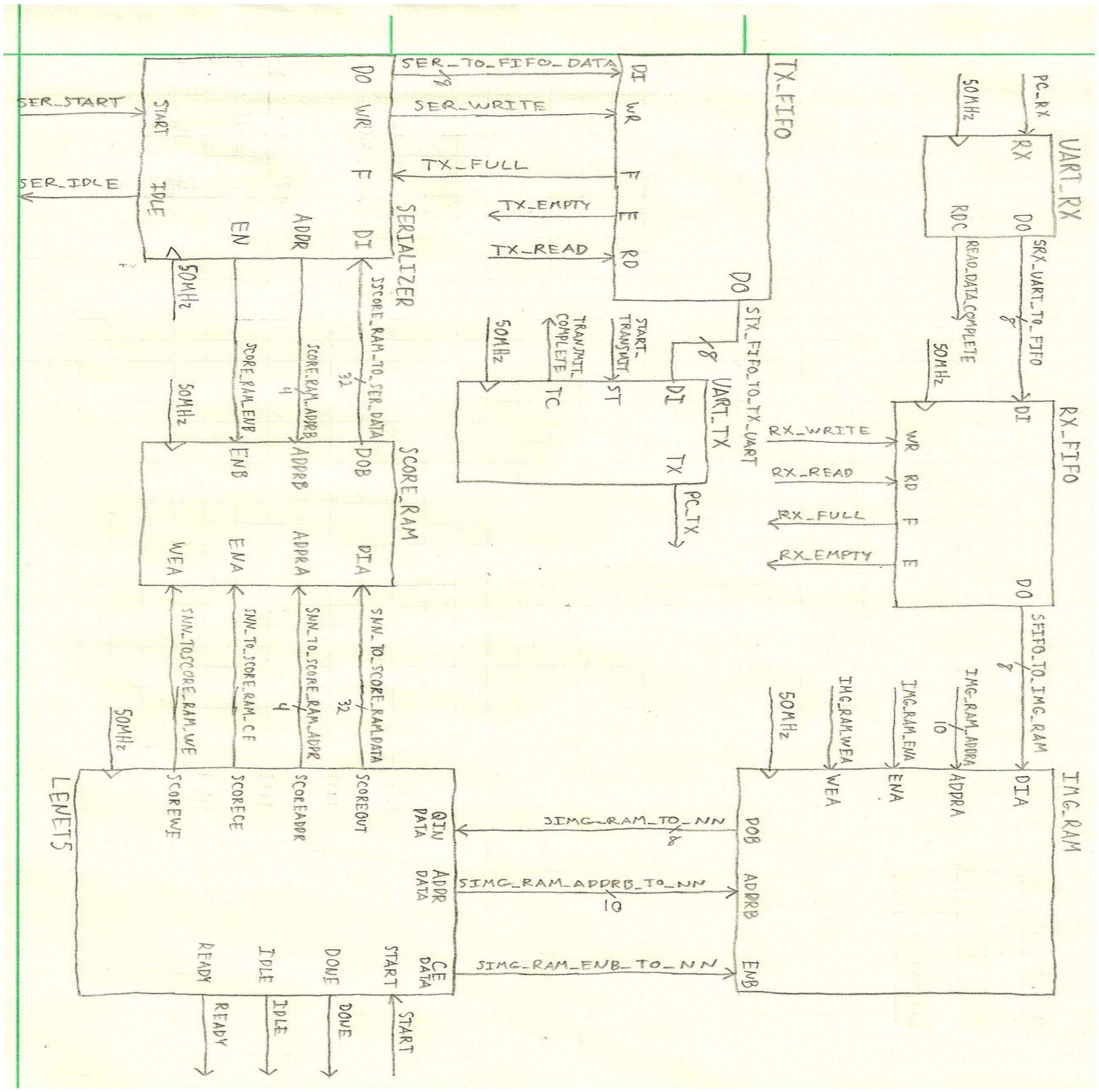


Figure 7: System datapath implementation.

Overall, the UART and FIFO portions of the datapath are identical to that used in the median filtering exercise. The only differences are the presence of the block RAMs, the network IP Core, and the serializer module. The purpose of the block RAM at the input of the network is to hold the image data that is to be classified. This RAM was configured to hold a single 32x32 8-bit image, and as a result, was configured with a depth of 1024 entries and a width of 8-bits each. This RAM was also configured as a dual port device. This allowed both the receive FIFO and the network to access the RAM without the need for any data or address bus switching. (Technically, this block RAM is redundant with the receive FIFO. However, the FIFO was kept, as it simplified interfacing with the UART by allowing the reuse of an additional state machine between the receive UART and the receive FIFO.)

The other block RAM was used in a similar manner, except that it holds the output scores from the network. This RAM was configured with a depth of 16 elements, each 32-bits wide. This is because the network outputs its scores as 32-bit floating point values. The first 10 slots in the RAM each correspond with the possible output classes, e.g.: slot 0x0 corresponds to the zero class, slot 0x1 corresponds to the one class, etc. This RAM was also configured as a dual port RAM, as it allowed the connection of the memory to both the network and the serializer without any bus switching elements.

The serializer is responsible for converting the 32-bit floating point values from the network into 8-bit values that can be enqueued in the transmit FIFO and sent out over the UART. The operation of this block is shown as Figure 8.

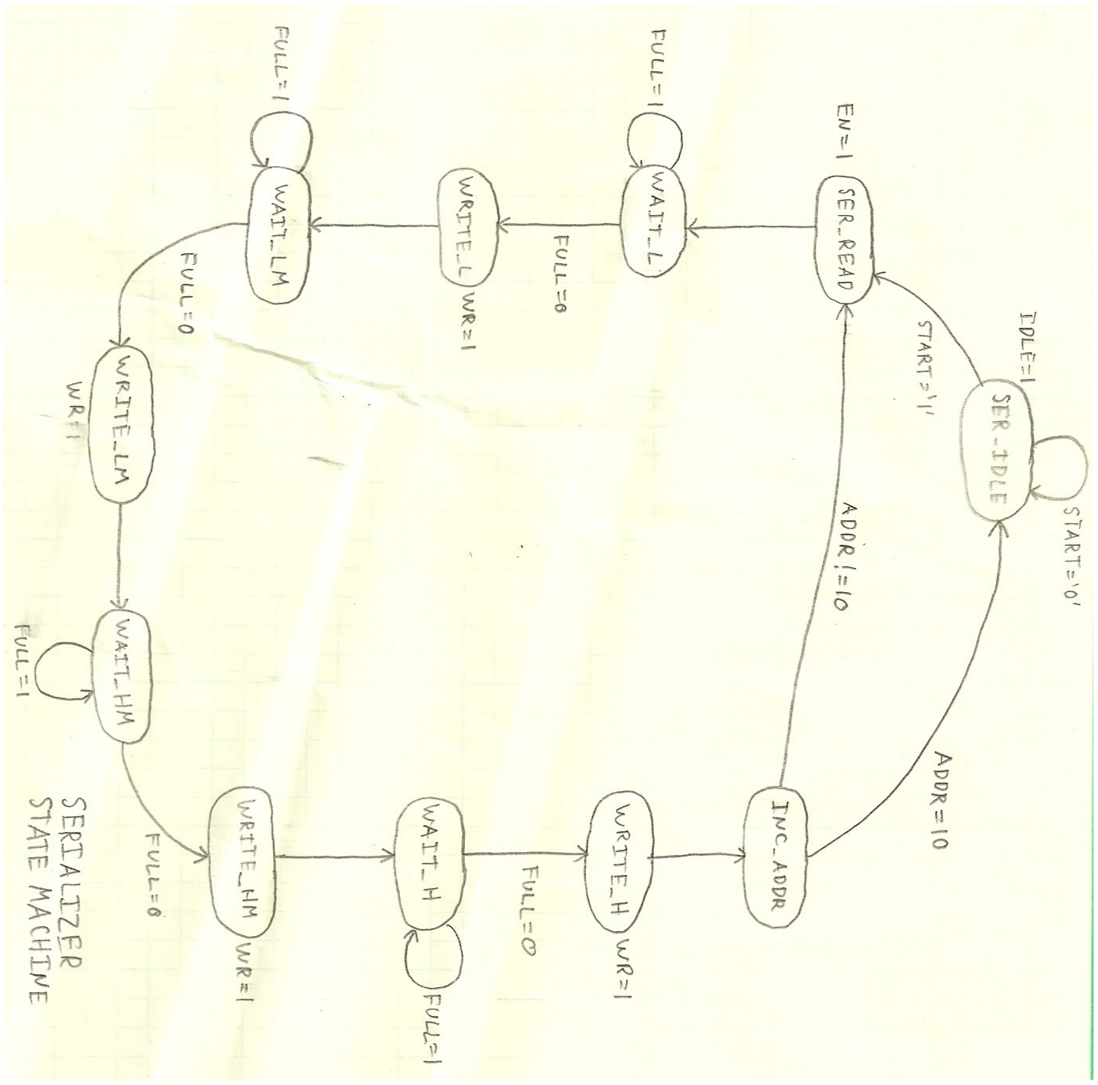


Figure 8: Internal serializer state machine.

It functions by reading a single floating point value in from the score RAM, then writing individual bytes into the transmit FIFO. In order to maintain compatibility with Matlab, this write was performed least significant byte first.

Finally, the LeNet5 network architecture was synthesized as a set of pipelined state machines. Each compute layer was fed by two block memories. One memory would hold the output data from the prior layer, while the other would hold the network coefficients. Both memories would be read by the compute function for the layer, either convolution, ReLU, max pooling, or fully connected, and evaluated. The output would then be stored in another block memory. These pipeline stages were sequenced by another internal state machine. All control logic for the network was synthesized via the Vivado HLS tool.

This datapath was then controlled by the state machine shown in Figure 9.

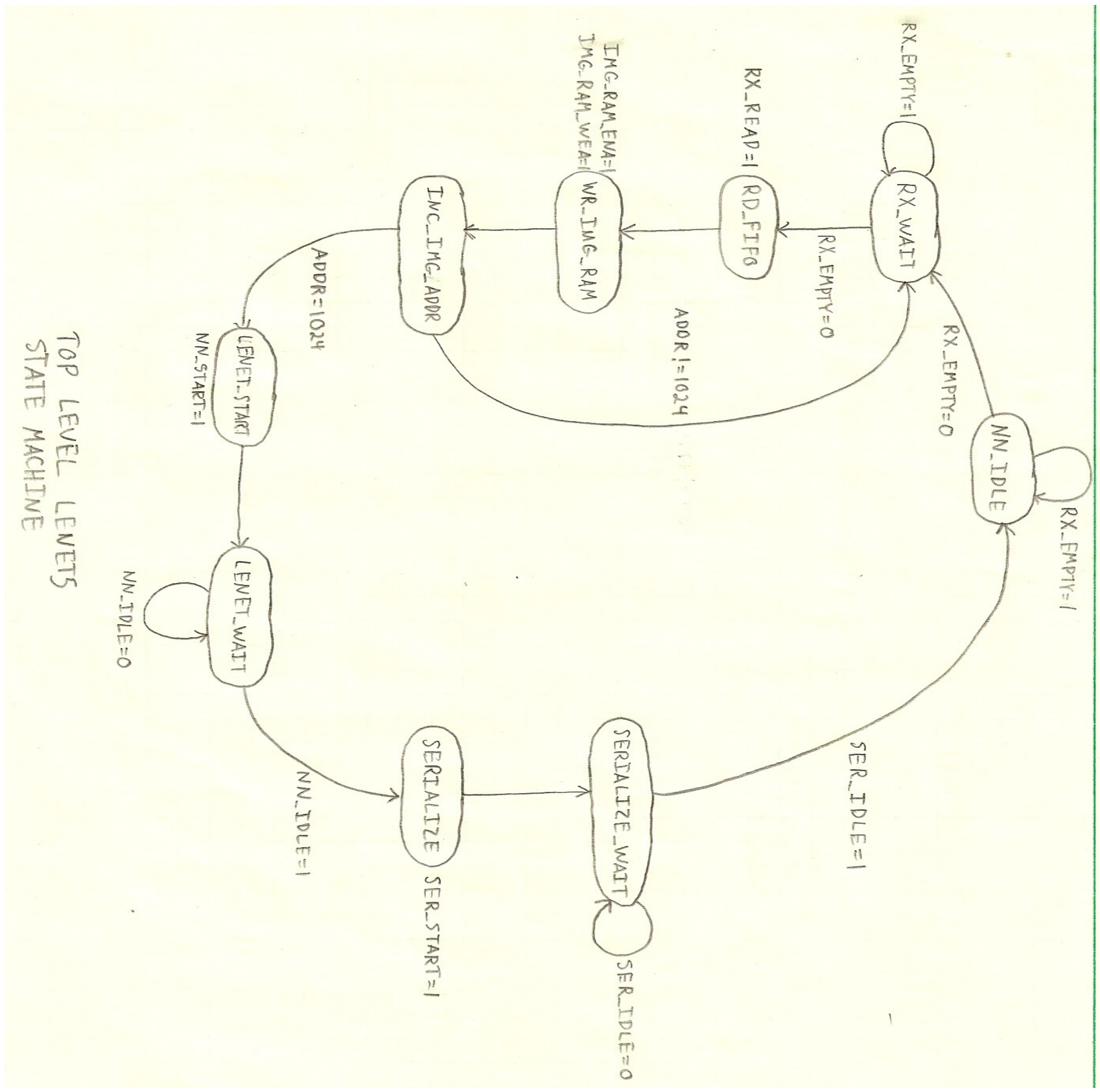


Figure 9: Datapath state machine.

This state machine dequeues received bytes from the receive FIFO and stores it in the image memory. Once 1024 bytes have been stored, the state machine triggers the neural network to begin processing the image. The state machine waits for the network to finish processing, which includes waiting for the scores to be written out to the score RAM. Once the write is complete, the state machine triggers the serializer to enqueue the floating point values into the transmit FIFO. Once the enqueueing is done, the controller returns to the idle state.

The final program written was the Matlab hardware testbench. This testbench reads in the entire test dataset, along with the respective ground truth, before opening a serial port to the FPGA running the network. Each image was vectorized, then written out to the FPGA. The scores were then read into Matlab, and converted to probabilities via the softmax function. The predicted class was then extracted by getting the index of the corresponding highest probability and comparing it with the ground truth. The corresponding counter was then incremented, based on the correctness of the prediction. The final accuracy was then reported to the user, along with periodic reports on specific sample scores.

Results

Figures 10, 11, 12, and 13 show the output of the full system simulation.

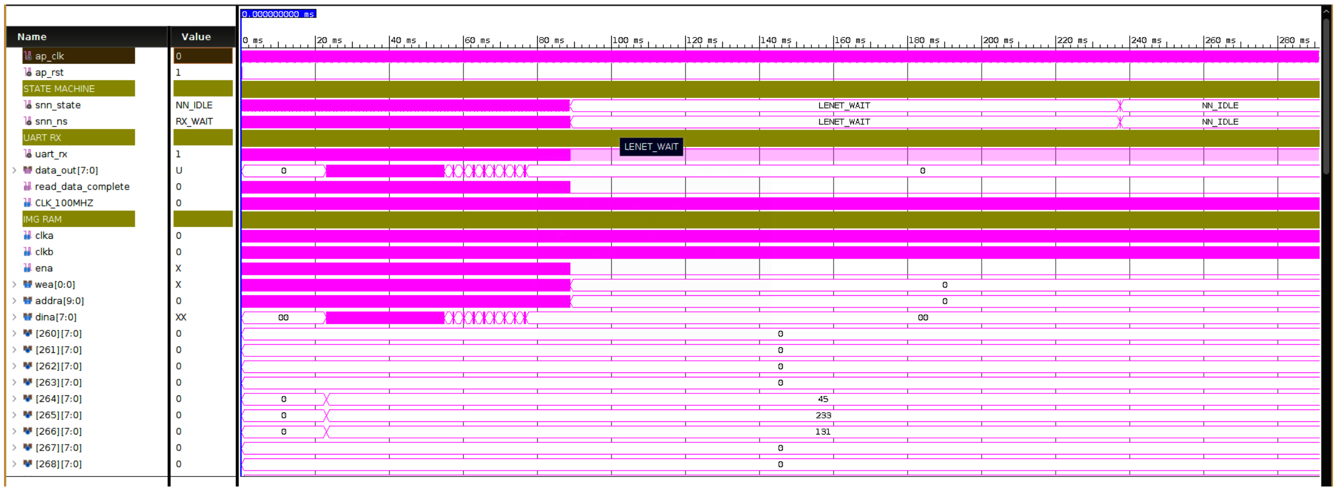


Figure 10: Full system simulation, part 1, state machine and receive UART.

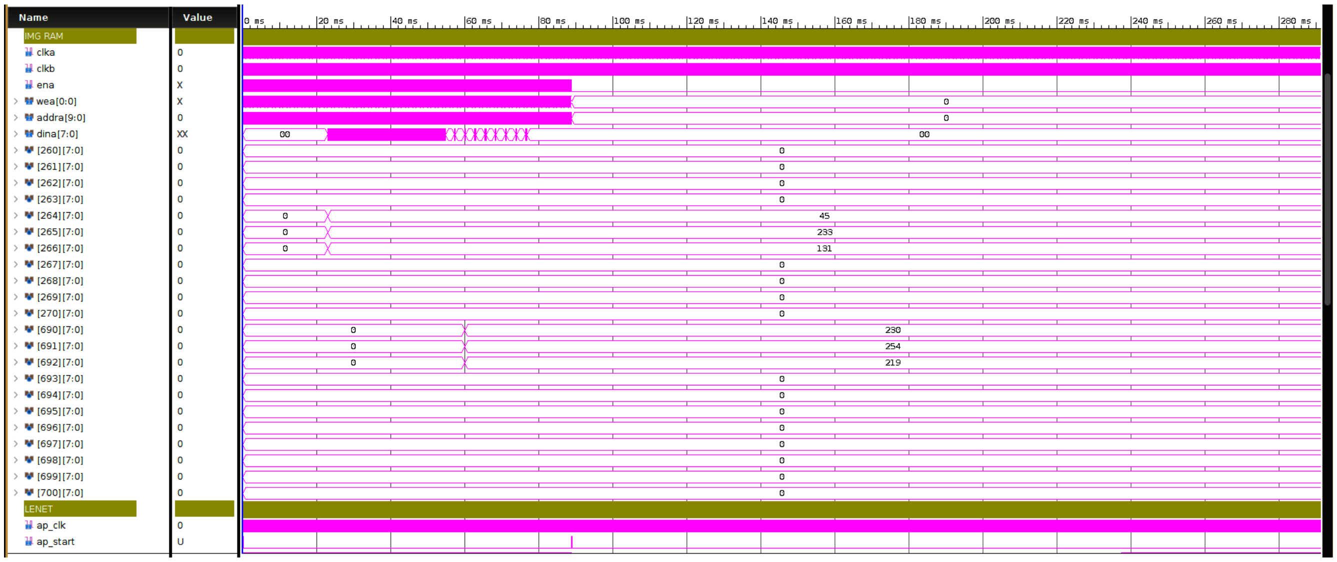


Figure 11: Full system simulation, part 2, selected contents of image RAM.

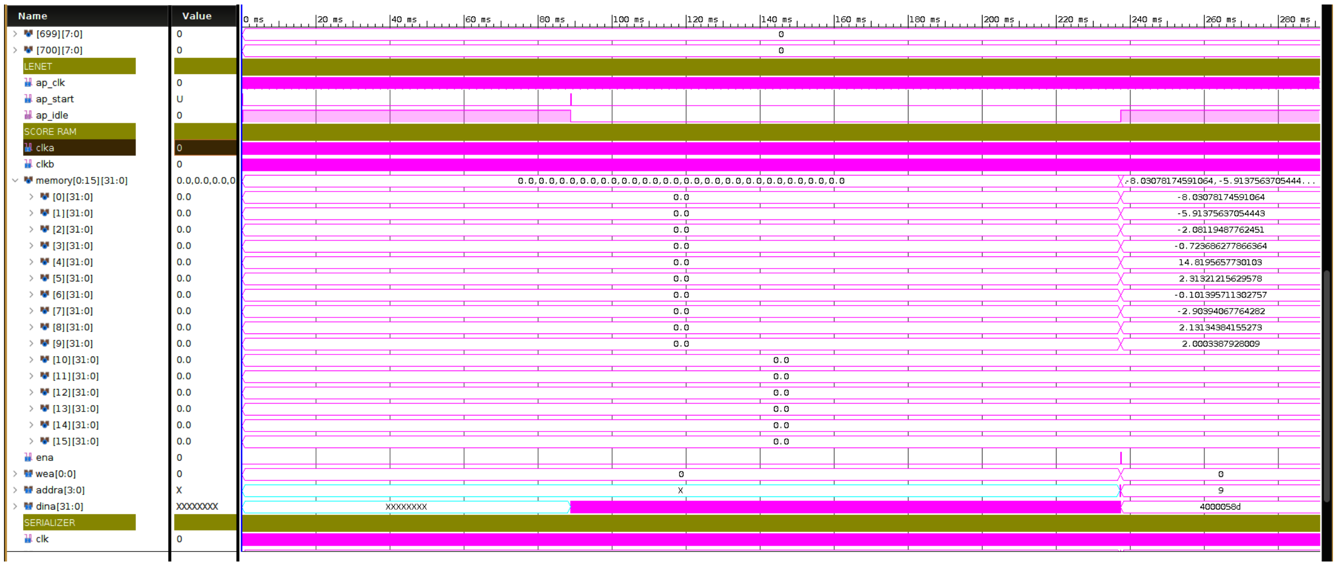


Figure 12: Full system simulation, part 3, LeNet5 activity and score RAM contents.

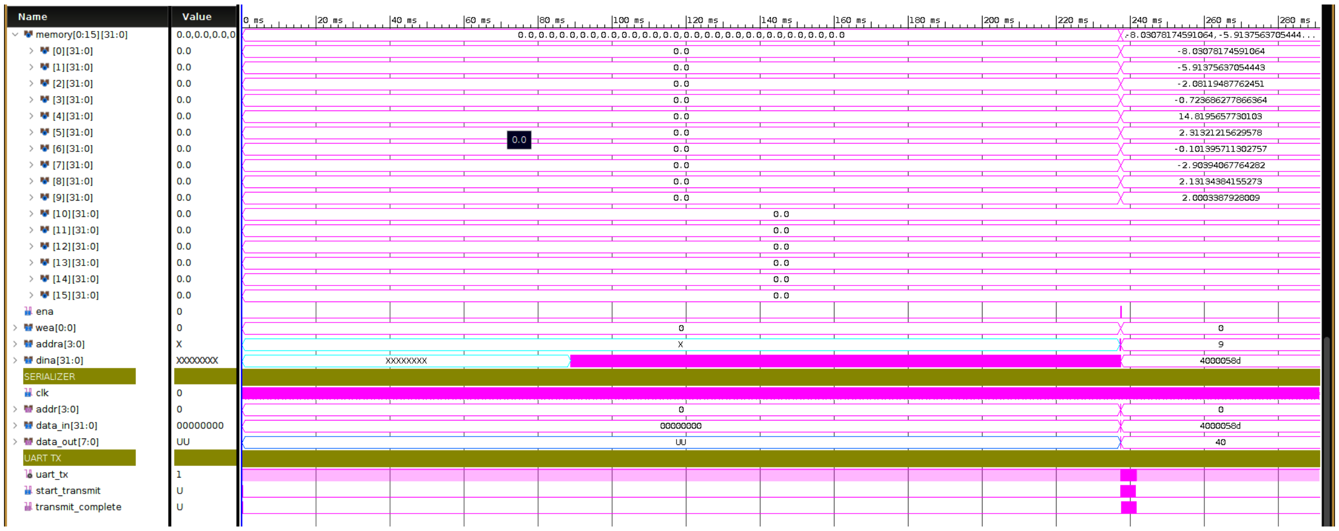
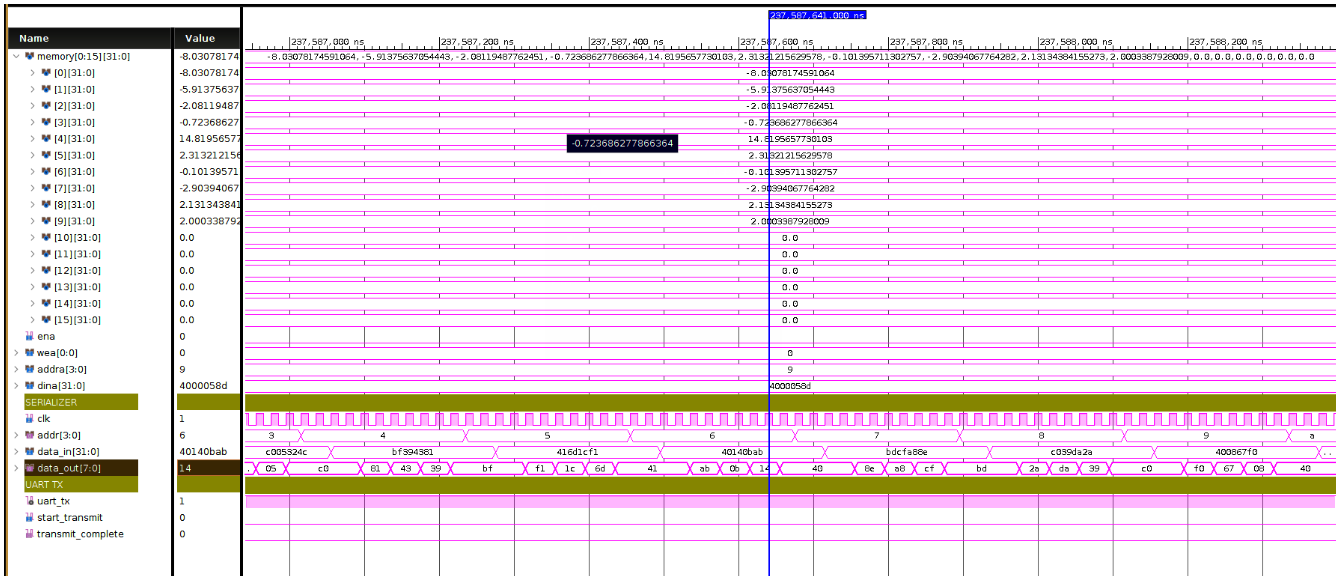


Figure 13: Full system simulation, part 4, serializer and transmit UART.

Figure 14 shows the detailed output of the serializer module.



Name	Value
memory[0:15][31:0]	-8.03078174
[0][31:0]	-8.03078174
[1][31:0]	-5.91375637
[2][31:0]	-2.08119487
[3][31:0]	-0.72368627
[4][31:0]	14.81956577
[5][31:0]	2.313212156
[6][31:0]	-0.10139571
[7][31:0]	-2.90394067
[8][31:0]	2.131343841
[9][31:0]	2.000338792
[10][31:0]	0.0
[11][31:0]	0.0
[12][31:0]	0.0
[13][31:0]	0.0
[14][31:0]	0.0
[15][31:0]	0.0
ena	0
wea[0:0]	0
addra[3:0]	9
dina[31:0]	4000058d
SERIALIZER	
clk	1
addr[3:0]	6
data_in[31:0]	40140bab
data_out[7:0]	14
UART TX	
uart_tx	1
start_transmit	0
transmit_complete	0

Figure 14: Detailed view of serializer input and output.

Discussion

Overall, the project was moderately successful. When tested independently, the network was able to successfully classify input samples correctly in simulation. This network was then integrated into a larger datapath system, controlled by a state machine, that enabled it to interface with a PC over UART. This full system was also successfully validated in simulation, proving that the system logic was correct. This is shown in Figures 10, 11, 12, and 13. (The CLK_100MHZ signal name is a typographical error. All clock signals in the system run at 50MHz.) Figures 10 and 11 focus on the input stage of the system and show the receive UART and image RAM portions of the system. Figure 11 shows a subset of key values in the image RAM, indicating that it is being properly populated with pixel values from the input image received over the UART.

Figures 12 and 13 focus on the LeNet5 network itself and the score RAM. Figure 12 shows that the network is being successfully triggered by the top level state machine and signaling that it is no longer idle. Once the network has finished processing, it can be seen that the score RAM has been populated with the network's raw output scores. In this case, the network was given a handwritten number four as its image input. Since the score RAM addresses correspond to the possible classes, the memory address with the highest score should be address 0x4. This matches what is present in the simulation, as the highest score of 14.82 was assigned to this slot.

Figures 13 and 14 show the output stage of the network, detailing the serializer and transmit UART. Figure 13 shows that the serializer successfully dequeued the floating point values from the score RAM and transmitted them a byte at a time over the UART, with the least significant byte being sent first. Figure 14 shows the detailed operation of the serializer. For each 32-bit floating point value, it details the decomposition of that value into its constituent bytes and enqueueing them into the transmit FIFO.

In order to get the network system to this point, several optimizations were applied at various points in the toolchain. These optimizations are detailed in the subheadings below:

Softmax Function Relocation

Early on in the development process, it was discovered that the softmax function, shown as Equation 1, would not be able to be easily implemented on the FPGA.

$$\sigma(z)_j = \frac{e^{z_j}}{\sum_{k=1}^{K} e^{z_k}} \tag{1}$$

The softmax function is responsible for converting the raw network scores into classification probabilities. It does this by first exponentiating all the scores. This causes negative scores to become very small positive numbers, while positive scores become very large positive numbers. These exponentiated scores are then summed together. Finally, each individual exponentiated class score is then divided by the sum-of-exponentials to obtain a probability for each class.

This process could not be implemented on the FPGA, as it requires access to both an exponential function and division operation. Both of these are very computationally expensive, especially for floating point operations. However, the exponential function is the most problematic, as there is no IP Core that is capable of providing an efficient implementation. (Vivado's CORDIC IP Core only supports trigonometric functions.) As a result, it was decided to move the softmax function into the PC side Matlab script. This presented no issue from a machine learning standpoint, as when the network is trained, no learning occurs in the softmax layer.

Network Weights as Globals

One significant issue in the Matlab synthesis function was that the network coefficients were passed into the synthesis function as parameters. This caused the synthesis toolchain to make these arrays into memory interfaces on the final VHDL entity. This interface style would have led to a large amount of extra work, as it would require the manual creation of ten block ROMs, each with their own coefficient files. Some of these coefficient files would contain up to 48,000 individual coefficients. They would then have to be manually converted from 4D arrays into 1D arrays. This process would be time intensive and error prone, as the row- or column-major order of each dimension would have to determined by trial and error, possibly without any internal feedback from examining the network's internal activation maps. Furthermore, during C code generation, these arrays were automatically flattened from 2D, 3D, or 4D structures into 1D structures, allowing for the easy automatic generation of block ROMs for the network coefficients.

Fully Connected Layers as Convolution

The final synthesis level optimization to be performed was the conversion of the first fully connected layer (FC1) from a vector multiply into a convolution. This was done to easily facilitate an otherwise expensive dimensionality conversion. This issue stems from the fact that the output of the final max pooling layer (POOL2) is a 3D activation map, while the expected input into the FC1 layer is a 1D array. However, both the vector multiply operation and the convolution operation are defined as multiply-accumulates. This means that convolving the 3D POOL2 activation map with a 3D convolution filter of the same size as the activation map, is functionally equivalent to performing a reshape operation, followed by a vector multiply. By using a convolution operation instead to perform the vector multiply at the boundary, it was possible to eliminate the reshape operation from the network. This means that only one 3D memory block is required at the interface between the POOL2 and FC1 layers. If the reshape operation was performed, then there would be two memory blocks, one 3D for the POOL2 activation map, and the other 1D for the FC1 input memory. This double memory structure would be expensive both in terms of hardware utilization and in terms of time required to physically perform the copy between memory banks. It also allowed the subsequent FC2 and FC3 layers to be implemented strictly using the FPGA's onboard DSP hardware, resulting in significant savings in FPGA fabric space.

Unfortunately, despite these optimizations, it was not possible to get the network running in hardware. This was due to several issues encountered when using the Matlab and Vivado synthesis tools, but was primarily caused by Vivado's implementation tool being unable to meet the timing requirements required for the circuit to operate properly. These issues are detailed in the subheadings below:

Failure of Matlab's HDL Coder Tool

The first attempt at implementing the LeNet5 architecture utilized only Matlab's HDL Coder toolchain. The HDL Coder was configured with several optimizations in mind. The clocks were configured with an asynchronous, active high reset, with clock enables. In order to simplify the entity interface, the system was configured to map the constant model parameters into block RAM. The model was also pipelined, with 10 stages, to account for the 10 stages of the LeNet5 model. The VHDL code was then generated.

Unfortunately, Matlab appears to have ignored several of the settings used to generate the VHDL code, specifically, the ones related to mapping constant model parameters into block RAM and the pipelining directives. Furthermore, Matlab does not appear to directly support the synthesis of state machines directly from the specified synthesis function. Lastly, it was not possible to configure the HDL Coder tool for the specific chip to be targeted. This meant that the tool was completely unaware of any hardware resources that may be available that would enhance the generated model's efficiency. As a result, the final VHDL model appeared to be an almost purely combinational implementation of the LeNet5 architecture. This led to the generation of a massive number of hardware devices that would be unimplementable on the provided FPGA. The generated hardware listing is shown as Table 3.

Multipliers	2,597,058
Adders/Subtractors	5,357,644
Registers	62,834
RAMs	0
Multiplexers	11,152
I/O Bits	501,924

Table 3: Summary of synthesized resource utilization.

This table indicates several things. First, is that the number of multipliers, adders, subtractors, and registers indicates that this is likely a combinational logic approach to implementing the network. Second, the number of block RAMs used by the design is zero. This means that the tool did not correctly identify the input coefficients as constants and store then in block RAM. Finally, the number of I/O bits, which corresponds to the number of required pins in the design, is over half a million. This is due to the tool's failure to generate valid memory interfaces for the arrays, meaning that the images and the coefficients would have to be applied “all at once” as one massive block of data to all the network's inputs. As a result of these shortcomings, a purely Matlab-based approach was abandoned and a new, hybrid Matlab/Vivado HLS approach was adopted instead. Under this new approach, Matlab would be used to generate C code, which would then be passed into Vivado HLS to generate the final VHDL model.

Vivado HLS memcpy Failure

One of the first issues encountered when moving to the new hybrid synthesis approach was that some of the C code generated by Matlab contained structures that were not synthesizable by Vivado HLS.

The structure that was most problematic was the C code's use of the memcpy function. This function was used to copy the network weights from a global data file into a different global data file. (This code generation pattern appears to be automatic and not under the user's control.) When this code was synthesized, the Vivado HLS tool was unable to identify the coefficients in the array. As a result, these arrays were empty in the final VHDL model, which meant the block ROMs were never allocated in hardware and the network would generate output scores that were always zero.

In order to remedy this situation, the memcpy function calls were removed. These arrays were then manually moved into the synthesis function as local variables. All files containing global references to these arrays were then deleted, ensuring that the arrays were purely local to the synthesis function. This allowed for the proper generation block ROMs containing the network coefficients.

Matlab Quantization Code Failure

In order to drive down resource consumption and eliminate the need for floating point hardware, an attempt was made to quantize the network into the fixed-point domain. Utilizing Matlab's Fixed-Point Designer tool, it was determined that the optimal level of quantization was 16-bit, which led to a 50% reduction in the memory footprint and the elimination of all floating point hardware from the FPGA fabric, with a less than 1% drop in classification accuracy. This quantized model was then used for C code generation.

Unfortunately, the C code that Matlab inserted to perform the quantization caused significant issues with the Vivado HLS tool's ability to generate VHDL. This manifested in two different ways. First, Vivado HLS generated an invalid top level VHDL entity for the network. A properly formed network entity has an input-only memory interface for the network's image input, an output-only memory interface for the network's score output, and a set of control lines for activating the network. In the quantized case, the generated network entity had two bidirectional memory interfaces, each with their own independent clock enables and write enables for the image input. The output score memory interface remained the same. Second, and more concerning, was that when this network was installed in a VHDL testbench, the internal state machine would fail to operate in a behavioral simulation. Rather than output a series of valid memory addresses at the input to read in data from the image RAM, the address lines remained locked in the high-Z mode, preventing any information from entering the network. Since the cause of this failure could not be determined, and at the time it appeared possible that a full floating-point implementation was possible from a resource consumption standpoint, quantization of the network was abandoned.

Simulation Failure

Once a correctly generated VHDL model was created, it was then simulated to verify that the logic was correct. This was done by adding the signals shown in Figures 10, 11, 12, and 13 to the simulation waveform. However, in this instance, the entire memory arrays for the image RAM and the score RAM were added to the plots. This resulted in nearly 9,000 signals being tracked by the simulator. Unfortunately, this proved to be too much for the simulator to process. This lead to the memory arrays remaining at zero, even though data was actually being written into the memories. The issue was discovered when it was observed that the memories were both reporting being filled with zeros, but the transmit UART was sending correct data.

To correct the issue, only a carefully selected subset of the memory waveforms were tracked. These addresses were selected by careful examination of a single image of a handwritten number four. The chosen sample was first converted from a PNG image into a PGM image. During the conversion, the 32x32 pixel grid was changed into a 1024x1 vector. This allowed the PGM line numbers to be correlated with the memory addresses of the image RAM. This criteria was then used to select twenty representative entries in the image RAM to validate that the memory was being populated correctly.

This issue also prevented the full validation of the network in simulation, as each of the 10,000 samples in the test set would have a different set of twenty representative entries in the image RAM. This, combined with the fact that each behavioral simulation requires approximately 25 minutes to run, prevented the simulation from being validated against every sample in the test set.

Failure of Block RAM Implementation Optimizer
Once the behavioral simulation was complete, the synthesis and implementation steps were run. Based on inspection of the synthesis report, this process ran correctly. However, the implementation process generated an unusual resource utilization report. Specifically, all of the system's block memories were removed from the final implementation. This meant that there was no storage allocated for the image RAM, score RAM, coefficient ROMs, or internal activation map RAMs. This was further reflected in the visualization of the implemented design, which lacked the characteristic vertical stripes, indicating the presence of large memory arrays.

Research indicated that this may be caused by an issue with one of the optimizers run during the implementation phase. The first solution was to apply the "dont_touch" directive to the network IP Core files containing the block memories. However, these files are marked as read-only, meaning that this directive was not able to be added directly to the IP Core files. It was then determined that it was possible to add this directive as part of the constraint file. However, this was also unsuccessful, as the directive requires a hierarchy path to the VHDL entity in question. It appears that since the files requiring the "dont_touch" directive were located inside an IP Core, it was unable to resolve the path to reach them. It was determined that applying this directive to the entire IP Core may result in all optimizations being disabled for the majority of the logic in the system. As a result, another solution was pursued.

Further research revealed a particular case extremely similar to what was observed in this instance. In this instance, this individual was able to enable prevent the implementation optimizer from removing the block memories through the use of a series of command line switches. These switches were applied under Settings → Implementation → Opt Design → More Options. The specific switches were "-retarget -propconst -bram_power_opt". These options were then researched in the Vivado Implementation User's Guide.

The -retarget flag is used when moving from one hardware platform to another. However, in this instance, the synthesized IP Core was created using the same chip configuration values as used on the Nexys4 DDR board. This flag may have helped resolve this block memory issue if there was a subtle configuration difference between the bare chip configuration used in Vivado HLS versus the board configuration files used by the system implementation.

The -propconst flag is used to automatically remove unnecessary logic from the system. It specifically concentrates on removing "transparent" logic, such as an AND gate with a constant logic 1. It is unlikely that this would have resolved the block memory issue; however, it was included, as it was part of the specified solution to the issue.

The -bram_power_opt flag modifies the block memories by improving the clock gating to the memories and modifying the write configurations the dual-port RAMs. It is not clear specifically from the documentation what is meant with respect to the clock gating, but it may be ensuring that no gated clocks are present on the block memories. The write configuration changes do not impact this design, as all dual-port RAMs in the network IP Core are used in both the read and write mode. (These RAMs compose the storage for the activation maps, meaning that they are being written to by the prior layer's output and read from as the subsequent layer's input.)

Further examination of the documentation reveals one more item of interest. All three of these flags are enabled by default. However, since the addition of these flags was the only change made to the project, it appears that there may be a bug in Vivado's implementation optimizer, specifically relating to enabling these optimizations. Further investigation is needed to determine which flag or combination of flags is responsible for keeping the block memories. Unfortunately, due to time constraints, it was not possible to perform this analysis.

Final Timing Constraint Failure

The final and most significant issue was determined from the final implementation reports. There were several warnings regarding a failure with respect to timing constraints, specifically regarding the UART module. In order to successfully route the design, it was necessary for the implementation optimizer to create several gated clocks for the UART. This was done automatically, even though the coding style of both the transmit and receive UART modules specified a clock enable instead. Gated clocks present an issue in FPGAs because the insertion of the gate into the clock tree takes the clock signal off the clock distribution tree and into the FPGA fabric. This means that guarantees regarding the management of clock skew cannot be maintained. As a result, the UART may not have been interfacing correctly with the rest of the system, which is consistent with the observed failure mode of the FPGA never returning any scores to the PC-side Matlab script.

This issue also occurred during the synthesis of the VHDL model from the generated C code. Due to how the VHDL model was generated, it was not possible for the tool to fully validate the timing of the logic internal to the IP Core. This timing issue may also have occurred inside the network itself. As a result, more detailed implementation level simulations may be needed to validate the network. Unfortunately, due to the size of the system in question and based off some limited testing, this simulation may have needed nearly 24 hours to successfully run to completion on a system with significant amounts of available memory.

Conclusion

Overall, the project was partially successful. The model was able to be validated in behavioral simulation, proving that the system is logically correct. However, the available tools are not able to synthesize functioning hardware. This appears to be due to the interaction between the synthesis and implementation tools and the available hardware. The root cause seems to be that the model cannot be instantiated on the available FPGA, due to timing constraints. The source of these constraint issues appear to be rooted in the limited resources available on-chip. Had it been possible to get the quantized C code to generate valid VHDL, the network would have been able to fit on the provided FPGA. Moving to a larger FPGA with a denser resource allocation may alleviate these issues and enable the timing constraints to be met. Unfortunately, this hardware is unavailable.