# Development of a Flexible, Instruction-less Hardware Architecture for Inference of Neural Network Models

A Senior Project Report

presented to

the Faculty of California Polytechnic State University

San Luis Obispo

In Partial Fulfillment

of the Requirements for the Degree

Bachelor of Science in Computer Engineering

By

Ashwin Rajesh

June 2025

**Abstract**

The sizes of artificial intelligence models have exploded in recent years with the introduction of large-language models (LLMs) such as ChatGPT, making environmental impact a more prominent concern. Improvements can be made to the hardware that trains and inferences these models by designing specific architectures that are optimized for the model design. Tensor Processing Units (TPUs), such as the Google TPU, utilize a systolic array to perform matrix multiplication which serves as the backbone computation of model inferencing. This paper presents a similar design with a couple key differences: the lack of an instruction set and the ability to size internal modules based on model size to limit overhead and waste. Specifically, the deliverable is a program that converts trained Keras models into a hardware design written in SystemVerilog, which can then be simulated or synthesized for an FPGA or ASIC. The results demonstrate the the hardware design accurately replicates the behavior of the original model.

**Table of Contents**

## Introduction

The development of artificial intelligence (AI) has vastly progressed over the past couple of decades. With the groundbreaking introduction of large-language models (LLMs), the most famous example being

OpenAI's ChatGPT, the use of machine learning-powered applications has become widespread. As a result, the field has steered towards "one size fits all" models that are diverse in capability but significantly larger in size and expensive to run. Concerns about power usage and environmental impact have thus come under more scrutiny [1].

While efforts can be made in software to reduce model size, another avenue to explore is the improvement of hardware architecture to be more efficient and less wasteful when inferencing models. Specifically, this project motivates the development of application-specific integrated circuits (ASICs) for the purpose of running inference on neural network models. ASICS are typically streamlined for the application they are designed to serve, potentially resulting in higher throughput and lower power usage in comparison to that of a general-purpose processor.

Designing hardware can be a daunting task, however. The purpose of this project therefore is to bridge the gap between the software development of neural networks and hardware design of architectures specific to those models. In particular, the goal is to write a conversion program that produces an HDL design optimized for models trained using popular libraries like Keras. The outcomes of this project aim to make hardware design for AI more approachable and inspire a movement towards specially designed architectures that minimize overhead, runtime and more importantly, environmental footprint.

## Background

*Machine Learning*

Machine learning can be described as providing a computer with a set of input and output data and instructing it to develop an algorithm that maps this data, on its own. Neural networks are a subset of this field and are heavily inspired by the function of neurons in the human brain; an input signal is passed in, on which some processing is performed, and the output is passed to other neurons [2]. In

artificial neural networks, the connections between neurons are weighted and each neuron stores an internal bias. Additionally, activation functions are applied to the outputs of each layer; these functions are intentionally nonlinear such that the model as a whole can learn nonlinear behavior.

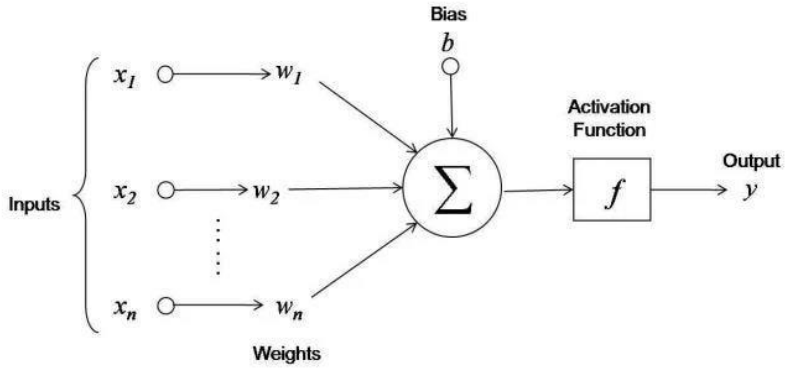


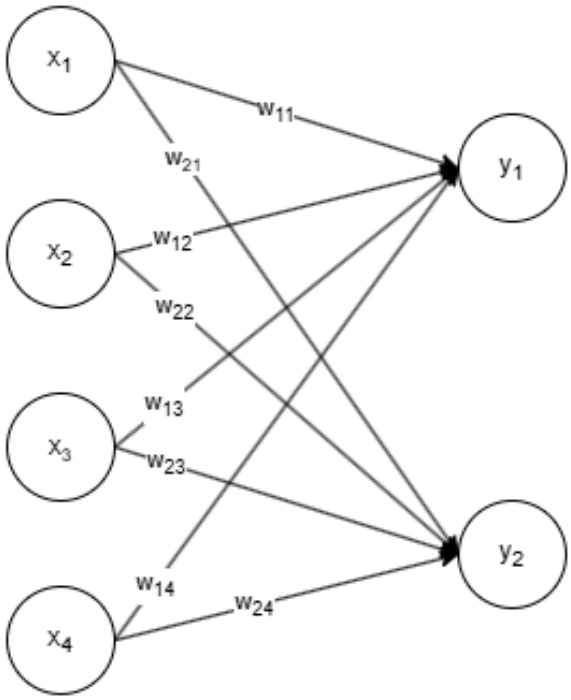

*Figure 1: Anatomy of a neuron in an artificial neural network [3].*

During the training process, sample data with known inputs and outputs are fed into the network, and the weights and biases are incrementally adjusted until the desired behavior is replicated. Once complete, the model can then perform "inference" on new input data to produce predicted output data. The scope of this project lies only in the inference stage but hopefully inspires readers to take similar approaches towards the training process as well.

Keras is a popular Python library that allows users to design neural networks on a layer-by-layer basis in addition to training and running inference on those networks.

*Matrix Multiplication in Neural Networks*

Matrix multiplication is used frequently in neural network inference. Applying a set of weights to the outputs of one layer produces the inputs of the new layer (before applying a bias and activation function) and can be performed with one such operation. This is shown in Figure 1 below.




$$\begin{bmatrix} y_1 \\ y_2 \end{bmatrix} = \begin{bmatrix} w_{11} & w_{12} & w_{13} & w_{14} \\ w_{21} & w_{22} & w_{23} & w_{24} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \end{bmatrix}$$

*Figure 2: Computing a neural network layer using matrix multiplication*

*Systolic Arrays and TPUs*

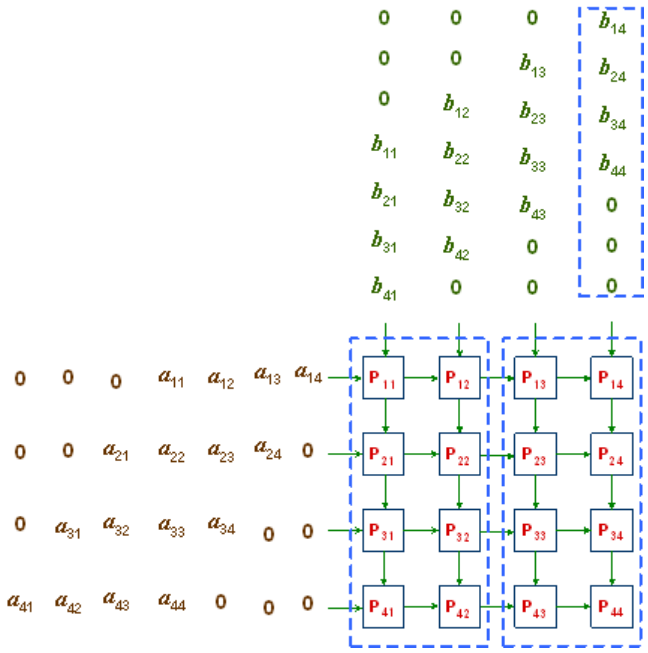



*Figure 3: Performing matrix multiplication using a systolic array [4]*

When the theory behind machine learning was first formulated, the general-purpose CPU was barely in existence. It was only decades later that the first GPUs were developed and introduced to machine learning due to their heavily parallelized structure. While a major advancement over the CPU, their versatility over a wide range of applications still meant that GPU design could not be fully optimized for running inference on AI models [5].

Kung et. al introduced the concept of systolic arrays in 1982, structures consisting of chained, light computation units (called processing elements, or PEs) that are best suited for performing tasks that are inherently parallel. This has utility in several applications, such as convolution and the discrete Fourier transform [6].

The systolic array can be designed to perform matrix multiplication by passing the values of one matrix through the array left to right and values of the other top to bottom. The PEs act as multiply-and-

accumulate cells and collectively store the value of the output matrix when the computation completes. The image in Figure 3 depicts two 4x4 matrices, a and b, multiplied to produce the matrix P.

For simple dense networks, where every neuron in one layer is connected to every neuron in the layers before and after, the dimensions of every layer's output will always be N x 1, where N is the number of neurons in the layer. An example of this is shown in Figure 2. For these layers, computation only requires a systolic array of N x 1 cells. Note that this is not the case for convolutional layers, where the inputs and outputs can be matrices rather than vectors. This project provides support for the latter but remains to be officially tested.

The Google Tensor Processing Unit (TPU) features a systolic array for performing the matrix multiplication required for neural network inference. The inputs are fed into the TPU via PCIe and the weights are pulled from DRAM. Additional hardware is designed to perform bias addition, apply an activation function to all intermediate outputs, and buffer the results of one layer to be used as inputs to the next. The architecture also exposes a few instructions to control the flow of data through the systolic array [5].

The Google Edge TPU is a smaller and lighter variation of the architecture for the purpose of edge applications [7]. The Coral USB stick is one such example.

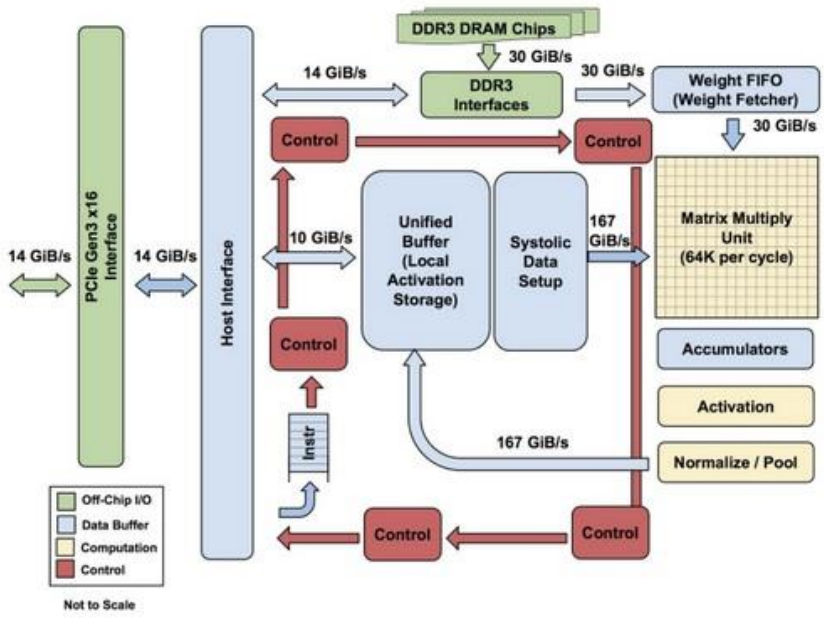



*Figure 4: Google TPU architecture [5]*

### Project Objectives

The primary requirement for this project is to develop a program in Python that reads a trained neural network model and produces a hardware design that not only replicates the behavior of the original model but is optimized for performing inference. Inspired by the Google TPU design, the underlying architecture shall be implemented with a systolic array at its core. However, the design will be instruction-less to reduce overhead as much as possible.

A secondary requirement is to match, if not improve upon, the timing and power benchmarks reported by existing TPU designs, such as the Edge TPU on the Coral USB. If time does not permit, this objective may be omitted and instead replaced with an analysis on the utilization of the systolic array and the efficiency of the final design.

Interfacing with the completed project is straightforward: a user designs and trains a model for a specific application and saves it to a standardized format (e.g. HDF5). The conversion program parses the saved model and produces SystemVerilog and memory files that describe the layout of the corresponding hardware architecture and the weights/biases that must be stored upon initialization. The user can then perform simulation to ensure accuracy before moving forward with synthesis for deployment on an FPGA or design of an ASIC.

## Design

Development of the foundational hardware design was a crucial aspect of the project. To streamline this process, modularization and unit testing were prioritized before implementation of the entire design. The modules are described in more detail in this section.

### *Systolic Array*

The systolic array is the central component of the design and performs matrix multiplication applicable to each layer of the neural network. One key design choice that lies here is the idea that weights and inputs are “pumped” through the array, while outputs are accumulated within the cells. The nature of the systolic array allows for other variations; one such example pumps the inputs and outputs and holds weights in the cells. This method, however, must deal with the issue of replacing weights after each layer, which could be time-costly for large arrays. Instead, by leveraging the idea that certain cells complete their computation cycle before others, this array ejects intermediate outputs in batches of N, where N is the width of the input vector (typically equal to 1 for dense networks). This allows for a more pipelined design overall.

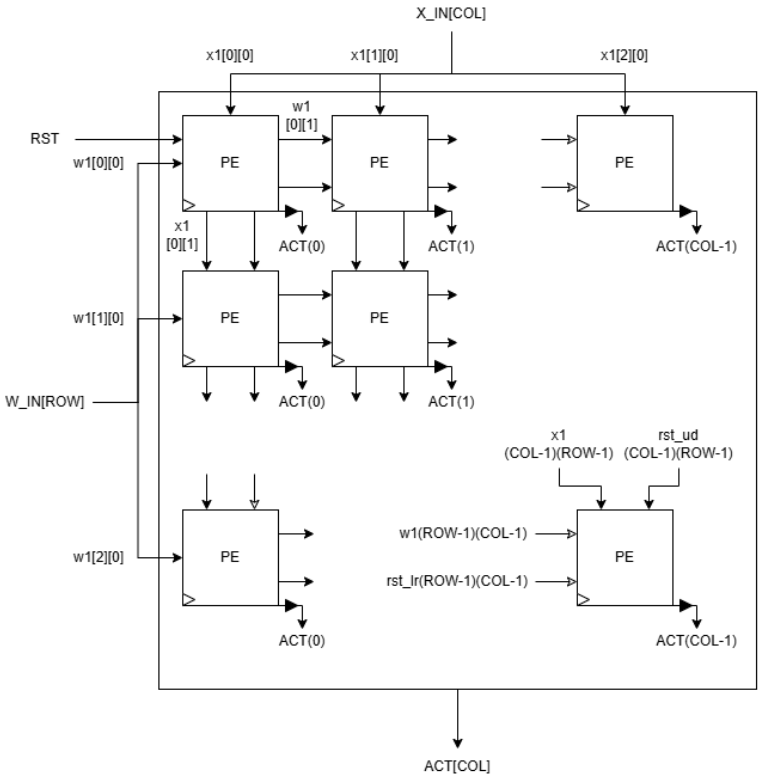



Figure 5: Internal design of the systolic array

*Processing Element (PE)*

In matrix multiplication, the PE is represented as a synchronous multiply-and-accumulate (MAC) cell with additional signals for systolic implementation and control flow. On each clock cycle, *w_in* and *x_in* are multiplied and added to a running total *y*, after which *w_in* is passed onto *w_out* and *x_in* to *x_out* (for other PEs). Reset signals come in and are pumped to other cells in a similar fashion; on reset, the PE outputs its total to *y_out* and sets *y* back to 0 in preparation for the next layer.

*Feed*

For the systolic array to perform matrix multiplication correctly, inputs on both sides of the array must be staggered such that values reach every cell at a specific time, as shown in Figure 3. More precisely,

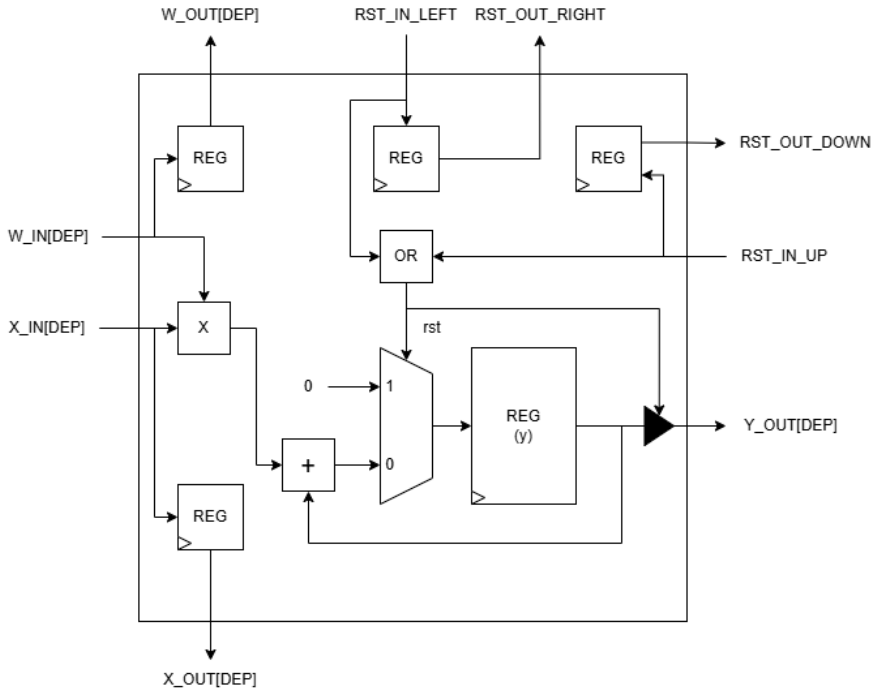



*Figure 5: Internal design of the systolic array*

values of the *k*th column/row must be delayed by *k – 1* clock cycles. Figure 6 shows how the feed module achieves this by introducing delays using the appropriate number of synchronous registers.

*Biasing and Activation*

Once a cell's calculation is complete, it sends the value out for biasing; the biases are stored in memory and timed with control signals to ensure that the correct bias is added to each cell output. This new output is then sent to the activation module, which applies the activation function of choice; this project uses the Rectified Linear Unit (ReLU) due its simple hardware implementation. Both the bias and activation modules employ combinational logic.

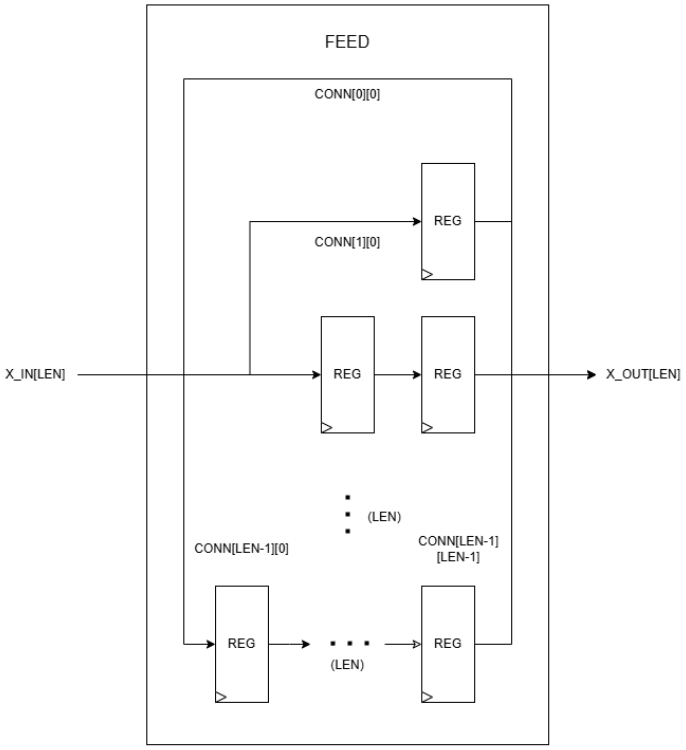



*Figure 6: Internal design of a feed module*

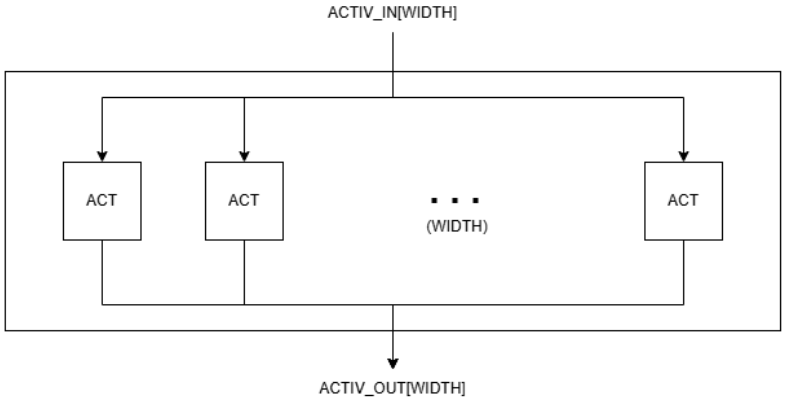



*Figure 7: Internal design of activation module*

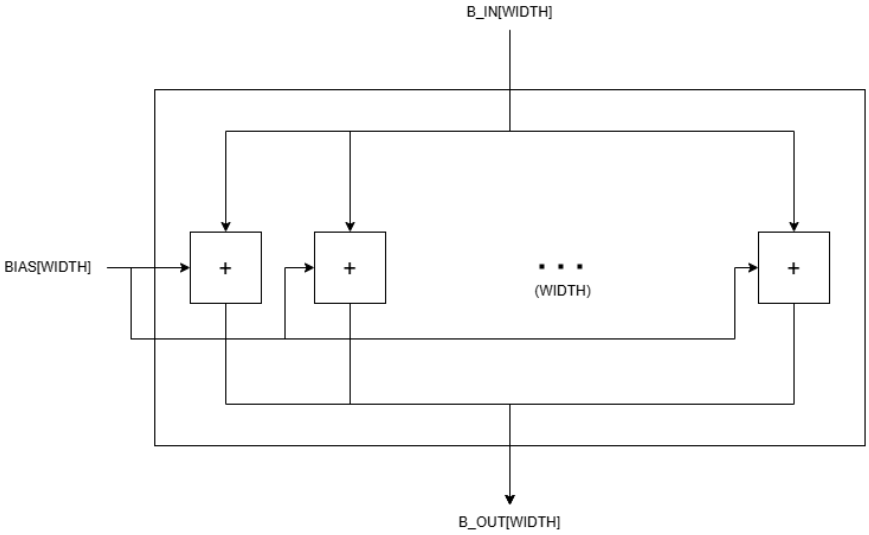



*Figure 8: Internal design of bias module*

*Memory*

Weights for the neural network model are stored in a register-based, read-only memory and passed into a feed module before entering the systolic array. This implementation for this project in particular supports reading multiple values from memory at a time. A similar design choice is made for the biases as well.

*Control*

The control module deploys signals to enable/hold the weight and bias memories when needed and determine when intermediate outputs should be fed back into the array as well as when the final outputs are ready to be sent out of the top module. To accomplish this, the module needs to know the layout of the neural network model, namely the total number of layers and dimensions of each layer. Counters are initialized and tracked to determine when a layer or the entire program completes.

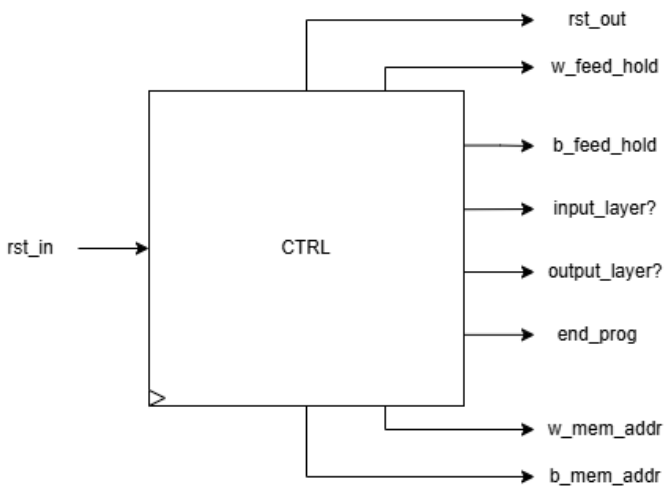



*Figure 9: Top-level block diagram of control module*

*Top*

All of the modules listed above are instantiated and connected within the Top module. Inputs are fed in and outputs are sent out as byte arrays. The RDY_OUT signal goes high when valid data is sent through OUT. Multiplexers are placed at the beginning and end of the module to guide the inputs to the first layer, route the outputs of one layer to the inputs of the next, and yield the outputs of the final layer.

The conversion program, written in Python, sizes modules based on the architecture of the provided neural network. This is a straightforward process due to the modularized and parameterized nature of the hardware architecture; the program need only choose the dimensions of the systolic array, memory, feed, bias and activation modules, in addition to storing model metadata in the control module. Models are accepted in the standard HDF5 format (which can be created in Keras). The layout of the neural network is inspected and the weights and biases are extracted; the final result is a collection of hardware design and memory files ready for simulation and/or synthesis.

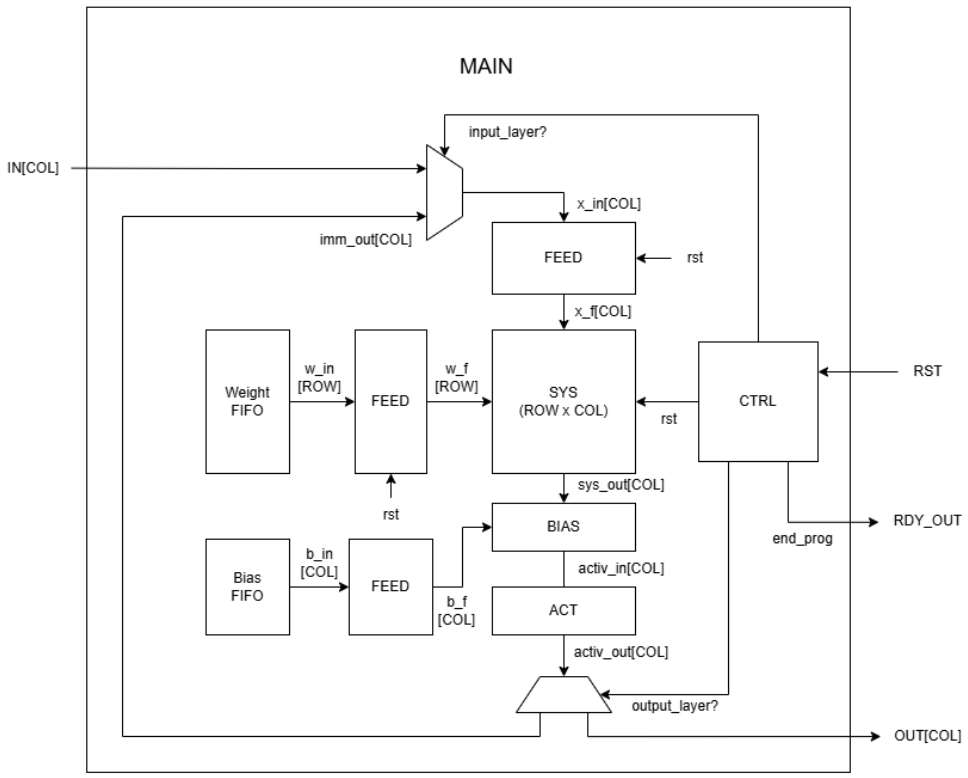


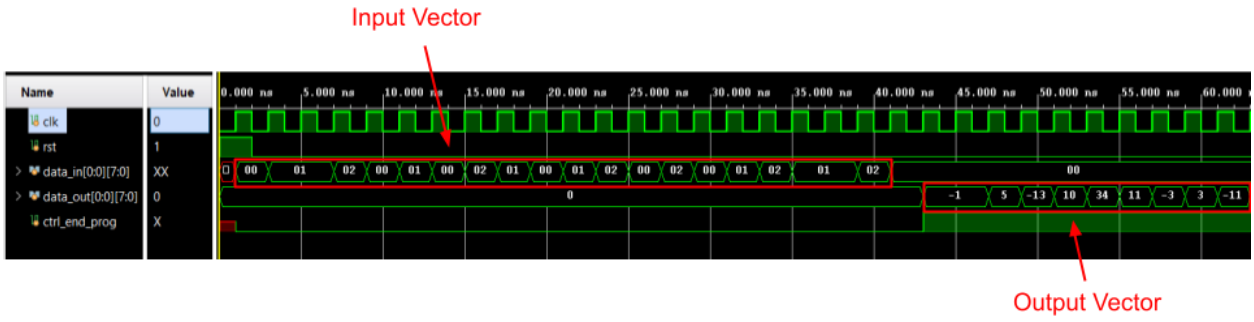

*Figure 10: Internal design of top module*

## System Design and Analysis

To ensure that the hardware design was properly replicating the behavior of the original model, sample neural networks with manipulated weights/biases were developed in Keras and sent through the conversion software to develop the hardware design in SystemVerilog. Simulation was performed in Xilinx Vivado, using the same inputs and weights/biases as in the trained model and a 500MHz clock (2ns period).

The first model (Test Case 1) takes an input vector of size 20 and consists of one layer with only 10 neurons. While not practical for real-world applications, this test primarily demonstrates and verifies the

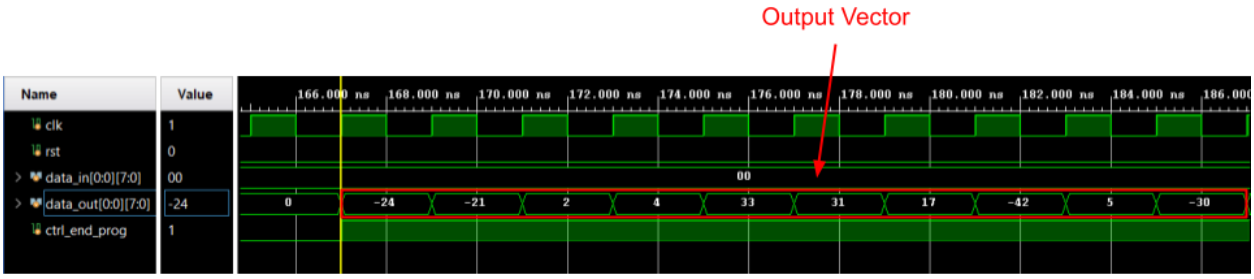
functionality of the systolic array. The waveform below shows the input and output vectors, the latter of which matches those produced by Keras for the original model.




```
tf.Tensor([[ -1.  -1.   5. -13.  10.  34.  11.  -3.   3. -11.]], shape=(1, 10), dtype=float32)
```

*Figure 11a: Waveform depicting input and output vectors for Test Case 1; Figure 11b: True output of Test Case 1 computed by Keras*

The second model (Test Case 2) is larger with an input vector size of 30 and consists of three layers with 30, 20, and 10 neurons respectively. This test targets the dataflow of the hardware architecture for a multi-layer model, ensuring that intermediate outputs are fed back into the systolic array in time for computation of the proceeding layer.




```
tf.Tensor([[-24. -21.   2.   4.  33.  31.  17. -42.   5. -30.]], shape=(1, 10), dtype=float32)
```

*Figure 12a: Waveform depicting the output vector for Test Case 2; Figure 12b: True output of Test Case 2 computed by Keras*

The outputs of both simulations appear to match that of the original model, conveying that the hardware design is working as expected. The inference time for each test, in clock cycles and actual time (with the assumption of a 500Mhz clock), are depicted in Table 1 below.

*Table 1: Test Case Inference Results*

| | # Parameters (Orig.) | # Parameters (Hardware) | Inference Time (clock cycles) | Inference Time (ns) |
|---|---|---|---|---|
| Test Case 1 | 210 | 210 | 21 | 42 |
| Test Case 2 | 1760 | 2460 | 83 | 166 |

Note that in case 2, the actual number of parameters in the hardware design is greater than that of the original model. This is due to zero padding stored in the weight memory that is passed into unused cells to avoid garbage output. For instance, as the systolic array in Test Case 2 was of size 30x1, not all cells were used in the computation of the second (20 neurons) and third layers (10 neurons). This disparity in parameter size appears to scale poorly with model size; a better approach includes additional control logic to mask certain cells when they are not in use for a specific layer.

In Test Case 1, a cell in the systolic array performed a relevant computation ~47.6% of the time, at maximum (10 of 21 clock cycles). In Test Case 2, this value jumped up to ~72.2% (60 of 83), corresponding to the first ten cells that were utilized in all three layers. Smaller models, especially those with less layers, suffer from a lack of utilization during the beginning of execution, when cells at the end of the array are not yet used; this effect diminishes as the total number of relevant computations increase with larger models.

## Conclusion and Future Work

This project demonstrated the successful implementation of a conversion program that produces hardware designs that can run accurate inference on neural networks developed in Keras.

There are several paths that can be taken to further advance the project. While the tests performed in the previous section produce encouraging results, they do not showcase the full capability of the design as they only implement simple, dense models with no practical application. A possible future test can be carried out using the MNIST dataset to train an image-based digit classification model, which would allow for implementation and verification of convolutional layers. This step would require all model weights and activations to be converted from 32- or 64-bit floats to 8-bit integers via quantization, which is available through an existing library such as TensorFlow Lite [8].

Further benchmarking should be undertaken to compare this design with existing GPU and TPU architectures and assess its practicality in terms of inference time and power usage. As mentioned in the Background section, the Coral USB stick can perform inference on trained neural network models and has a modified Google (Edge) TPU architecture.

## References


*[1] McDonald, Joseph et al. (2022). Great Power, Great Responsibility: Recommendations for Reducing Energy for Training Language Models. 10.48550/arXiv.2205.09646.*

*[2] R. E. Uhrig, "Introduction to artificial neural networks," Proceedings of IECON '95 - 21st Annual Conference on IEEE Industrial Electronics, Orlando, FL, USA, 1995, pp. 33-37 vol.1, doi: 10.1109/IECON.1995.483329.*

*[3] Ren, Shaofei et al. (2018). A Deep Learning-Based Computational Algorithm for Identifying Damage Load Condition: An Artificial Intelligence Inverse Problem Solution for Failure Analysis. Computer Modeling in Engineering & Sciences. 117. 287-307. 10.31614/cmes.2018.04697.*

*[4] Ziavras, S. Experiment 3: Systolic-array implementation of matrix-by-matrix multiplication. ECE 459 - Systolic Array Implementation of Matrix-By-Matrix Multiplication. https://ecelabs.njit.edu/ece459/lab3.php*

*[5] Norman P. Jouppi et al. 2017. In-Datacenter Performance Analysis of a Tensor Processing Unit. SIGARCH Comput. Archit. News 45, 2 (May 2017), 1–12. https://doi.org/10.1145/3140659.3080246*

*[6] Kung, "Why systolic architectures?," in Computer, vol. 15, no. 1, pp. 37-46, Jan. 1982, doi: 10.1109/MC.1982.1653825.*

*[7] L. Kljucaric, A. Johnson and A. D. George, "Architectural Analysis of Deep Learning on Edge Accelerators," 2020 IEEE High Performance Extreme Computing Conference (HPEC), Waltham, MA, USA, 2020, pp. 1-7, doi: 10.1109/HPEC43674.2020.9286209.*

*[8] David, Robert et al. (2020). TensorFlow Lite Micro: Embedded Machine Learning on TinyML Systems. 10.48550/arXiv.2010.08678.*

## Appendix

Link to GitHub repository: https://github.com/ashwinr000/CPE_Senior_Project