Spring 5-1-2023

A Reconfigurable Architecture for Matrix Multiplication for Low Power Applications

Jeffrey Love
University of New Mexico

Jeffrey Love
Candidate

Electrical and Computer Engineering
Department

This thesis is approved, and it is acceptable in quality and form for publication:

Approved by the Thesis Committee:

Dr. Marios Pattichis, Chairperson

Dr. Xiang Sun

Dr. Alonzo Vera

A Reconfigurable Architecture for Matrix Multiplication

for Low Power Applications

by

Jeffrey Love

B.S., Computer Engineering, University of New Mexico, 2015

THESIS

Submitted in Partial Fulfillment of the
Requirements for the Degree of

Master of Science

Computer Engineering

The University of New Mexico
Albuquerque, New Mexico

May 2023

A Reconfigurable Architecture for Matrix Multiplication for Low Power Applications

by

Jeffrey Love

B.S., Computer Engineering, University of New Mexico, 2015

M.S., Computer Engineering, University of New Mexico, 2023

ABSTRACT

This thesis presents a hardware architecture for performing matrix multiplication via a systolic array to reduce time complexity and power consumption. The proposed architecture, the Neural Network Accelerator (NNA), was designed in Verilog HDL to perform 8-bit multiplication to reduce the resources required to implement the NNA on low-power FPGAs. The NNA's open architecture is designed to support radiation testing for fault tolerant designs targeting space applications.

The NNA was compared against two matrix multiplication architectures: SARS and DAE. Max operating frequency for SARS and NNA was 210.2 MHz and 225 MHz respectively. DAE and NNA required 3,681 and 449 clock cycles for matrix multiplication respectively. The reconfigurable NNA architecture includes unique features: an 8-bit Instruction Set Architecture (ISA) to control ALU operations and data-flow, a neural network activation function (ReLU) module, 16-bit to 8-bit scaling of ALU results, and a max systolic array size of 255 x 255.

Table of Contents

List of Figures

1. Chapter 1: Introduction

Matrix multiplication is an essential component in machine learning and image processing for accumulating the weighted sums of neuron inputs and for performing convolution to smooth, sharpen and blur images. Matrix multiplication of matrices A and B, with dimensions M x N and N x P respectively, will have a time complexity of $\Theta(MNP)$, when performed using the naïve matrix multiplication algorithm executed on Central Processing Units (CPUs) without parallelization. Graphics Processing Units (GPUs), Tensor Processing Units (TPUs), Application Specific Integrated Circuits (ASICs) and hardware accelerators such as Field Programmable Gate Arrays (FPGAs), are examples of domain specific hardware developed to optimize the execution of specific applications using parallelism to increase performance. A simplified hardware architecture designed only for matrix multiplication, with compute units arranged in a systolic array, can perform matrix multiplication with a time complexity of $\Theta(n)$.

1.1. Motivation

In order to perform machine learning applications in space, efficient and radiation tolerant hardware is required. To determine the radiation tolerance of commercial hardware, Total Ionizing Dose (TID) and Single Event Effects (SEE) testing can be performed on the device. Without access to the device's architecture information, determining the resilience of the hardware is difficult. Commercial hardware architecture information is not public knowledge, which led us to build our own matrix multiplication architecture so that we could study its feasibility for space applications. Developing the architecture in HDL using non-vendor specific resources will allow the architecture to be deployed on the latest high-performance and energy efficient space grade FPGAs.

After reviewing current literature, we found two matrix multiplication architectures (SARS [2] and DAE [7]) developed for FPGAs and one architecture (EES [8]) developed for an ASIC flow. The SARS architecture uses a systolic array design. The DAE and EES architectures use a non-systolic array design. The SARS, DAE and EES architectures were chosen because they provided metrics that could be used for comparison; max frequency when implemented on an FPGA or clock cycles required for operation. All architectures are presented in Chapter 2 and used for comparison in Chapter 4.

1.2. Thesis Statement

The goal of this thesis is to develop a low-power, domain-specific architecture for performing 8-bit matrix multiplication, called the Neural Network Accelerator (NNA). The architecture will be developed to accelerate matrix multiplication for space applications, using the fundamental concepts of domain-specific architecture and systolic arrays.

1.3. Contribution

The architecture will be written in Verilog HDL and implemented on a low-power FPGA. The architecture will use an 8-bit Instruction Set Architecture (ISA) to control ALU operations and data-flow through each node of the systolic array, a feature not found in any other architecture that was found in the literature. Prior research was focused on decomposing matrix multiplication into smaller matrices, resulting in an exponentially increasing number of cycles. The NNA architecture's systolic array is configurable up to 255 x 255 to avoid unnecessary matrix decompositions and thus perform matrix multiplication in less clock cycles than alternative approaches. In addition, each compute unit in the systolic array will include a module for implementing a rectified linear activation function (ReLU)

commonly used in neural networks. Every ALU can be configured to scale the 16-bit result to 8-bits.

1.4. Overview

The remainder of the thesis will contain four chapters. In Chapter 2, we provide background information on current literature review of matrix multiplication algorithms, evidence for implementing matrix multiplication using systolic arrays and utilizing domain-specific architecture concepts to reduce time complexity and power consumption. Systolic and Non-systolic array, matrix multiplication architectures are presented in Chapter 2. We provide an analysis of the accuracy of 8-bit neural networks in Chapter 2. In Chapter 3, we provide details and block diagrams for the proposed Neural Network Accelerator (NNA) architecture. In Chapter 4, testbench results and performance metrics for the architecture will provide evidence that the NNA architecture performs matrix multiplication in less clock cycles than matrix multiplication architectures found in recent literature. Chapter 5 will provide concluding remarks and a plan for future work.

Chapter 2: Background

This chapter will provide references to current research on reducing the time complexity of matrix multiplications algorithms, systolic array architecture for matrix multiplication, the advantages of domain specific architecture and non-systolic array matrix multiplication architectures.

2.1. Matrix Multiplication Algorithms

Research conducted to decrease time complexity to the theoretical lower bound has led to new algorithms; three examples are the Strassen algorithm [8][11], Coppersmith-Winograd [8][11] algorithm and the Laser method [10][11]. Naive matrix multiplication of an n x n matrix requires $m^2(m-1)$ additions, and m^3 multiplications. Both algorithms reduce the number of multiplications in favor of more additions.

The Strassen algorithm uses the divide-and-conquer technique to perform matrix multiplication in parallel. The algorithm has two requirements:

1. Both matrices must have dimensions n x n.
2. n must be a power of two.

The matrices can be padded with zeros to meet the requirements. Any matrix that is greater than 2 x 2 will be decomposed into m, 2 x 2 matrices where,

$$m = 2^{2[log_2 n-1]}.$$

Each 2 x 2 matrix multiplication requires seven multiplications and eighteen additions as shown in the Equation 1 below for the multiplication of matrices A and B.

$$\begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix} \begin{bmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{bmatrix} = \begin{bmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{bmatrix}. \qquad (1)$$

$P_1 = (A_{11} + A_{22})(B_{11} + B_{22})$.

$P_2 = (A_{21} + A_{22})B_{11}$.

$P_3 = A_{11}(B_{11} - B_{22})$.

$P_4 = A_{22}(B_{21} - B_{11})$.

$P_5 = (A_{11} + A_{12})B_{22}$.

$P_6 = (A_{21} - A_{11})(B_{11} + B_{22})$.

$P_7 = (A_{12} - A_{22})(B_{21} + B_{22})$.

$C_{11} = P_1 + P_4 – P_5 + P_7$.

$C_{12} = P_3 + P_5$.

$C_{21} = P_2 + P_4$.

$C_{22} = P_1 - P_2 + P_3 + P_6$.

Strassen's algorithm reduces the time complexity of matrix multiplication to $\Theta(n^{2.81})$ [11].

The Coppersmith-Winograd algorithm improves upon the Strassen algorithm by reducing the number of additions to fifteen, which reduces the time complexity to $\Theta(n^{2.376})$ [11]. Until 2010, this was considered the fastest matrix multiplication algorithm. As of 2022, researchers are still attempting to reduce the time complexity to $\Theta(n^2)$, the Laser Method was used to create a new algorithm from the Coppersmith-Winograd algorithm, and is currently the fastest algorithm developed, with a time complexity of $\Theta(n^{2.372})$ [11].

The Laser method is an indirect method used to create new matrix multiplication algorithms from existing algorithms. Strassen developed the Laser method, and it was optimized by Coppersmith and Winograd to achieve better results [11]. The matrices are converted to a trilinear polynomial tensor to perform matrix multiplication. The Laser method reduces the direct sum of matrix multiplication tensors (T) to powers of tensors. In 2022, the Laser

method was used to produce a new algorithm with a $n^{0.004}$ improvement over the original Coppersmith-Winograd algorithm.

2.2. Systolic Arrays

The concept of the systolic array was first published in 1979 by H.T. Kung and Charles Leiserson [5]. Systolic arrays are inspired by biology where dataflow is similar to blood flow; the heart pumps blood to many cells which process it before returning to the heart [1]. In systolic arrays data flows from memory to ALUs or processing units, allowing each component to access the data before it returns to memory. The components are arranged in a two-dimensional array, where each component shares connections to other components allowing data to flow between them. Components in the middle of the array only have connections to their nearest neighbors, and only the boundary components have access to memory. Also, data only flows one-way, read once from memory and written to memory at the end. The components take two operands as inputs and produce one result, the operands are shifted into the ALU when needed so that local (distributed) memory is not required. Figure 1 shows two matrices being shifted into the systolic array to perform matrix multiplication. After matrix multiplication is complete, the result for each component is

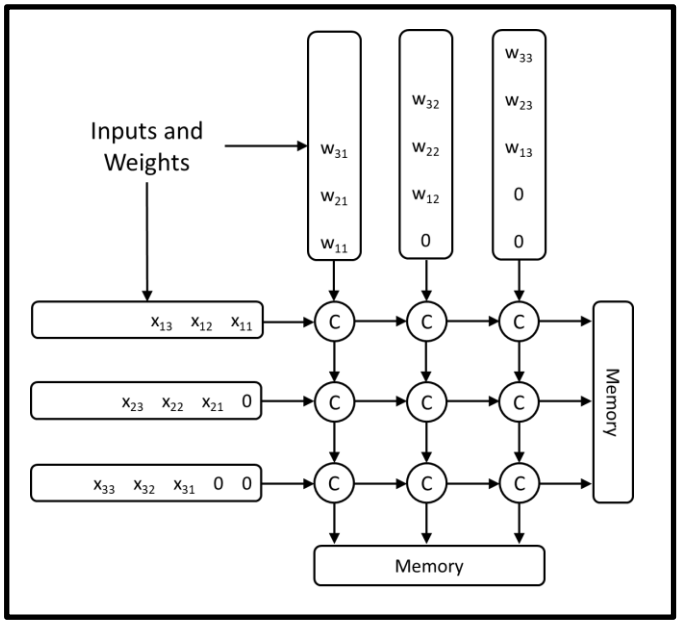

Figure 1: Systolic Array of Components.

shifted out of the matrix to the right or bottom until it is stored in memory. If an n x m matrix is too large for the systolic array, the matrix can be decomposed into smaller matrices.

2.3. Domain Specific Architecture

General purpose computing hardware offer generic resources suitable for a wide range of applications, from simple arithmetic to simulating ANNs. We first summarize implementations on CPUs and GPUs. CPUs are designed for low-latency and low throughput processing and GPUs are designed for high latency and high throughput processing. CPUs and GPUS can accomplish most tasks, but neither excels at any one application, for application specific performance improvements we can design domain specific hardware.

According to Amdahl's law, significant performance improvements are gained by using a faster mode of execution, if the faster mode of execution can be used for most of the computation time. Domain specific hardware are designed to provide performance improvements to specific applications and is optimal when the size of the memory(s), parallelism and data types match the target application [4]. To decrease power consumption and area, Arithmetic Logic Units (ALUs) or processing units are designed with only the specific hardware required for the application.

AMD/Xilinx has developed their Versal family of FPGAs to provide specialized hardware engines for applications: cloud computing, network processing, domain-specific parallelism, convolution, machine learning and video/image processing [13]. The Versal AI engine is a 2D array of vector-vector and matrix-matrix compute engines with tightly coupled memories.

With AMD/Xilinx developing architecture to accelerate application performance we can conclude there is a market for highly efficient domain-specific hardware.

2.4. Systolic Array Matrix Multiplication Architectures

In 2011, researchers Mahendra Vucha and Arvind Rajawat from the Maulana Azad National Institute of Technology (MANIT) presented their Systolic Architecture on Reconfigurable Systems (SARS) design for FPGAs [2]. Their architecture contains Processing Elements (PE) connected in a systolic array. Each PE contains a 4-bit multiplier and accumulator and the design shown in Figure 2 was implemented on a Spartan-3E FPGA.

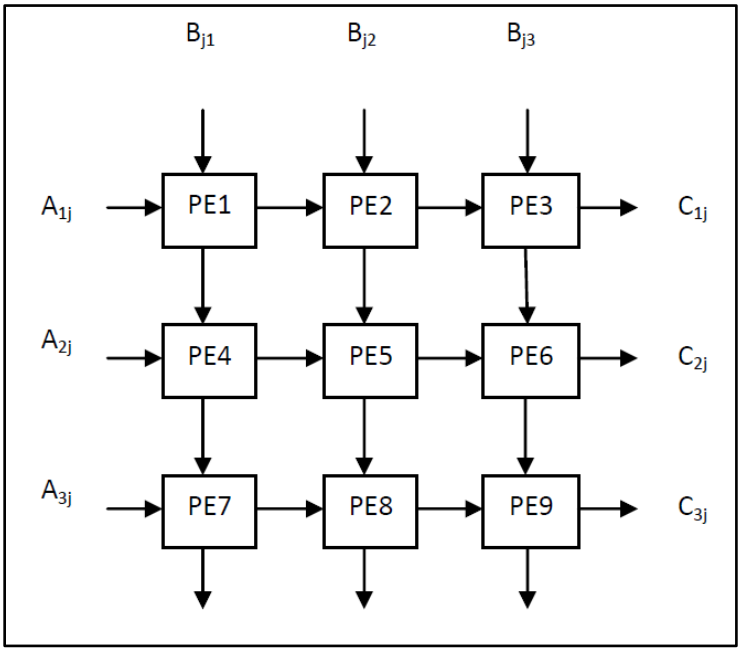

Figure 2: SARS Design [2].

The SARS design features a clock and reset and does not contain memory for the inputs and outputs. This design will be used in comparison with the NNA architecture.

2.5. Non-Systolic Array Matrix Multiplication Architectures

In 2018, Zhe Chen [7], a researcher from Uppsala Universitet, presented a FPGA hardware accelerator for matrix multiplication based on a Decoupled Access-Execute (DAE) architecture [7]. The architecture separates data processing into two steps, memory access and data execution, which are performed concurrently. The architecture contains four subsystems: Data Select, Buffer, Calculation, and Processing/Output, shown in Figure 3.

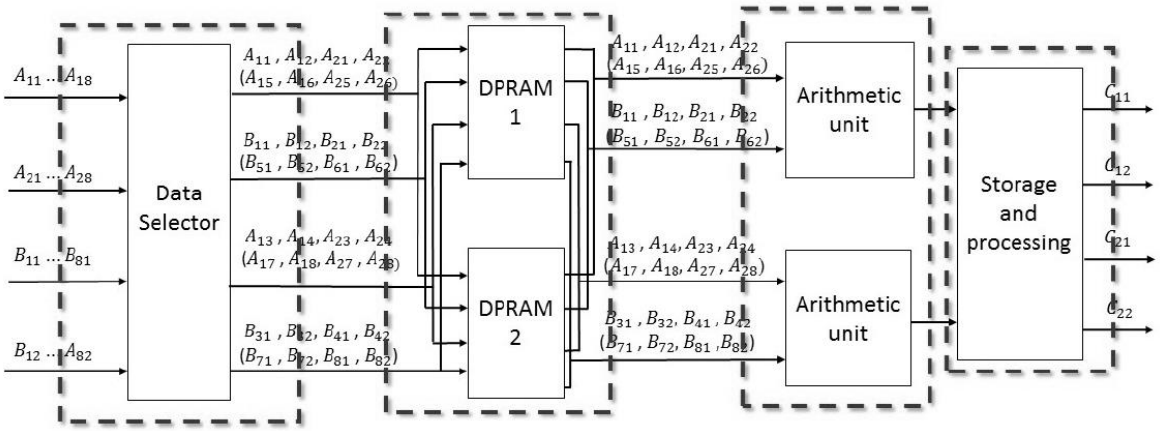

Figure 3: Multiplication Sub-Modules. [7]

In the Data Select module, the input is the first two rows of matrix A and the first two columns of matrix B, each divided into four 2 x 2 matrices which become the inputs of the Buffer module. The Buffer module uses two BRAMs to implement the DAE design, the output of the two BRAMs are added together and passed to the Calculation module. The Calculation module performs matrix multiplication on the four 2 x 2 matrices received from the Buffer module. Each 2 x 2 matrix is passed to the Processing/Output module which combines the 2 x 2 matrices and produces the output matrix C.

In 2020, Shanmugakumar, M., Srinivasavarma, V. S. M., and Mahammad, N. from Madanapalle Institute of Technology and Science and IIITDM Kancheepuram, presented an energy efficient and scalable (EES) ASIC hardware architecture for matrix multiplication [8].

The architecture performs matrix multiplication by taking all elements of the first matrix in parallel and the second matrix is separated into columns and each row is passed into the matrix sequentially. Figure 4 shows the design implemented for a 3 x 3 array.

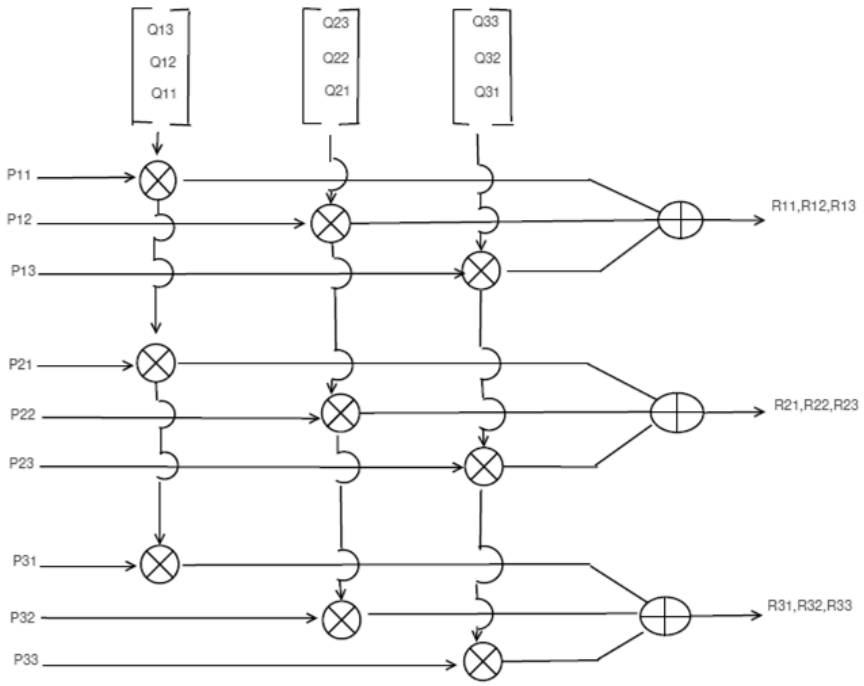

Figure 4: 3x3 Matrix Multiplier. [8]

Each column of the output matrix is produced in parallel until all columns have been produced. The architecture is scalable to any m x m matrix but requires additional passes when m > 4; additional elements are shifted in during additional cycles. The ASIC design does not include memory to store the inputs and outputs.

Both of the architectures referenced in this section will be used in comparison with the NNA architecture.

2.6. Classification Accuracy of 8-Bit Neural Networks for MNIST

Google has incorporated post-training quantization into their TensorFlow software, allowing 64-, 32-, and 16-bit networks to be converted into 8-bit networks with minimal loss in

inference accuracy. The activation function outputs are converted from floating point to integer by running input data through the floating-point model and scaling the maximum and minimum values from 127 to -128. Post-training quantization may introduce errors in the neural network that cannot be corrected, requiring the developer to modify the network, perform training and post-training quantization and measure the accuracy of the network. As evidence of the effectiveness of post-training quantization we developed two neural networks created in TensorFlow/Keras and trained them using the MNIST database. After post-training quantization we measured the accuracy of both networks, shown below as "Network 1" and "Network 2".

Layer (type)	Output Shape	Param #
input_layer (Conv2D)	(None, 26, 26, 32)	320
input_pooling (AveragePooling2D)	(None, 13, 13, 32)	0
hidden_layer (Conv2D)	(None, 11, 11, 50)	14450
hidden_pooling (AveragePooling2D)	(None, 5, 5, 50)	0
flatten (Flatten)	(None, 1250)	0
dense_layer (Dense)	(None, 500)	
output_layer (Dense)	(None, 10)	

Total params: 645,280
Trainable params: 645,280
Non-trainable params: 0

Figure 5: Network 1.

Network 2 was configured with layers of 196 dropout neurons. Four dropout rates were tested to determine which achieved the best accuracy: 0%, 10%, 30% and 50%. A dropout rate of 10% achieved the highest accuracy.

```
 Layer (type)                 Output Shape              Param #
=================================================================
 input_layer (Dense)          (None, 196)               153860
 input_dropout (Dropout)      (None, 196)               0
 hidden_layer (Dense)         (None, 196)               38612
 hidden_dropout (Dropout)     (None, 196)               0
 output_layer (Dense)         (None, 10)                1970
=================================================================
Total params: 194,442
Trainable params: 194,442
Non-trainable params: 0
```

Figure 6: Network 2.

	Network 1	Network 2
32-bit Floating Point Accuracy	96.7%	97.0%
8-Bit Integer Accuracy	94.8%	97.1%

Figure 7: Network Accuracy.

After post-training quantization the accuracy of Network 1 decreased, and the accuracy Network 2 remained the same. By reducing the data types from 32-bit to 8-bit, the resources and power required to perform matrix multiplication are reduced.

This analysis will be included in a future feasibility study to determine if the NNA architecture can perform inferencing using 8-bit TensorFlow neural networks.

Chapter 3: Neural Network Accelerator (NNA) Architecture

The proposed NNA architecture was developed as a systolic array with specialized ALUs to reduce the time complexity of matrix multiplication and power consumption when implemented on a FPGA. The goal was to create the hardware resources necessary for matrix multiplication and optimize the architecture for neural network applications. The architecture was developed in Verilog HDL and is scalable to meet application requirements and FPGA resource constraints. The architecture was developed for embedded systems with size, weight, and power (SWaP) constraints. The following sections present the design of the three subsystems of the NNA architecture: compartment, systolic array, and top level (data flow and control).

3.1. Compartment

The compartment contains interconnects (West, East, North, and South Bus) and control signals that move data into and out of the compartment. Data meant for neighboring compartments, moves from West to East, and North to South. The Instruction Memory is an 8-bit, single port memory with a configurable depth, which stores instructions for the compartment received through the North Bus. When the compartment is not running instructions, instructions are written into memory using a 2-bit control signal from the top level:

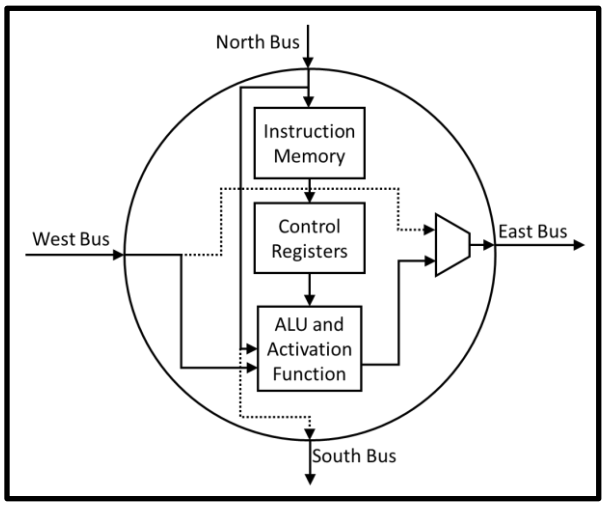

Figure 8: Compartment.

- Mode[1:0]: 00 = Reset, 01 = Read Address, 10 = Load Instructions, 11 = Run Instructions

When Mode is set to Reset, the compartment is idle. Each compartment contains a unique 8-bit address, configured when it is instantiated during synthesis providing every compartment in a column with a unique address, and a global address of 256 (0xFF). When Mode is set to Read Address, the North Bus value is stored in a register. When Mode is set to Load Instruction, the North Bus value is stored in the instruction memory. The unique address allows each compartment to process a unique set of instructions allowing the architecture to be optimized for specific applications such as convolution.

Each compartment contains control registers for the ALU and the multiplexers that direct the flow of data. Control registers are updated every clock cycle when the program counter (PC) is running and according to the instructions stored in memory. The compartments instruction set architecture (ISA) uses 8-bit instructions.

East Mux Select		Store	Activation Function		Clear	ALU	
7	6	5	4	3	2	1	0

Figure 9: Compartment ISA.

The East Mux Select bits route data to the East Bus interconnect: 00 = West Bus, 01 = MACC Result, 10 = Multiply Result, 11 = Activation Function Result. The Store bit is asserted when the East Bus output needs to be stored in memory. The Activation Function bits select the activation function: 00 = ReLU, and 01, 10, 11 = Reserved for Future Use. The Clear bit resets the accumulator in the ALU to zero. The ALU bits select the ALU operation: 00 = NOOP, 01 = Multiply, 10 = MACC (multiply and accumulate), 11 = Activation Function.

Each ALU contains a multiplier, adder, and activation function module and performs signed or unsigned integer operations. The ALUs operand A input is connected to the West Bus, operand B is connected to North Bus. The input of the adder module is the output of the multiplier module, creating a MACC. The input of the activation function module is the output of the MACC. With the three modules in series, the output of each module can be selected as the output of the ALU: multiply, MACC, or activation function unit. The 16-bit output of the ALU is scaled with a Verilog parameter that specifies the number of right-shifts to perform resulting in efficient scaling by powers of two. The inputs and outputs of the ALU are 8-bits, requiring minimal hardware resources. Thus, multiplication and addition require one clock cycle.

3.2. Systolic Array

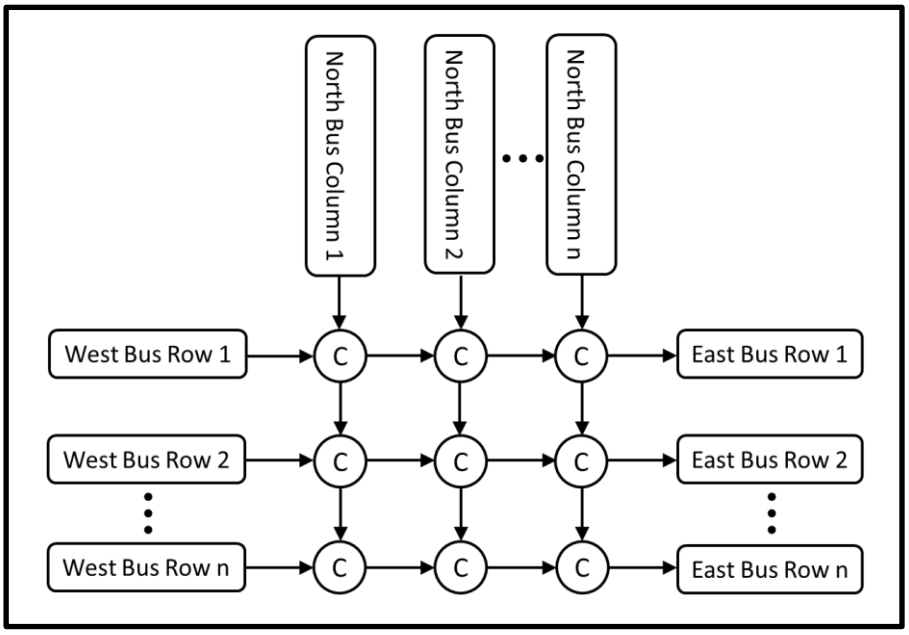

Figure 10: Systolic Array.

The systolic array connects multiple instantiations of the compartments in a n x n array, where the max value of n is 255 (256 is the global address). Data moves into the systolic

array from the West Bus and North Bus, and data moves out of the array through the East Bus as shown in Figure 10. The compartments are synchronized by a global clock and synchronous reset from the top-level. Data will shift from each compartment to the next every clock period. The configuration of the systolic array is parametrized to take advantage of the data bus width at the top-level. Here, n must be an integer value divisible by the number of bytes in the data bus.

3.3. Top-Level: Data Flow and Control

The top-level controls the flow of data to and from the systolic array using control signals, memories, and multiplexers. The North Bus input is connected to an 8-bit memory or the data bus using a multiplexor. Instructions for each compartment are shifted in through the data bus and inputs for the matrix operations are stored memory until the program counter is activated. The West Bus input is connected to memory which is used to store the inputs from the data bus and the outputs of the matrix operations from the compartments. After the inputs for the matrix operations are shifted into the systolic array, the compartments will assert a signal to store the outputs of the matrix operations into memory starting at address zero. The output of each row memory is connected to inputs of a multiplexer, allowing the output of the matrix operations to be sent to a processor or simple logic.

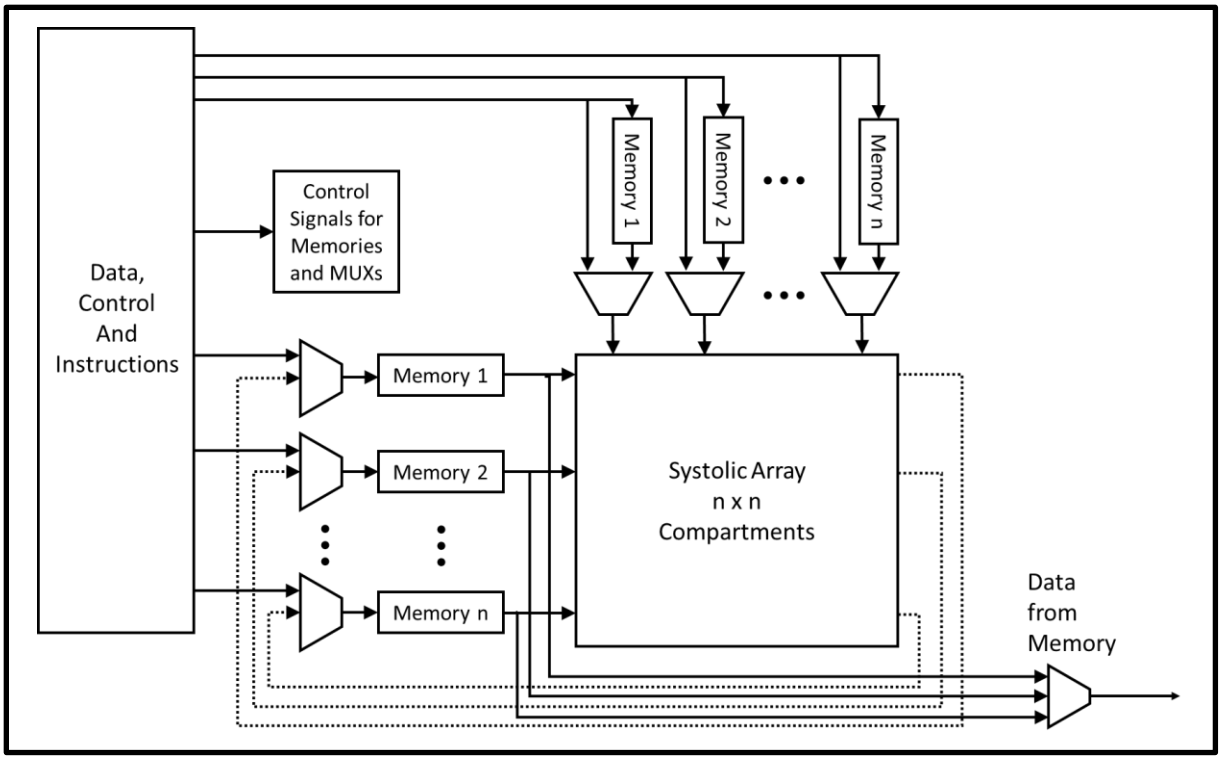

Figure 11: Top-Level of NNA Architecture.

The top-level module shown in Figure 11, has input and outputs that can be mapped to registers of an interface wrapper (AXI, APB, Wishbone, etc.). To reduce power consumption and hardware resources the memory depth and systolic array size can be configured using the parameters in the Verilog code. Each compartment is programmed to perform matrix multiplication via instructions loaded into local memory; the architecture can perform matrix multiplication on various matrix sizes without requiring the user to resynthesis the design for new applications.

The top-level module has ten input and two output ports, as shown in Figure 12.

```
// Inputs
input wire                                  clk;
input wire                                  rst;
input wire [1:0]                            mode;
input wire [addr_size-1:0]                  addr;
input wire [array_size-1:0]                 row_we;
input wire [array_size-1:0]                 col_we;
input wire                                  row_sel;
input wire                                  col_sel;
input wire [(bus_width*8)-1:0]              din;
input wire [(array_size/bus_width)-1:0] bank_sel;

// Outputs
output wire [(bus_width*8)-1:0]             dout;
output reg                                  done;
```

Figure 12: Top-Level Ports.

Chapter 4: Results

The NNA architecture was tested using Verilog test benches for the ALU, compartment, and top-level design; each behavioral simulation was run using the Riviera-Pro functional verification software. The coverage report for each testbench includes statement and branch coverage results. The Verilog test benches can be found in Appendices A through D.

4.1. Testbench Results

The ALU testbench selects operands “A” and “B” for the ALU and verify the output of the multiplier, MACC, and activation function modules. The operands are incremented each loop iteration to test all permutations, the test will continue running until a verification fails. The simulation also tests the five-bits of the instruction (4:0) that control ALU operations. The ALU module testbench passed with 98.9% statement coverage and 96.5% branch coverage.

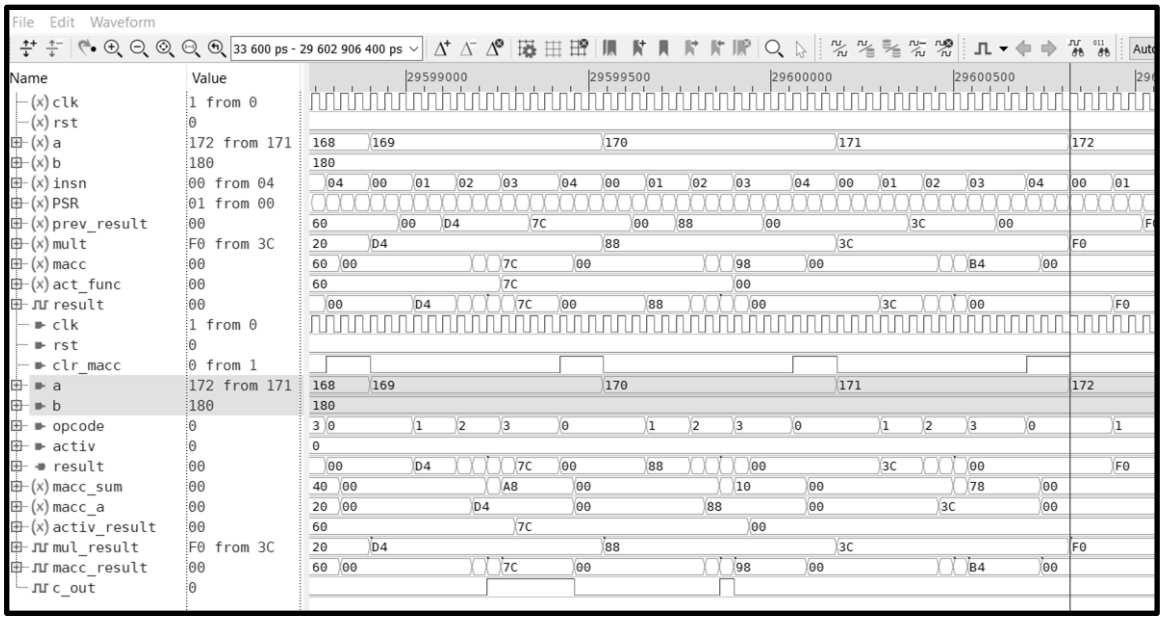

Figure 13: ALU Testbench Waveform.

The Compartment testbench ensures instructions and data flow through the module and bus interconnects, and the program counter is advancing and resetting when instructed. The Compartment testbench passed with 98.7% statement coverage and 97.5% branch coverage.

The Top-Level testbench ensures data flows from the top-level to the systolic array and ensures that the architecture can perform matrix operations. Instructions and matrix inputs are sent from the top-level to each compartment, the matrix operation is performed, and the results are shifted from the array to memory, and to the top-level module for verification by the testbench. The Top-Level testbench passed with 95.6% statement coverage and 88.8% branch coverage.

4.2. Performance

Performance results were obtained by synthesizing the architecture for three FPGAs: Microchip RTG4 and PolarFire, and Xilinx Kintex-7. The architecture was configured with a 32-bit data bus, 12 x 12 systolic array (144 compartments) and 1K of instruction memory for each Compartment, so that it could be implemented in all three FPGA fabrics and not exceed the available resources. Multiplication and addition are performed using shift registers and adders synthesized in the FPGA fabric, the Math (PolarFire), MACC (RTG4) and DSP (Kintex-7) blocks are not used, as shown in Figure 14.

Resources	Kintex-7	PolarFire	RTG4
LUTs - Used	19574	29526	31499
LUTs - %	19.85%	6.13%	20.75%
Flip-Flops (FF) / Registers - Used	14014	19190	19396
Flip-Flops (FF) / Registers - %	6.91%	3.99%	12.78%
RAM (1KB) - Used	132*	144	144
RAM (1KB) - %	20.31%	9.47%	68.90%
DSP / Math / MACC Blocks - Used	0	0	0
DSP /Math / MACC Blocks - %	0.00%	0.00%	0.00%

Figure 14: FPGA Resources.

The maximum operating frequency for each FPGA implementation and the place & route strategy used to achieve those results are shown in Figure 15 below.

FPGA	Max Frequency (MHz)	Strategy
Kintex-7	225	Performance_ExploreWithRemap(2018)
PolarFire	175	Timing-Driven, High-Effort Layout, Seed = 6
RTG4	90	Timing-Driven, High-Effort Layout, Seed = 6

Figure 15: Max Operating Frequency.

The NNA architecture performs matrix multiplication in four steps: load instructions, load matrices, run instructions, and return results. As evidence of the architecture's performance, a testbench was designed to perform matrix multiplication on two 8 x 8 matrices. The testbench moves data into the architecture through the 32-bit data bus and sets the top-level control signals to route the data to the Compartments and memories, after the architecture asserts the DONE flag the testbench retrieves the results from the memories.

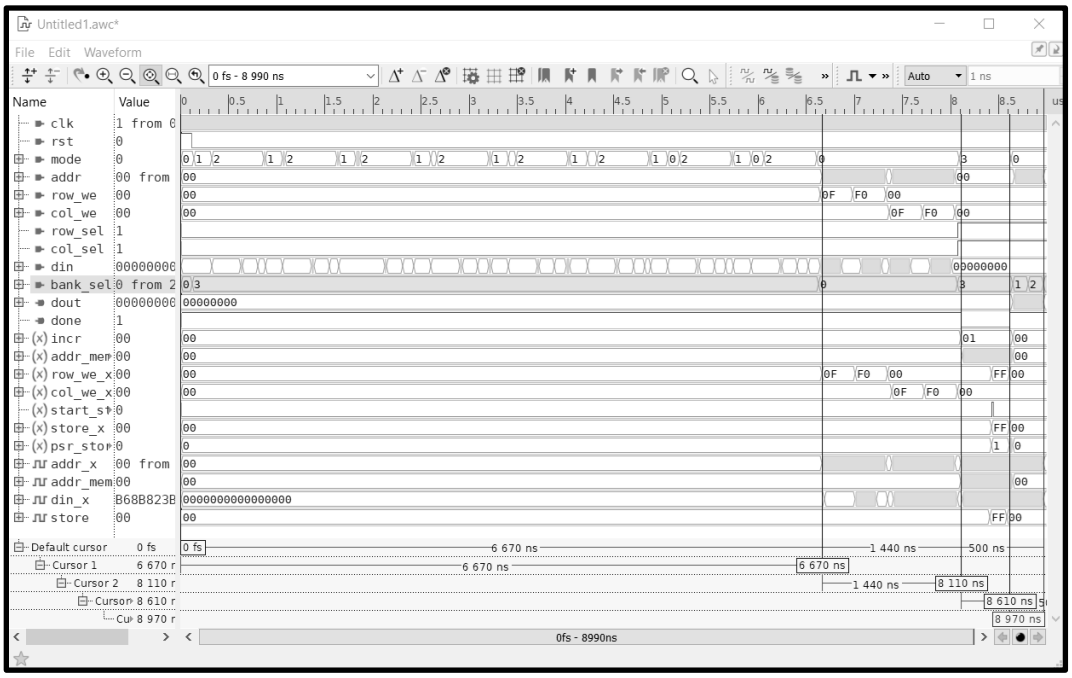

Figure 16: Waveform Timing Measurements.

Loading the instructions required 334 clock cycles, loading the matrices required 72 clock cycles, running the instructions required 25 clock cycles, and returning the results required 18 clock cycles; for a total of 449 clock cycles. When performing matrix multiplication on matrices of the same size, instructions are loaded into the architecture for the first operation and the same instructions are used for subsequent operations.

The NNA architecture is designed to provide CPUs an accelerator for matrix multiplication,

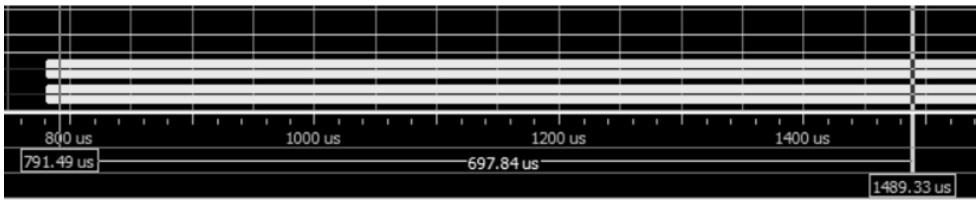

Figure 17: RISC-V Simulation Results.

we have chosen a RISC-V open-ISA soft-core CPU (MiV) to pair with the architecture. A behavioral simulation of the RISC-V running C code that implements naïve matrix multiplication using pointers for efficiency was created to determine the clock cycles required for the operation. The C code contains a matrix multiply function that takes two matrices as input, the code also contains software for CPU register initialization, so the time required for execution was measured from the first instruction of the matrix multiplication function to the last. Matrix multiplication on the RISC-V required ~34,892 clock cycles.

Based on these results, the RISC-V used in the simulation required ~77x more clock cycles to perform 8-bit matrix multiplication versus the NNA architecture.

Synthesis and place & route of the NNA architecture was performed in LiberoSoC for the PolarFire and RTG4, and Vivado for the Kintex7; the timing and resource utilization reports were used to estimate power consumption for each FPGA. Figure 18 below provides the total scenario power.

Device	Dynamic Power (85%)	Static Power (15%)	Total Power
Polarfire (MPF500T)	869 mW	154 mW	1,023 mW
Kintex7 (XC7K325T)	971 mW	167 mW	1,138 mW
RTG4 (RT4G150L)	2,104 mW	178 mW	2282 mW

Figure 18: FPGA Power Consumption.

Figure 19 below shows the tradeoff between power and frequency for each device.

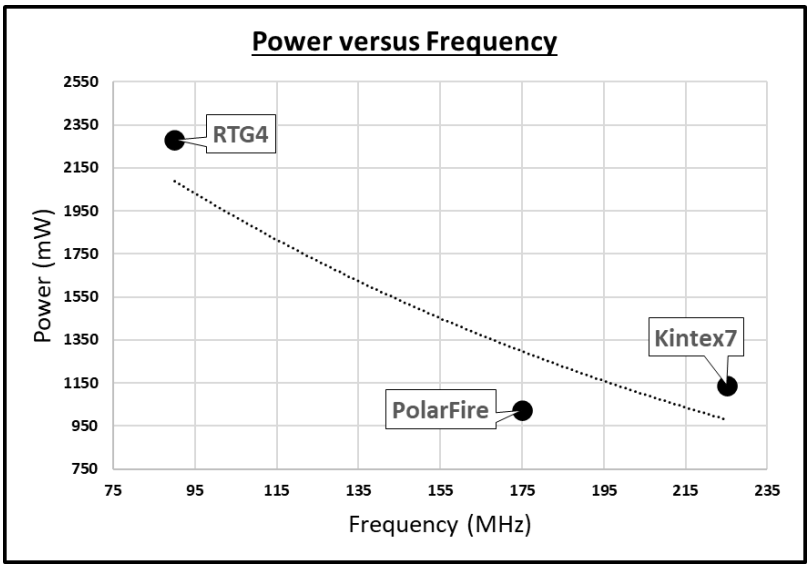

Figure 19: Power Versus Frequency.

The NNA architecture achieves the lowest power consumption when implemented on the PolarFire and the best power versus frequency performance when implemented on the Kintex7.

The largest array size that could be implemented on the PolarFire FPGA was 40 x 40 (1600 compartments) with a max frequency of 100 MHz.

4.3. Comparison

The SARS [2] hardware design presented in the International Journal of Computer Applications in 2011 performed 4-bit matrix multiplication on 3 x 3 matrices. When implemented on a Spartan-3E FPGA the architecture achieved a max operating frequency of 210.2 MHz. The NNA architecture performed 8-bit matrix multiplication on 8 x 8 matrices and achieved a max operating frequency of 225 MHz on a Kintex-7 FPGA. The NNA architecture showed a small performance improvement and improved upon the SARS design by performing 8-bit operations on larger matrices.

A DAE [7] hardware design from *Uppsala Universitet* published in 2018 (Hardware Accelerator of Matrix Multiplication on FPGAs), performed matrix multiplication on two 8 x 8 matrices. The design decomposed each 8 x 8 matrix into sixteen 2 x 2 matrices to perform matrix multiplication in parallel. The architecture contains five parts: memory modules, registers, cache, high performance computing (HPC) multiplier, and HPC accumulator. Reading data from RAM, performing matrix multiplication and writing the data back to RAM required 3681 clock cycles. The NNA architecture performed matrix multiplication in 449 clock cycles which is a significant increase in performance.

Researchers presenting at the 2020 IEEE 4th Conference on Information & Communication Technology (CICT) proposed the EES [8] hardware design which decomposed the m x m input matrices into 2 x 2 matrices and performed the calculations in parallel. Performing an 8 x 8 matrix multiplication on their architecture requires 66 clock cycles. Their design performs matrix multiplication but does not implement memory or logic to move data into or out of the architecture (see Figure 4). The NNA architecture is a complete system which uses memory to store data before and after matrix multiplication. For performing matrix multiplication alone, the NNA architecture requires 25 clock cycles which is a moderate increase in performance.

Chapter 5: Conclusion and Future Work

The NNA architecture was developed to perform matrix operations in lieu of a CPU in a heterogeneous system. The ALU operations are configurable using software to allow designers to perform matrix operations for a specific application without having to resynthesis the design; this feature differentiates the NNA architecture from other designs.

As part of our future research, we intend to synthesis the NNA architecture for an ASIC flow. The Verilog HDL has an option for implementing adders as Kogge-Stone or Carry-Ripple, the first uses propagate and generate signals for faster performance while the second offers a smaller footprint. Because a majority of the ASIC design area will be utilized by the RAMs, a customized RAM will be needed to reduce the amount of area used.

Deep Neural Networks (DNNs) require at least two hidden layers. Thus, our future goal for the architecture is to modify it to become a neuromorphic architecture; all the computations for the neural network would be performed in the NNA architecture without the need for an external CPU to control the hardware. The next step would be to determine the optimal memory hierarchy to allow the results of each matrix operation to be forwarded into the next matrix operation.

Convolution and cross correlation are the most widely used image processing operations, Convolutional Neural Networks (CNNs) contain convolutional layers which can be implemented using matrix multiplication but there are more efficient methods. In 2020 [9][12], members of the Image and Video Processing and Communications Lab (ivPCL) presented a novel method for increasing 2D convolution and cross-correlation performance

using CPUs, we intend to use their research to create 2D convolution optimized hardware for the NNA.

The analysis from chapter 2.6, will be used as part of a feasibility study to determine if the NNA architecture can perform inferencing. The study will implement an 8-bit TensorFlow neural network on the NNA architecture and verify the results against an implementation executed on a laptop/desktop computer.

Appendix A: Adder and Multiplier Testbench

```
module tb_alu();

  // Registers
  reg signed [7:0] a;
  reg signed [7:0] b;
  reg              clk;
  reg              rst;

  // Wires
  wire [7:0] sum1, sum2;
  wire       c_out1, c_out2;
  wire [15:0] prod1, prod2;

  // clock
  initial begin
    clk = 1'b0;
    rst = 1'b1;
    #40 rst = 1'b0;
  end

  always @(clk) begin
    #20 clk <= ~clk;
  end

  // KS Adder instantiation
  kogge_stone_adder #(.width(8)) adder1(
    .a(a),
    .b(b),
    .sum(sum1),
    .c_out(c_out1)
  );

  // Ripple Carry Adder instantiation
  n_adder #(.width(8)) adder2(
    .a(a),
    .b(b),
    .sum(sum2),
    .ovflw(c_out2)
  );

  // Full Adder Multiplier instantiation
  multiply #(.width(8)) multiply1(.a(a), .b(b), .result(prod1));

  // Kogge Stone Adder Multiplier instantiation
```

```
  multiply_ksa #(.width(8)) multiply2(.a(a), .b(b), .result(prod2));

  // Test logic
  always @(posedge clk) begin
    if (rst) begin
      a <= -8'h7;
      b <= 8'h3;
    end
    else begin
      a <= a + 1;
      b <= b + 1;
    end
  end
endmodule
```

Appendix B: ALU Module Testbench

```
module tb_alu_complete();

  // Parameters
  localparam width              = 8;
  localparam INITIAL            = 0;
  localparam OPCODE             = 1;
  localparam WAIT               = 2;
  localparam CHECK_CODE         = 3;
  localparam CLR_MACC           = 4;
  localparam WAIT2              = 5;
  localparam CHECK_MACC         = 6;

  // Registers
  reg                 clk;
  reg                 rst;
  reg [width-1:0]     a;
  reg [width-1:0]     b;
  reg [4:0]           insn;
  reg [4:0]           PSR;
  reg [7:0]           prev_result;
  reg [7:0]           mult;
  reg [7:0]           macc;
  reg [7:0]           act_func;

  // Wires
  wire [width-1:0] result;

  // Clock and reset
  initial begin
    clk <= 1'b0;
    rst <= 1'b1;
    macc <= 8'h0;
    #120 rst <= 1'b0;
  end

  always @(clk) begin
    #20 clk <= ~clk;
  end

  // Instantiate ALU module
  alu #(.width(width)) dut(
    .clk(clk),
    .rst(rst),
    .clr_macc(insn[2]),
```

```
  .a(a),
  .b(b),
  .opcode(insn[1:0]),
  .activ(insn[4:3]),
  .result(result)
);

// Test logic to test first five bits
// instruction bits used for ALU
always @(*) mult <= a * b;
always @(posedge clk) begin
  if (insn[2] == 1'b1)   macc <= 8'h0;
  else if (insn[2:0] == 3'h2) macc <= mult + macc;
end
always @(*) begin
  if (insn[1:0] == 2'h3) begin
    if (macc[7] == 1'b1) act_func <= 'h0;
    else act_func <= macc;
end
end

always @(posedge clk) begin
  if (rst) begin
    a <= 8'h0;
    b <= 8'h0;
    insn <= 5'h0;
    prev_result <= 8'h0;
  end
  else begin
    case(PSR)
      INITIAL: begin     // Increment operands
        a <= a + 1;
        if (a == 8'hFF) b <= b + 1;
        insn <= 5'h0;
        PSR <= OPCODE;
      end
      OPCODE: begin      // Increment opcode
        PSR <= WAIT;
      end
      WAIT: begin
        prev_result <= result;
        PSR <= CHECK_CODE;
      end
      CHECK_CODE: begin   // Check opcode results
        case(insn[1:0])
          2'h0: if(result != prev_result) $finish;
```

```
          2'h1: if(result != mult) $finish;
          2'h2: if(result != macc) $finish;
          2'h3: if(result != act_func) $finish;
        endcase

        if (insn[1:0] == 2'h3) PSR <= CLR_MACC;
        else begin
          insn[1:0] <= insn[1:0] + 1;
          PSR <= OPCODE;
        end
      end
      CLR_MACC: begin
        insn[2:0] <= 3'h4;
        PSR <= WAIT2;
      end
      WAIT2: PSR <= CHECK_MACC;
      CHECK_MACC: begin
        if(result != 'h0) $finish;
        else PSR <= INITIAL;
      end
      default: PSR <= INITIAL;
    endcase
  end
end

endmodule
```

Appendix C: Compartment Testbench

```
module tb_compartment();

  // Parameters
  localparam   INITIAL            = 0;
  localparam   LOAD_INSN_1   = 1;
  localparam   LOAD_INSN_2   = 2;
  localparam   LOAD_INSN_3   = 3;
  localparam   LOAD_INSN_4   = 4;
  localparam   LOAD_INSN_5   = 5;
  localparam   LOAD_INSN_6   = 6;
  localparam   LOAD_INSN_7   = 7;
  localparam   CHECK_INSN_1  = 8;
  localparam   CHECK_INSN_2  = 9;
  localparam   RUN_INSN_1     = 10;
  localparam   RUN_INSN_2     = 11;
  localparam   DONE               = 12;
  localparam   END                = 13;
  localparam   OPERAND_A      = 8'h3;
  localparam   OPERAND_B      = 8'h7;

  localparam   [7:0] INSN [15:0] = {8'h00, 8'h01, 8'h02, 8'h03,
                          8'h40, 8'h41, 8'h80, 8'h81,
                          8'hC0, 8'hC2, 8'h40, 8'h42,
                          8'h80, 8'h83, 8'hC0, 8'hC3};

  // Registers
  reg             clk;
  reg             rst;
  reg [1:0]       mode;
  reg [7:0]       west_bus;
  reg [7:0]       north_bus;
  reg [5:0]       PSR;
  reg [3:0]       count;

  // Wires
  wire            clk_out;
  wire            rst_out;
  wire [1:0]    mode_out;
  wire [7:0]    east_bus;
  wire [7:0]    south_bus;

  integer i;

  // Clock and reset
  initial begin
```

```
  clk <= 1'b0;
            rst <= 1'b1;
            #(120) rst <= 1'b0;
end

always @(clk) begin
  #(20) clk <= ~clk;
end

// Instantiate Compartment, address "5"
compartment #(.comp_addr(5), .width(8)) node1(
  .clk_in(clk),
  .rst_in(rst),
  .mode_in(mode),
  .clk_out(clk_out),
  .rst_out(rst_out),
  .mode_out(mode_out),
  .west_bus(west_bus),
  .north_bus(north_bus),
  .east_bus(east_bus),
  .south_bus(south_bus)
);

// Test logic for compartment
always @(posedge clk) begin
  if(rst) begin
    mode       <= 2'h0;
    west_bus   <= 8'h0;
    north_bus  <= 8'h0;
    count      <= 8'h1;
    PSR        <= INITIAL;
  end
  else begin
    case(PSR)
      INITIAL : begin
        PSR <= LOAD_INSN_1;
      end
      LOAD_INSN_1 : begin      // Read address "6"
        mode      <= 2'h1;
        north_bus <= 8'h6;
        PSR       <= LOAD_INSN_2;
      end
      LOAD_INSN_2 : begin
        PSR       <= LOAD_INSN_3;
      end
      LOAD_INSN_3 : begin
        if(node1.addr_valid) begin
          $display("Failed: Node addr_valid is '1'");
```

```
    $finish;
  end
  else begin
    north_bus <= 8'h5;
    PSR       <= LOAD_INSN_4;
  end
end
LOAD_INSN_4 : begin      // Read address "5"
  PSR       <= LOAD_INSN_5;
end
LOAD_INSN_5 : begin
  if(node1.addr_valid == 1'b0) begin
    $display("Failed: Node addr_valid is '0'");
    $finish;
  end
  else PSR <= LOAD_INSN_6;
end
LOAD_INSN_6 : begin      // Start loading instructions
  mode      <= 2'h2;
  north_bus <= INSN[0];
  PSR       <= LOAD_INSN_7;
end
LOAD_INSN_7 : begin      // Load remaining 15 instructions into memory
  if(count == 4'hF) begin
    north_bus <= INSN[count];
    PSR <= CHECK_INSN_1;
  end
  else begin
    count     <= count + 1;
    north_bus <= INSN[count];
  end
end
CHECK_INSN_1 : begin
  mode    <= 2'h0;
  PSR     <= CHECK_INSN_2;
end
CHECK_INSN_2 : begin
  for(i=0;i<16;i++) begin
    if(node1.insn_mem[i] != INSN[i]) begin
      $display("Instruction memory address %d failed", i);
      $finish;
    end
  end
  PSR       <= RUN_INSN_1;
  mode      <= 2'h3;
  count     <= 4'h0;
  west_bus  <= OPERAND_A;
  north_bus <= OPERAND_B;
```

```
      end
      RUN_INSN_1 : begin
       if(count == node1.pc) begin
         if(count < 4'hF) count <= count + 1;
       end
       else begin
         $display("Program counter is not incrementing");
         $finish;
       end

       if(count == 4'hF) PSR <= RUN_INSN_2;

       north_bus  <= north_bus + 1;
       west_bus   <= west_bus  + 1;
      end
      RUN_INSN_2 : begin
       mode    <= 2'h0;
       PSR     <= DONE;
      end
      DONE : begin
       north_bus <= 8'h0;
       west_bus  <= 8'h0;
       PSR       <= END;
      end
      END : begin
       $display("Simulation completed successfully!");
       $finish;
      end
      default: PSR <= INITIAL;
     endcase
   end
  end

endmodule
```

Appendix D: Top-Level Testbench

```
module tb_systolic_array();

  // Parameters
  localparam clock_period             = 20;   // Clock period
  localparam INITIAL                  = 5'h0;
  localparam LOAD_ADDR                = 5'h1;
  localparam LOAD_INSN                = 5'h2;
  localparam DISABLE_1                = 5'h3;
  localparam WRITE_ROW_1              = 5'h4;
  localparam WRITE_ROW_2              = 5'h5;
  localparam WRITE_ROW_3              = 5'h6;
  localparam DISABLE_2                = 5'h7;
  localparam WRITE_COL_1              = 5'h8;
  localparam WRITE_COL_2              = 5'h9;
  localparam WRITE_COL_3              = 5'hA;
  localparam DISABLE_3                = 5'hB;
  localparam RUN                      = 5'hC;
  localparam READ_1                   = 5'hD;
  localparam READ_2                   = 5'hE;
  localparam DONE                     = 5'hF;
  localparam scale                    = 0;

  localparam [31:0] insn_bank[0:29] = '{
      32'h02020202, 32'h02020202, 32'h02020202, 32'h02020202, 32'h02020202,
      32'h02020202, 32'h02020202, 32'h02020202, 32'h02020202, 32'h02020202,
      32'h02020202, 32'h02020202, 32'h42424242, 32'h04040404, 32'h04040404,
      32'h04040404, 32'h04040404, 32'h24242424, 32'h24242424, 32'h24242424,
      32'h00000000, 32'h00000000, 32'h00000000, 32'h00000000, 32'h00000000,
      32'h00000000, 32'h00000000, 32'h00000000, 32'h00000000, 32'h00000000};
  localparam [31:0] weights[0:14]     = '{
      32'h0000000F, 32'h00003344, 32'h00667788, 32'h00AABBCC, 32'h00EEFF00,
      32'h00121314, 32'h00161718, 32'h001A1B1C, 32'h00124500, 32'h00110000,
      32'h00000000, 32'h00000000, 32'h00000000, 32'h00000000, 32'h00000000};
  localparam [31:0] inputs[0:14]      = '{
      32'h00000002, 32'h00000301, 32'h00020104, 32'h00050206, 32'h00080107,
      32'h000A0305, 32'h00020102, 32'h00010201, 32'h00020100, 32'h00010000,
      32'h00000000, 32'h00000000, 32'h00000000, 32'h00000000, 32'h00000000};

  // Registers
  reg             clk;
  reg             rst;
  reg  [1:0]      mode;
  reg  [4:0]      addr;
  reg  [7:0]      row_we;
```

```
reg  [7:0]       col_we;
reg              row_sel;
reg              col_sel;
reg  [31:0]    din;
reg  [1:0]      bank_sel;

reg  [4:0]       PSR;
reg  [15:0]     counter;
reg  [7:0]       row_count;
reg  [2:0]       pad_count;
reg  [3:0]       addr_count;

reg  [15:0]      test1;
reg  [15:0]      test2;
reg  [15:0]      test3;
reg  [7:0]       y1,y2,y3,y4,y5,y6,y7,y8,y9;

// Wires
wire [31:0]     dout;
wire             done;

// Integers
integer unsigned insn_count;
integer unsigned i,error;
integer unsigned test_i;

// Clock and reset
initial begin
  clk <= 1'b0;
  rst <= 1'b1;
  #120 rst <= 1'b0;
end

always @(clk) begin
  #(clock_period/2) clk <= ~clk;
end

// Verify results
initial begin
  y1=0;
  y2=0;
  y3=0;
  y4=0;
  y5=0;
  y6=0;
  y7=0;
```

```
  y8=0;
  y9=0;
  test_i =0;
  #10;
  for(i=0;i<9;i++) begin
    y1=y1+inputs[i][7:0]*weights[i][7:0];
    y2=y2+inputs[i][7:0]*weights[i+1][15:8];
    y3=y3+inputs[i][7:0]*weights[i+2][23:16];
    y4=y4+inputs[i+1][15:8]*weights[i][7:0];
    y5=y5+inputs[i+1][15:8]*weights[i+1][15:8];
    y6=y6+inputs[i+1][15:8]*weights[i+2][23:16];
    y7=y7+inputs[i+2][23:16]*weights[i][7:0];
    y8=y8+inputs[i+2][23:16]*weights[i+1][15:8];
    y9=y9+inputs[i+2][23:16]*weights[i+2][23:16];
  end
  y1=y1 >> scale;
  y2=y2 >> scale;
  y3=y3 >> scale;
  y4=y4 >> scale;
  y5=y5 >> scale;
  y6=y6 >> scale;
  y7=y7 >> scale;
  y8=y8 >> scale;
  y9=y9 >> scale;
end

always @(posedge clk) begin
  if(rst) begin
    test1 <= 16'h0;
    test2 <= 16'h0;
    test3 <= 16'h0;
  end
  else if(test_i < 10) begin
    test1 <= test1 + (inputs[test_i][7:0]*weights[test_i+1][15:8]);
    test2 <= test2 + (inputs[test_i][7:0]*weights[test_i][7:0]);
    test3 <= test3 + (inputs[test_i+1][15:8]*weights[test_i][7:0]);
    test_i <= test_i + 1;
  end
end

// Instantiate systolic array
systolic_array array #(
  .scale(scale))
array(
  .clk(clk),
  .rst(rst),
```

```
  .mode(mode),
  .addr(addr),
  .row_we(row_we),
  .col_we(col_we),
  .row_sel(row_sel),
  .col_sel(col_sel),
  .din(din),
  .bank_sel(bank_sel),
  .dout(dout),
  .done(done)
);

// Test logic
always @(posedge clk) begin
  if(rst) begin
    mode <= 2'h0;
    addr <= 5'h0;
    row_we <= 8'h0;
    col_we <= 8'h0;
    row_sel <= 1'b0;
    col_sel <= 1'b0;
    din <= 32'h0;
    bank_sel <= 2'h0;
    row_count <= 8'h0;
    insn_count <= 20;
    counter <= 16'h0;
    pad_count <= 3'h0;
    addr_count <= 4'h0;
    PSR <= INITIAL;
  end
  else begin
    case(PSR)
      INITIAL: begin
        row_count <= 8'h0;
        insn_count <= 30;
        counter <= 16'h0;
        bank_sel <= 2'h3;
        pad_count <= 3'h0;
        addr_count <= 4'h0;
        PSR <= LOAD_ADDR;
      end
      LOAD_ADDR: begin
        counter <= 16'h0;
        mode <= 2'h1;
        din <= {row_count,row_count,row_count,row_count};
        pad_count <= 3'h0;
```

```
  if (addr_count < 4'h8) begin
    addr_count <= addr_count + 1;
    PSR <= LOAD_ADDR;
  end
  else begin
    row_count <= row_count + 1;
    PSR  <= LOAD_INSN;
  end
end
LOAD_INSN: begin
  addr_count <= 4'h0;
  din  <= insn_bank[counter];
  if(counter < insn_count) begin
    if (pad_count < row_count - 1) begin
      mode <= 2'h0;
      pad_count <= pad_count + 1;
    end
    else mode <= 2'h2;
    counter <= counter + 1;
    addr_count <= 4'h0;
    PSR <= LOAD_INSN;
  end
  else begin
    mode <= 2'h0;
    if (row_count < 8'h8) begin
      pad_count <= 3'h0;
      PSR <= LOAD_ADDR;
    end
    else begin
      counter <= 16'h0;
      PSR <= DISABLE_1;
    end
  end
end
DISABLE_1: begin
  mode <= 2'h0;
  row_sel <= 1'b0;
  col_sel <= 1'b0;
  counter <= 16'h0;
  addr <= 4'h0;
  bank_sel <= 2'h0;
  PSR <= WRITE_ROW_1;
end
WRITE_ROW_1: begin
  din <= inputs[addr];
  row_we <= 8'hFF;
```

```
  PSR <= WRITE_ROW_2;
end
WRITE_ROW_2: begin
  if(addr < 15) begin
    din <= inputs[addr+1];
    addr <= addr + 1;
    row_we <= 8'hFF;
  end
  else begin
    addr <= 4'h0;
    row_we <= 8'h00;
    din <= 32'h0;
    PSR <= DISABLE_2;
  end
end
WRITE_ROW_3: begin
  if(addr < 15) begin
    din <= inputs[addr+1];
    addr <= addr + 1;
    row_we <= 8'hF0;
  end
  else begin
    addr <= 4'h0;
    row_we <= 8'h00;
    din <= 32'h0;
    PSR <= DISABLE_2;
  end
end
DISABLE_2: begin
  mode <= 2'h0;
  row_sel <= 1'b0;
  col_sel <= 1'b0;
  counter <= 16'h0;
  addr <= 4'h0;
  row_we <= 8'h00;
  PSR <= WRITE_COL_1;
end
WRITE_COL_1: begin
  din <= weights[addr];
  col_we <= 8'hFF;
  PSR <= WRITE_COL_2;
end
WRITE_COL_2: begin
  if(addr < 15) begin
    din <= weights[addr+1];
    addr <= addr + 1;
```

```
    col_we <= 8'hFF;
  end
  else begin
    addr <= 4'h0;
    col_we <= 8'h00;
    din <= 32'h0;
    PSR <= DISABLE_3;
  end
end
WRITE_COL_3: begin
  if(addr < 15) begin
    din <= weights[addr+1];
    addr <= addr + 1;
    col_we <= 8'hF0;
  end
  else begin
    addr <= 4'h0;
    col_we <= 8'h00;
    din <= 32'h0;
    PSR <= DISABLE_3;
  end
end
DISABLE_3: begin
  mode <= 2'h0;
  row_sel <= 1'b1;
  col_sel <= 1'b1;
  counter <= 16'h0;
  addr <= 4'h0;
  row_we <= 8'h00;
  col_we <= 8'h00;
  bank_sel <= 2'h3;
  PSR <= RUN;
end
RUN: begin
  mode <= 2'h3;
  counter <= counter + 1;
  if ((counter > 4) && done) begin
    counter <= 16'h0;
    mode <= 2'h0;
    PSR <= READ_1;
  end
end
READ_1: begin
  mode <= 2'h0;
  bank_sel <= 2'h1;
  addr <= 5'h0;
```

```
        PSR <= READ_2;
      end
      READ_2: begin
        if(addr == 4'h2) PSR <= DONE;
        else addr <= addr + 1;
      end
      DONE: PSR <= DONE;
      default: begin
        mode <= 2'h0;
        addr <= 5'h0;
        row_we <= 8'h0;
        col_we <= 8'h0;
        row_sel <= 1'b0;
        col_sel <= 1'b0;
        din <= 32'h0;
        bank_sel <= 2'h0;
        row_count <= 0;
        insn_count <= 20;
        counter <= 16'h0;
        pad_count <= 3'h0;
        PSR <= INITIAL;
      end
    endcase
  end
 end

 // Verification Logic
 always @(posedge clk) begin
  if(rst) error <= 0;
  case(PSR)
    DISABLE_1: begin
      for(i=0;i<30;i++) begin   // Instruction
        if(array.array_row[0].array_column[0].genblk1.neuron.insn_mem[i] !=
insn_bank[i][7:0]) begin
          $display("Memory Failure: Row 0, Column 0, Insn %d", i);
          error = 1;
        end
        if(array.array_row[1].array_column[1].genblk1.neuron.insn_mem[i] !=
insn_bank[i][15:8]) begin
          $display("Memory Failure: Row 1, Column 1, Insn %d", i);
          error = 1;
        end
        if(array.array_row[2].array_column[2].genblk1.neuron.insn_mem[i] !=
insn_bank[i][23:16]) begin
          $display("Memory Failure: Row 2, Column 2, Insn %d", i);
          error = 1;
```

```
        end
        if(array.array_row[3].array_column[3].genblk1.neuron.insn_mem[i] !=
insn_bank[i][31:24]) begin
          $display("Memory Failure: Row 3, Column 3, Insn %d", i);
          error = 1;
        end
        if(array.array_row[4].array_column[4].genblk1.neuron.insn_mem[i] !=
insn_bank[i][7:0]) begin
          $display("Memory Failure: Row 4, Column 4, Insn %d", i);
          error = 1;
        end
        if(array.array_row[5].array_column[5].genblk1.neuron.insn_mem[i] !=
insn_bank[i][15:8]) begin
          $display("Memory Failure: Row 5, Column 5, Insn %d", i);
          error = 1;
        end
        if(array.array_row[6].array_column[6].genblk1.neuron.insn_mem[i] !=
insn_bank[i][23:16]) begin
          $display("Memory Failure: Row 6, Column 6, Insn %d", i);
          error = 1;
        end
        if(array.array_row[7].array_column[7].genblk1.neuron.insn_mem[i] !=
insn_bank[i][31:24]) begin
          $display("Memory Failure: Row 7, Column 7, Insn %d", i);
          error = 1;
        end
      end

      if(error) begin
        $display("Neuron Instruction Failure!");
        $finish;
      end
    end
    DISABLE_3: begin
      for(i=0;i<10;i++) begin   // Data
        // Check row memories
        if(array.array_row[0].row_memory.mem[i] != inputs[i][7:0]) begin
          $display("Input Failure: Row 0, Data %d", i);
          error = 1;
        end
        if(array.array_row[1].row_memory.mem[i] != inputs[i][15:8]) begin
          $display("Input Failure: Row 1, Data %d", i);
          error = 1;
        end
        if(array.array_row[2].row_memory.mem[i] != inputs[i][23:16]) begin
          $display("Input Failure: Row 2, Data %d", i);
```

```
          error = 1;
        end
        if(array.array_row[3].row_memory.mem[i] != inputs[i][31:24]) begin
          $display("Input Failure: Row 3, Data %d", i);
          error = 1;
        end
        if(array.array_row[4].row_memory.mem[i] != inputs[i][7:0]) begin
          $display("Input Failure: Row 4, Data %d", i);
          error = 1;
        end
        if(array.array_row[5].row_memory.mem[i] != inputs[i][15:8]) begin
          $display("Input Failure: Row 5, Data %d", i);
          error = 1;
        end
        if(array.array_row[6].row_memory.mem[i] != inputs[i][23:16]) begin
          $display("Input Failure: Row 6, Data %d", i);
          error = 1;
        end
        if(array.array_row[7].row_memory.mem[i] != inputs[i][31:24]) begin
          $display("Input Failure: Row 7, Data %d", i);
          error = 1;
        end
        // Check column memories
        if(array.array_row[0].array_column[0].genblk2.col_memory.mem[i] !=
weights[i][7:0]) begin
          $display("Weight Failure: Col 0, Data %d", i);
          error = 1;
        end
        if(array.array_row[0].array_column[1].genblk2.col_memory.mem[i] !=
weights[i][15:8]) begin
          $display("Weight Failure: Col 1, Data %d", i);
          error = 1;
        end
        if(array.array_row[0].array_column[2].genblk2.col_memory.mem[i] !=
weights[i][23:16]) begin
          $display("Weight Failure: Col 2, Data %d", i);
          error = 1;
        end
        if(array.array_row[0].array_column[3].genblk2.col_memory.mem[i] !=
weights[i][31:24]) begin
          $display("Weight Failure: Col 3, Data %d", i);
          error = 1;
        end
        if(array.array_row[0].array_column[4].genblk2.col_memory.mem[i] !=
weights[i][7:0]) begin
          $display("Weight Failure: Col 4, Data %d", i);
```

```
          error = 1;
        end
        if(array.array_row[0].array_column[5].genblk2.col_memory.mem[i] !=
weights[i][15:8]) begin
          $display("Weight Failure: Col 5, Data %d", i);
          error = 1;
        end
        if(array.array_row[0].array_column[6].genblk2.col_memory.mem[i] !=
weights[i][23:16]) begin
          $display("Weight Failure: Col 6, Data %d", i);
          error = 1;
        end
        if(array.array_row[0].array_column[7].genblk2.col_memory.mem[i] !=
weights[i][31:24]) begin
          $display("Weight Failure: Col 7, Data %d", i);
          error = 1;
        end
      end

      if(error) begin
        $display("Input/Weight Failure!");
        $finish;
      end

    end
    READ_1: begin
      if(array.array_row[0].row_memory.mem[2] != y1) $display("Y1 failed!");
      if(array.array_row[0].row_memory.mem[1] != y2) $display("Y2 failed!");
      if(array.array_row[0].row_memory.mem[0] != y3) $display("Y3 failed!");
      if(array.array_row[1].row_memory.mem[2] != y4) $display("Y4 failed!");
      if(array.array_row[1].row_memory.mem[1] != y5) $display("Y5 failed!");
      if(array.array_row[1].row_memory.mem[0] != y6) $display("Y6 failed!");
      if(array.array_row[2].row_memory.mem[2] != y7) $display("Y7 failed!");
      if(array.array_row[2].row_memory.mem[1] != y8) $display("Y8 failed!");
      if(array.array_row[2].row_memory.mem[0] != y9) $display("Y9 failed!");
    end
    DONE: begin
      $display("Simulation complete!");
      $finish;
    end
  endcase
 end

endmodule
```

References

[1] N.A. Campbell et al. *Biology: Concepts & Connections, 6th Edition*, pp. 563-576, 2009.

[2] M. Vucha, A. Rajawat, "Design and FPGA Implementation of Systolic Array Architecture for Matrix Multiplication," *International Journal of Computer Applications (0975-8887)*, vol. 26, Jul 2011.

[3] A. Geron, *Hands-On Machine Learning with Scikit-Learn & TensorFlow*, Sebastopol: O'Reilly Media, 2017.

[4] W. J. Dally, Y. Turakhia, S. Han, "Domain-specific hardware accelerators," *Communications of the ACM*, vol 63, pp. 48-57, Jul 2020.

[5] H. T. Kung, "Let's Design Algorithms for VLSI Systems," in *CALTECH Conference on VLSI*, Pasadena, CA, 1979

[6] G. Maan, "Hardware acceleration of matrix multiplication," BS thesis, Dept. Computer Science, Universiteit Leiden, The Netherlands, 2019.

[7] Z. Chen, "Hardware Accelerator of Matrix Multiplication on FPGAs," thesis, Dept. Information Technology, Uppsala Universitet, Uppsala, Sweden, 2018

[8] M. Shanmugakumar, V. S. M. Srinivasavarma, N. Mahammad, "Energy Efficient Hardware Architecture for Matrix Multiplication," in *2020 IEEE 4th Conference on Information & Communication Technology (CICT)*, Chennai, India, 2020

[9] C. Carranza, D. Llamocca, M. Pattichis, "Fast and Scalable 2D Convolutions and Cross-correlations for Processing Image Databases and Videos on CPUs," in *IEEE Southwest Symposium on Image Analysis and Interpretation (SSIAI) 2020*, Sante Fe, NM, 2020.

[10] R. Duan, H. Wu, R. Zhou, “Faster Matrix Multiplication via Asymmetric Hashing,” Arvix [Online], Nov 9 2022, Available: https://arxiv.org/pdf/2210.10173.pdf

[11] J. Alman, V.V. Williams, “A Refined Laser Method and Faster Matrix Multiplication,” Arvix [Online], Oct 12 2020. Available: https://arxiv.org/pdf/2010.05846.pdf

[12] C. Carranza, M. Pattichis, D. Llamocca, “Fast and Parallel Computation of the Discrete Periodic Radon Transform on GPUs, multi-core CPUs and FPGAs,” in *2018 25th IEEE International Conference on Image Processing (ICIP)*, Athens, Greece, 2018, pp. 4158-4162

[13] AMD/Xilinx, “System-Level Benefits of the Versal Platform”, WP539 v1.2, Feb 15, 2022