FLORIDA STATE UNIVERSITY

FAMU-FSU COLLEGE OF ENGINEERING

FPGA-BASED ACCELERATOR GENERATION FOR ARTIFICIAL NEURAL NETWORKS

By

CHRISTOPHER WADE JOHNSON

A Thesis submitted to the
Department of Electrical Engineering
in partial fulfillment of the
requirements for the degree of
Master of Science

2024

Copyright © 2024 Christopher Wade Johnson. All Rights Reserved.

Christopher Wade Johnson defended this thesis on April 5th, 2024.

The members of the supervisory committee were:

Uwe Meyer-Baese
Advisor

Linda DeBrunner
Committee Member

Simon Foo
Committee Member

The Graduate School has verified and approved the above-named committee members, and certifies that the thesis has been approved in accordance with university requirements.

I dedicate this work to my late brother, Collin, who taught me everything about computers from a young age, and instilled the passion I have for the field today. He was a brilliant cyber-security professional, and my best friend.

I also dedicate this work to my parents, Wade and Christel, who have encouraged and supported me throughout my education, and I hope this work can inspire my niece and nephews to pursue higher education.

ACKNOWLEDGMENTS

I want to thank my advisor, Dr. Uwe Meyer-Baese, for his guidance, support, patience, and technical advise throughout this process. His courses and books provided the foundation of knowledge and skills that made this research possible.

I also want to thank my committee members, Dr. Linda DeBrunner and Dr. Simon Foo, for their time, support, and instruction. Their courses were also pivotal to the accomplishments made in this paper.

I thank all faculty of the ECE department, and my fellow lab-mates for their support and help throughout my time in graduate school.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

LIST OF SYMBOLS

$\hat{y}$: Output of an Artificial Neural Network
q: Number of categories of a classification task
x: An image
d: Number of input features
x_i: Input feature i
$\mathbb{R}$: Real number space
y: A one-hot encoded label
ψ: An activation function
T_j : FPGA Transistor Junction Temperature
Fmax: Maximum clock frequency

LIST OF ABBREVIATIONS

ANN: Artificial Neural Network
AI: Artificial Intelligence
FPGA: Field-Programmable Gate Array
CPU: Central Processing Unit
GPU: Graphics Processing Unit
NIST: National Institute of Standards and Technology
MLP: Multi-Layer Perceptron
ReLU: Rectified Linear Unit
IEEE: Institute of Electrical and Electronics Engineers
SoC: System On a Chip
TSMC: Taiwan Semiconductor Manufacturing Company
HPS: Hard Processor System
LAB: Logic Array Block
DSP: Digital Signal Processing
ALM: Adaptive Logic Module
MLAB: Memory Logic Array Block
RAM: Random Access Memory
ROM: Read-Only Memory
FIFO: First-In, First-Out
SRAM: Static Random Access Memory
LUT: Look-Up Table
LED: Light-Emitting Diode
SDRAM: Static-Dynamic Random Access Memory
FBGA: Fine-Pitch Ball Grid Array
LE: Logic Element
EDA: Electronic Design Automation
IP: Intellectual Property
GUI: Graphical User Interface
RISC: Reduced Instruction Set Computer
CI: Custom Instruction
ALU: Arithmetic Logic Unit
MAC: Multiply-Accumulate
JTAG: Joint Test Action Group
UART: Universal Asynchronous Receiver-Transmitter
PIO: Parallel Input/Output
WSL2: Windows Subsystem for Linux
SGD: Stochastic Gradient Descent
IDE: Integrated Development Environment
ANSI: American National Standards Institute
BSP: Board Support Package

ABSTRACT

Artificial neural networks (ANNs) are a pivotal component of highly successful modern Artificial Intelligence (AI) applications, such as OpenAI's ChatGPT [1]. FPGA's capabilities of parallelism and efficiency provide an opportunity to provide quicker computation of ANNs than both CPUs or GPUs, whilst consuming much less power in both data center and edge computing environments. A custom instruction component for the Nios II soft processor [2] and supporting software were developed to allow easy, configurable generation and deployment of high-performance ANN inference on FPGA devices. The custom instruction performs integer matrix-vector multiplications, with each column of the product computed in parallel with pipelined multiply accumulate units. Nios II embedded computer systems were created both with and without the generated custom instruction. FPGA device resource utilization and execution time were evaluated for each system when performing ANN-based image classification on the MNIST [3] handwritten digit database. Speedups of over 300x were achieved when using the custom instruction, at the expense of higher FPGA resource utilization.

CHAPTER 1

INTRODUCTION

Artificial neural networks (ANNs) have been a core component of in field artificial intelligence (AI). The computational graphs of ANNs resemble, or aim to emulate the function of biological neurons [4]. The idea of creating computational structures that mimic biology isn't new. In 1943, a pair of biophysicists (McCulloch and Pitts) first mathematically modeled the behavior of neurons and suggested their use for computation [5]. In 1957, a psychologist, Frank Rosenblatt proposed that such a computational machine (a *perceptron*) could be "taught" to perform human like functions [6]. Since then, a variety of new structures, optimization, and training techniques have been developed, but the incredible performance of modern compute devices has undoubtedly been the driving force of the technology's success.

CHAPTER 2

IMAGE CLASSIFICATION WITH ARTIFICIAL NEURAL NETWORKS

While ANN's have a variety of different uses, they are commonly used for *classification* tasks. A classification tasks entails making a prediction, $\hat{y}^{(i)}$, of which of q categories a particular instance of data $x^{(i)}$ belongs to [7]. The constituent components of an instance of data $x_i^{(i)}$, are referred to as *features*, and can take the form of a vector, matrix, or a k-th order tensor. A classification problem can have any discrete number q of possible categories, or *classes.* In image classification problems, the data of an image takes the form of a matrix, $x^{(i)} \in \mathbb{R}^{h \times w}$ (greyscale images), or a 3rd order tensor, $x^{(i)} \in \mathbb{R}^{c \times h \times w}$ (color images), where h is the vertical resolution, w is the horizontal resolution, and c is the number of different color channels per pixel [7]. For simplicity, we will only consider greyscale images in this work. An image classification *dataset* comprises of N pairs of images, $x^{(i)}$, and labels, $y^{(i)}$ where the labels indicate which class the image belongs to. A label y corresponding to a class i is represented using *one-hot encoding* [7] by a vector $y \in \mathbb{R}^q$, where

$$y_j = \begin{cases} 1 & j = i \\ 0 & j \neq i \end{cases}$$

Likewise, the output of an image classifier, $\hat{y} \in \mathbb{R}^q$, takes the same form, where we can view each element y_i of the output as the *confidence* that the image belongs to class i. It is important to note that in digital computing systems, the real number space $\mathbb{R}$ cannot be accurately represented. Instead, numbers can only be represented in a finite subset of $\mathbb{R}$ [8], such as finite bit width integer, fixed point, or floating point number systems.

2.1 Databases

There are many publicly available databases for training image classifiers. The MNIST database of handwritten digits [3], works as great database to use for initial testing of an image classifier. It is comprised of 70,000 (image, label) pairs. The images are derived from handwritten decimal numeric digits (0-9) collected by NIST from US census workers and high school students [9]. Therefore, it has $q = 10$ classes, where each class's label is represented by the one-hot encoding of it's corresponding decimal digit. The images are greyscale, with a 28 x 28 pixel resolution. The images' low resolution and single color channel allows even relatively small models to achieve low error rates. Like most machine learning databases, it's data is split into two sections: a training set, and a testing set. It's testing set includes 60,000 images, and the training set includes 10,000 images, which were randomly drawn from the same source [3]. Each image's 784 (28 x 28) pixels are encoded as unsigned 8-bit integers, where the lowest value (0) represents white, the highest value (255) represent black, and the values (1-254) indicate varying shades of gray.

The EMNIST database [10] extends the MNIST database to include upper-case and lowercase English letters. It uses the same image format as MNIST, which allows the use of similar classifier structures. It was developed to create a more challenging classification task for neural network image classifiers [10]. It contains images of decimal digits (10), lowercase letters (26), and uppercase letters (26), for a total of $q = 62$ classes. It contains many more data-points than the MNIST database: in total, 731,668 training images and 82,587 testing images. It contains 344,307 training and 58,646 testing images of numeric digits [10], which far surpasses the original MNIST database.

2.2 Multi-Layer Perceptron

Multi-Layer Perceptrons (MLPs), are one of the earliest types of ANN. They extend the idea of Rosenblatt's perceptron [6] with a number of interconnected *hidden layers*, each having a number of *neurons* (or *nodes*), in addition to an *output layer* [4].

MLP's use *fully-connected* hidden layers, where each node of the layer is connected to the output of each node of the previous layer, or for the 1st hidden layer, each the input features [4]. A MLP with two fully-connected hidden layers is shown in Figure 2.1. Each connection has an associated *weight.* A node sums the weighted connections and adds a *bias.* It then applies a nonlinear *activation function* ψ, to the sum [4]. Without the activation function, a hidden layer would simply be a linear affine mapping. The activation function provides the nonlinearity that mimics that of a biological neuron. Typically, differentiable functions such as the logistic function or hyperbolic tangent function are used as ψ [4], but modern ANNs often use the the simple ReLU (rectified linear unit) function as ψ [7]. ReLU is defined as,

$$\text{ReLU}(x) = max(x, 0).$$

The output of a single node with weights $(w_1, ...w_n)$, and bias b, connected to a previous layer with n nodes with outputs $(a_1, ..., a_n)$, can then be described as,

$$\psi((\sum_{i=1}^{n} w_i a_i) + b)$$

Note that output nodes do not need to apply an activation function for classification tasks. The prediction made by a MLP is simply the class belonging to the element in the MLP's output vector with the largest value.

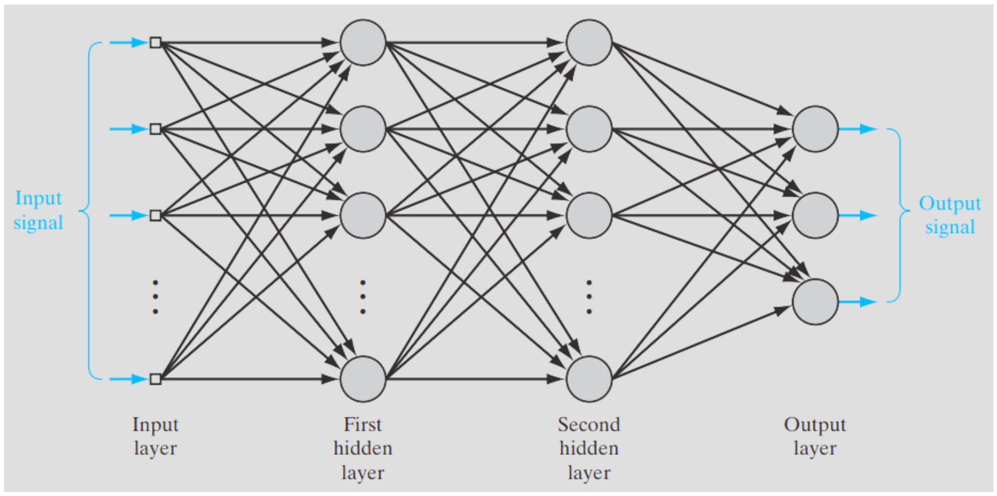


Figure 2.1: MLP Example [4]

2.3 Quantized Neural Networks

While training artificial neural networks, a floating point representation of a ANN's inputs, outputs, intermediate products, and parameters is most often used, as it allows for more accurate approximations of gradients during optimization. The IEEE-754 64-bit and 32-bit are commonly used, but other non-standard representations are used for specific hardware and use cases.

However, floating-point formats are not ideal for use in inference, especially when deployment is necessary on the edge, with embedded systems. Floating point operations typically take more device area, use more power, and use more clock cycles than an equivalent integer operation.

In essence, quantization is the process of rounding floating point numbers to the nearest integer. A detailed discussion of the neural network quantization methods used in this paper is given in [11].

CHAPTER 3

CYCLONE V SOC ARCHITECTURE

The Cyclone V SoC device family is built on TSMC's 28-nm low power (28LP) process technology [12]. It consists of an FPGA fabric, a dual-core ARM Cortex-A9 hard processor system (HPS), high speed FPGA-HPS interconnects, and supporting hard peripheral blocks, as shown in Figure 3.1.

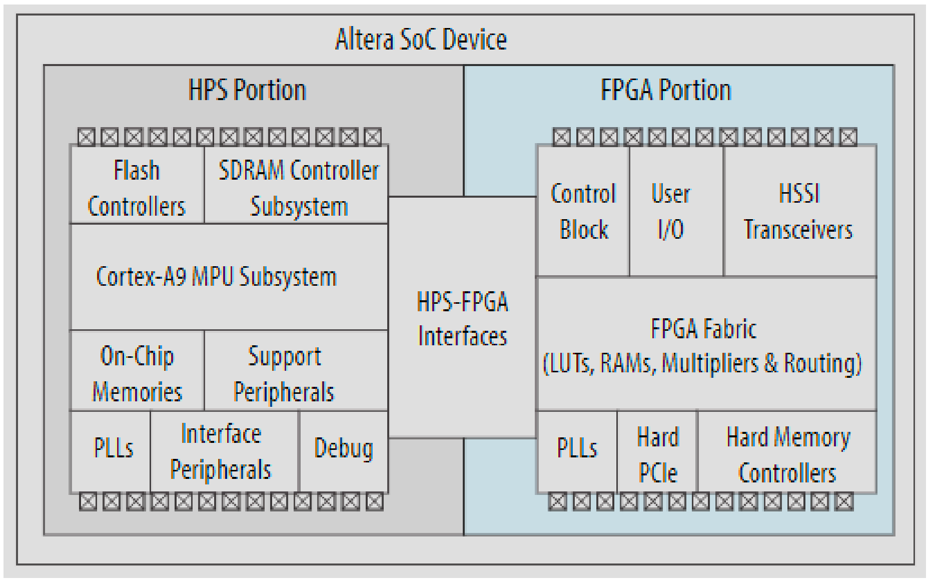


Figure 3.1: Cyclone V SoC High-Level Block Diagram [13]

The FPGA fabric contains Logic Array Blocks (LABs), Memory Logic Array Blocks (MLABs), M10K internal memory blocks (M10Ks), and Variable-Precision DSP Blocks (DSPs). The FPGA fabric of a Cyclone V device is laid out physically on the chip, as depicted in Figure 3.2, with large regions of general purpose logic blocks, and adjacent rows of M10K and DSP blocks.

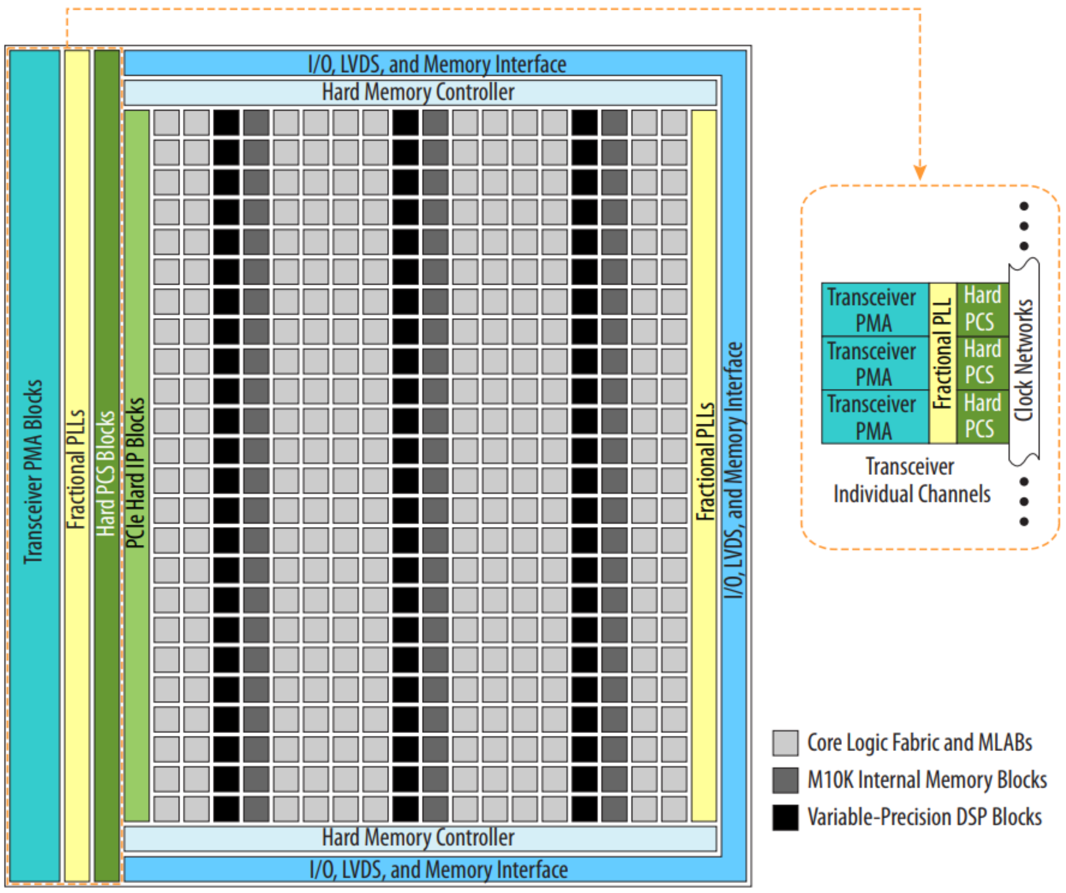


Figure 3.2: Cyclone V FPGA Floorplan [12]

3.1 Adaptive Logic Module (ALM)

The ALM is the fundamental logic block of the Cyclone V SoC architecture. An ALM is comprised of 2, 6-Input LUTs, 2 full adders, 2 registers, and multiplexers, as shown in Figure 3.3. They have 8 input data bits, 4 output data bits, 2 input bits from an adjacent ALM, and two output bits to a (different) adjacent ALM (see Figure 3.3). These connections to adjacent ALMs are extremely fast, and are referred to as the “carry chain” and “shared arithmetic chain”. As the name implies, they are often used for carry-ins and carry-outs for additions and subtractions that cannot fit in a single ALM [12].

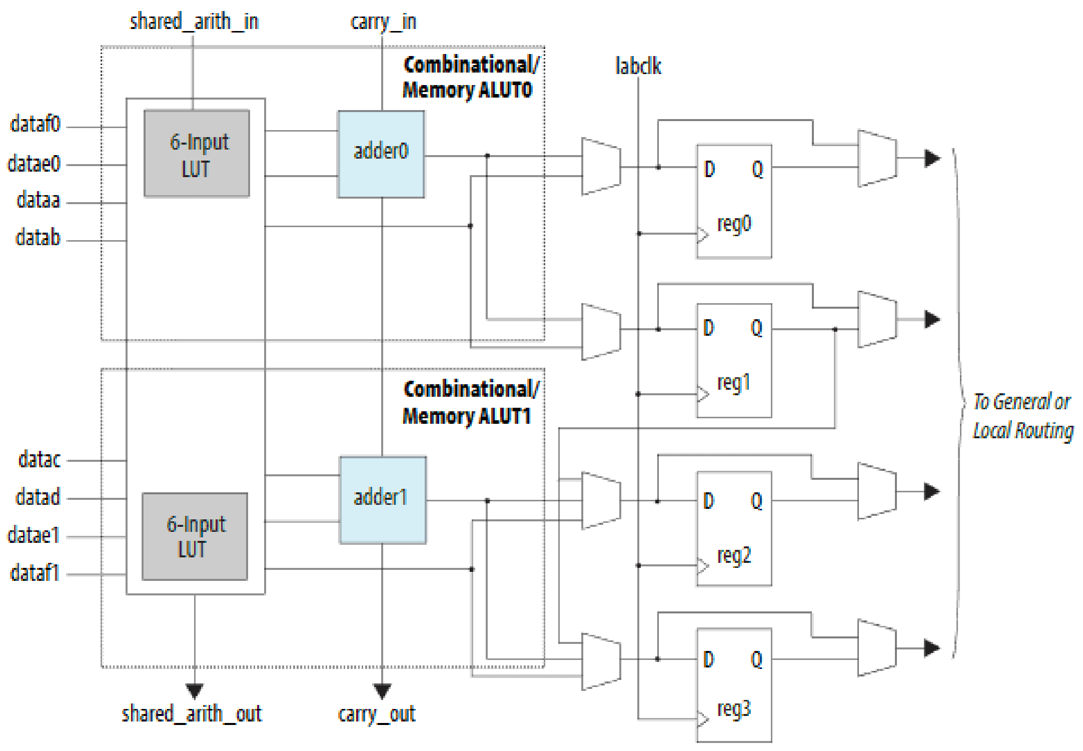


Figure 3.3: High-Level Block Diagram of a Cyclone V ALM [12]

3.2 Logic Array Block (LAB)

A Logic Array Block (LAB) is a group of 10 ALMs with associated control circuitry, as shown in Figure 3.4. ALMs in a LAB are connected to each-other through the “carry chain” and “shared arithmetic chain”.

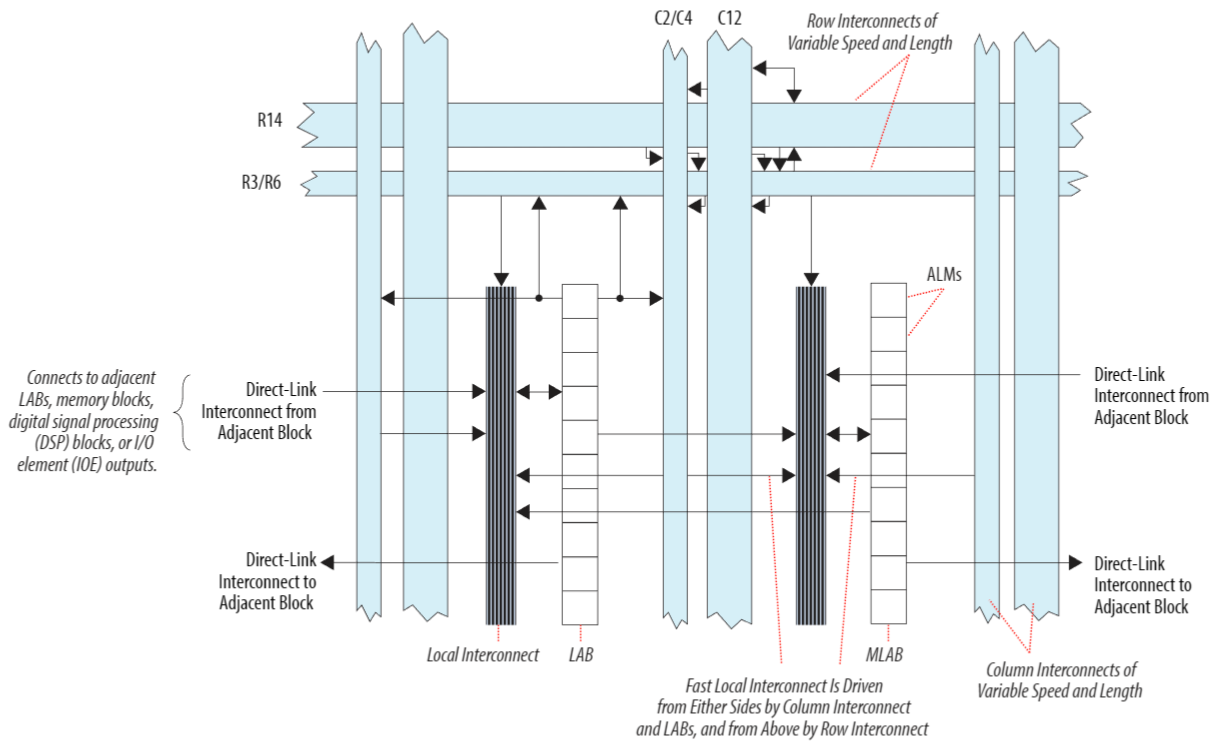

C2/C4
C12
Row Interconnects of Variable Speed and Length
R14
R3/R6
ALMs
Connects to adjacent LABs, memory blocks, digital signal processing (DSP) blocks, or I/O element (IOE) outputs.
Direct-Link Interconnect from Adjacent Block
Direct-Link Interconnect to Adjacent Block
Direct-Link Interconnect from Adjacent Block
Direct-Link Interconnect to Adjacent Block
Local Interconnect
LAB
MLAB
Fast Local Interconnect Is Driven from Either Sides by Column Interconnect and LABs, and from Above by Row Interconnect
Column Interconnects of Variable Speed and Length

Figure 3.4: Structure of a Cyclone V LAB [12]

3.3 Embedded Memory

Cyclone V devices have a variety of options for a hardware designer to consider when memory is needed. While off-chip memory might be used when a large amount of data must be stored, on-chip, embedded memory is preferable to use when possible because of it's vastly increased speed. Embedded memory is relatively small compared to off-chip memory modules, but they have fast interconnections and are tightly coupled with logic resources. Cyclone V devices have two types of embedded memory: Memory Logic Array Blocks (MLABs) and dedicated M10K Blocks. They can be configured as a variety of different memory types.

A Memory Logic Array Block (MLAB) is a specialized LAB that can function as fast on-chip memory. Like a LAB, and MLAB is comprised of 10 ALMs (see Figure 3.5). A quarter of a devices available LABs can be configured as MLABs. Each MLAB can provide up to 640 bits of memory. They can be configured as a single-port RAM, simple dual-port RAM, shift-register, ROM, or FIFO buffer [14]. MLABs have a fast local interconnect to adjacent LABs, making them ideal for use in

DSP applications. MLABs operate at a maximum of 420MHz, which is faster than that of M10K Blocks [12].

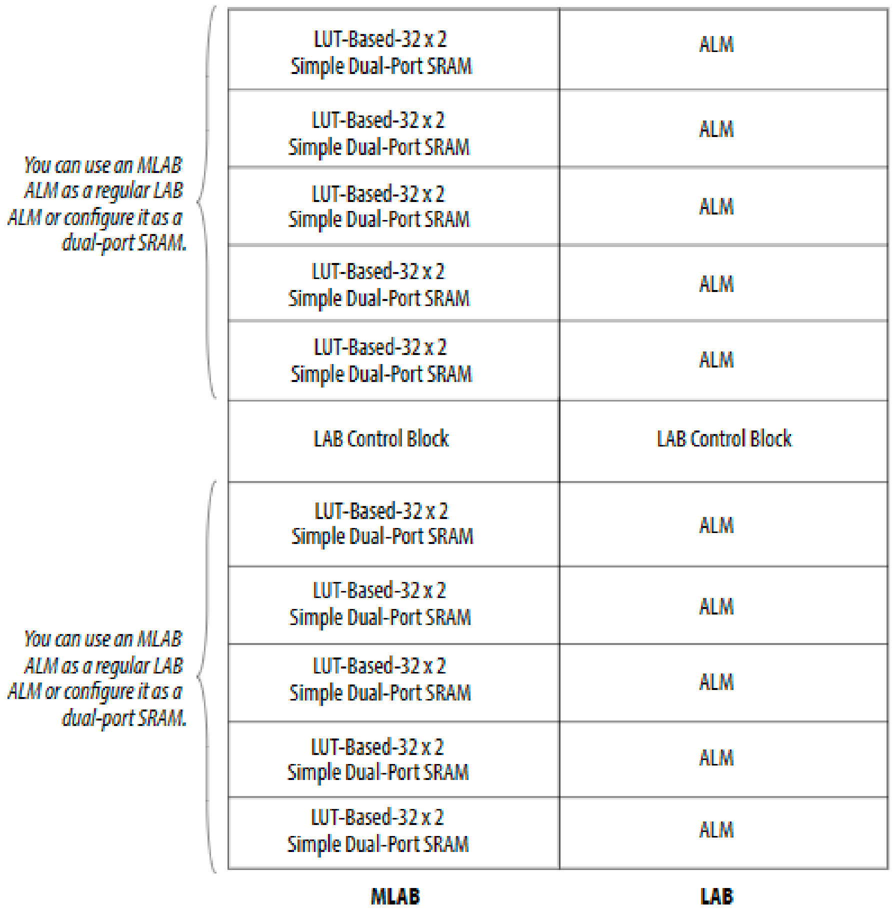

LUT-Based-32 x 2 Simple Dual-Port SRAM
ALM
LAB Control Block
LAB Control Block
You can use an MLAB ALM as a regular LAB ALM or configure it as a dual-port SRAM.
You can use an MLAB ALM as a regular LAB ALM or configure it as a dual-port SRAM.
MLAB
LAB

Figure 3.5: Cyclone V MLAB [12]

Cyclone V devices contain many "M10K" dedicated memory blocks. These memory blocks each provide 10 Kbits of memory. They provide the majority of embedded memory on a Cyclone V device, but aren't as tightly coupled to logic resources as MLABs. They are ideal for use with large arrays of memory, and they still provide multi-port access [12].

3.4 Variable-Precision DSP Blocks

Cyclone V Variable-Precision DSP blocks have a a variety of features, described in [12], and in Figure 3.6, that allow efficient and high performing implementations of basic multiplications, multiply-accumulate operations, finite impulse response filters, and even floating point arithmetic. A single DSP block can be configured as three 9x9-bit multipliers, two 18x18-bit multipliers, or a single 27x27-bit multiplier.

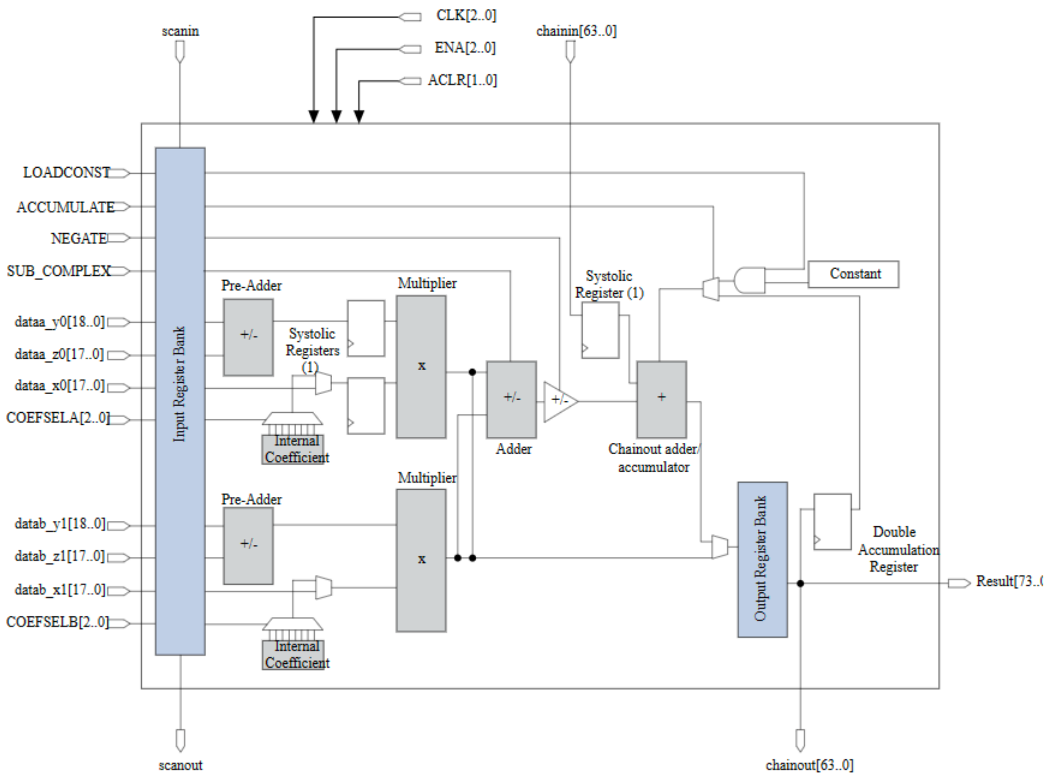


Figure 3.6: Structure of a Cyclone V Variable Precision DSP Block [12]

3.5 Terasic DE-10 Standard Development Board

The Terasic DE-10 Standard Development Board was used to test the hardware system. It contains an Intel/Altera Cyclone V 5CSXFC6D6F31C6N SoC chip and wide variety of peripheral devices, as shown in Figure 3.7, that allow development for a variety of different types of applications [15]. In this work, only the USB Blaster II, Switches, LEDs, and SDRAM peripherals were utilized. The specifications of the 5CSXFC6D6F31C6N chip are shown in Table 3.1.

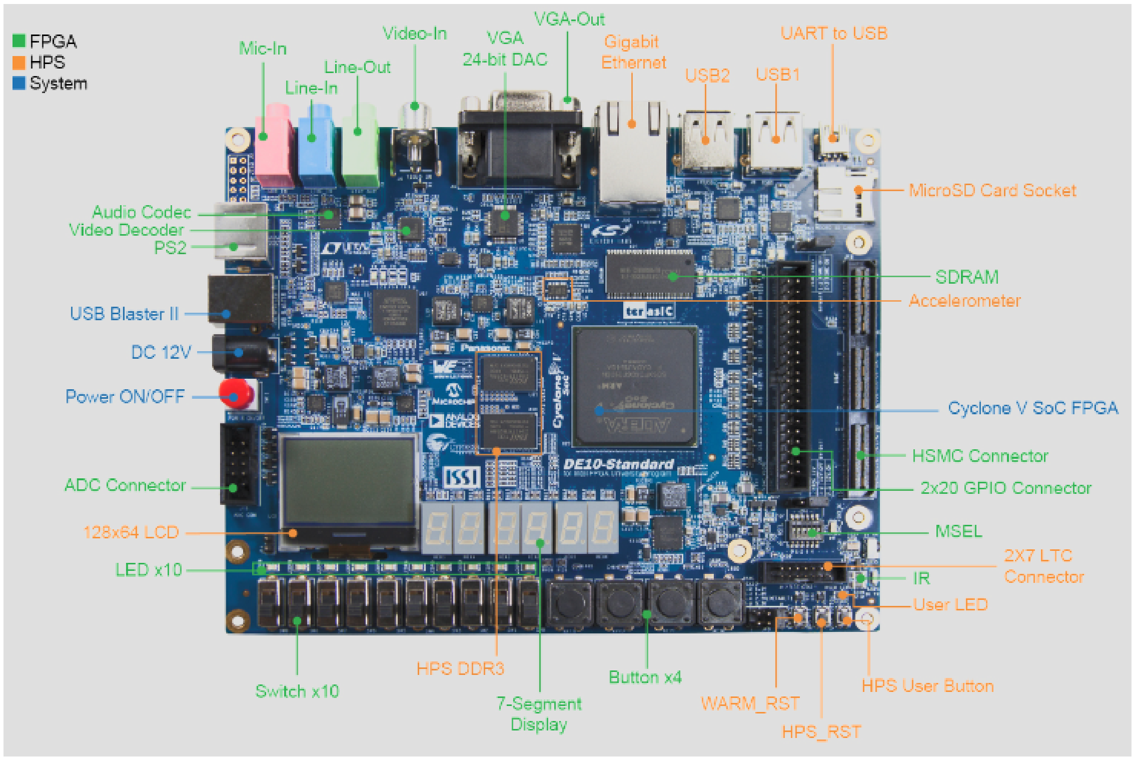


Figure 3.7: DE-10 Development Board Layout [15]

Table 3.1: Specifications of Cyclone V 5CSXFC6D6F31C6N SoC [12]

Attribute	Value
Manufacturing Process	TSMC 28nm low-power (28LP)
Package	FBGA (896 pin, lead-free)
Operating Temperature	$T_J = 0°$ C to $85°$ C
FPGA Fabric Speed Grade	'6' (fastest)
LE's	110,000
ALM's	41,910
M10K Blocks	553
MLAB Blocks	994
Variable-Precision DSP Blocks	112
18x18 Multipliers	224
Registers	166,036
M10K Memory	5,570 kB
MLAB Memory	621 kB
Total Memory	6,151 kB

CHAPTER 4

HARDWARE DESIGN

The embedded computer systems were developed with the *Platform Designer* tool, which is integrated into the *Quartus Prime 18.1 Standard Edition* EDA software. It contains a library of parameterized IP blocks, and mechanisms for creating IP blocks, that allow a user to create custom computer systems with an intuitive GUI. IP blocks are connected with an *Avalon* switch fabric [16]. The *Avalon* switch fabric has clock/reset, memory mapped, interrupt, streaming, and conduit interfaces [17].

The accelerators were designed around integration with a Nios II configurable soft processor as a custom instruction. `SystemVerilog` was used to describe the logic of the accelerator circuits, and to create testbenches for verification.

4.1 Nios II Soft Processor

The Nios II Soft Processor was developed by Intel/Altera as a configurable soft-core processor. It is a three-address, little-endian machine, with a 32-bit word length, instruction length, and address space. It is available in Intel/Altera's EDA tool, Quartus Prime, as an IP block. It uses a proprietary RISC ISA with a GNU based `C/C++` compiler [2]. As of Quartus Prime 18.1, it comes in two variants: Nios II/f ("fast") and Nios II/e ("economy"). The /f variant utilizes a 6-stage pipeline, whereas the /e variant takes a minimum of six clock cycles per instruction. The /f variant has much greater configuration options than the /e variant. It supports optional data and instruction caches, tightly coupled memories, dynamic or static branch prediction, and hardware multiply, divide and shift implementations [18]. While the /f variant achieves much higher performance than /e, it utilizes significantly more device resources. Importantly, the /e variant is free to use, while the /f variant requires a paid license.

4.2 Nios II Custom Instructions

Both Nios II/e and Nios II/f cores support custom instruction (CI) implementations [18]. Custom instructions are implemented in the Platform Designer tool as a *component* using the `nios_custom_instruction_slave` signal interface. Custom instruction logic is connected directly to the ALU of a Nios II core [19], as shown in Figure 4.1. Because the FPGA fabric is not tightly integrated into the HPS subsystem's ALU, the HPS does not support custom instructions.

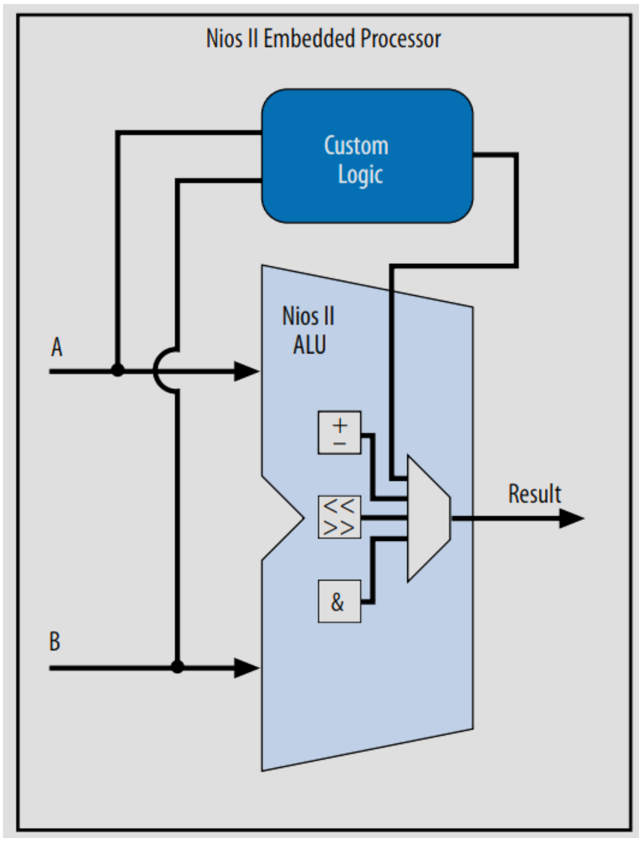


Figure 4.1: Nios II Custom Logic Integration [19]

The Nios II processor supports several different types of custom instructions. They vary on their input and output interfaces, as well as the number of cycles to complete. The different types of custom instructions are illustrated in Figure 4.2. Combinational instructions use only two input word operand ports, and one output word operand port. Multi-cycle instructions use additional ports to control timing, and extended instructions have an additional opcode extension field [19].

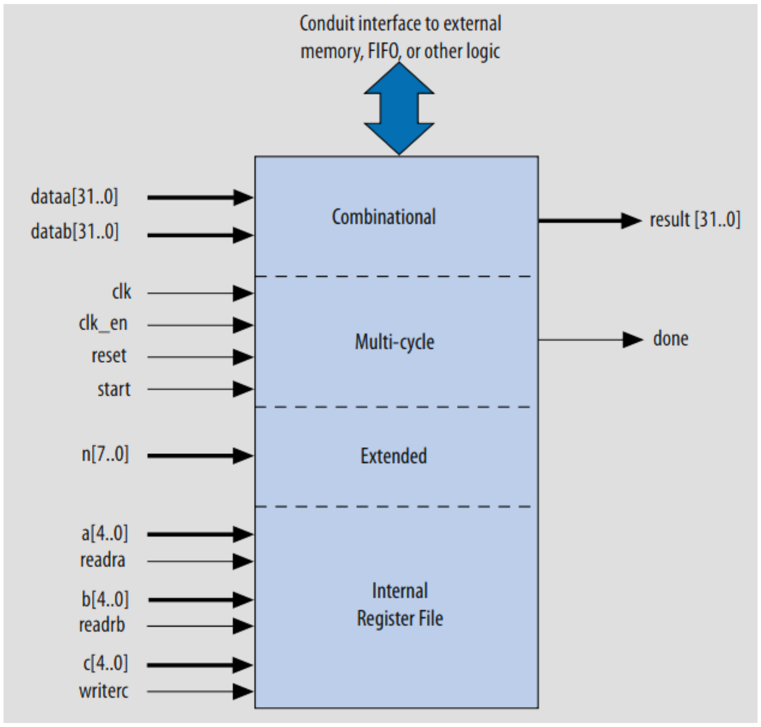


Figure 4.2: Nios II Custom Instruction Types [19]

4.3 ANN Accelerator Custom Instruction

The designed ANN Accelerator custom instruction utilizes the variable, multi-cycle format with a 2-bit extended `n` field. The Nios II requires this type of custom instruction to abide by the timing diagram show in Figure 4.3. The input operands ports, `dataa`, `datab`, and `n`, are valid from the rising edge of the `start` signal, until the module asserts the `done` signal. The `result` port is read on the rising edge when `done` is asserted. Then, the instruction is complete. In the design, the 2-bit extended field, `n`, serves as an opcode extension for the custom instruction with the mapping outlined in Table 4.1. The `n` field decides if the instruction will perform the CLEAR, WRITE, READ, or WAIT functionality.

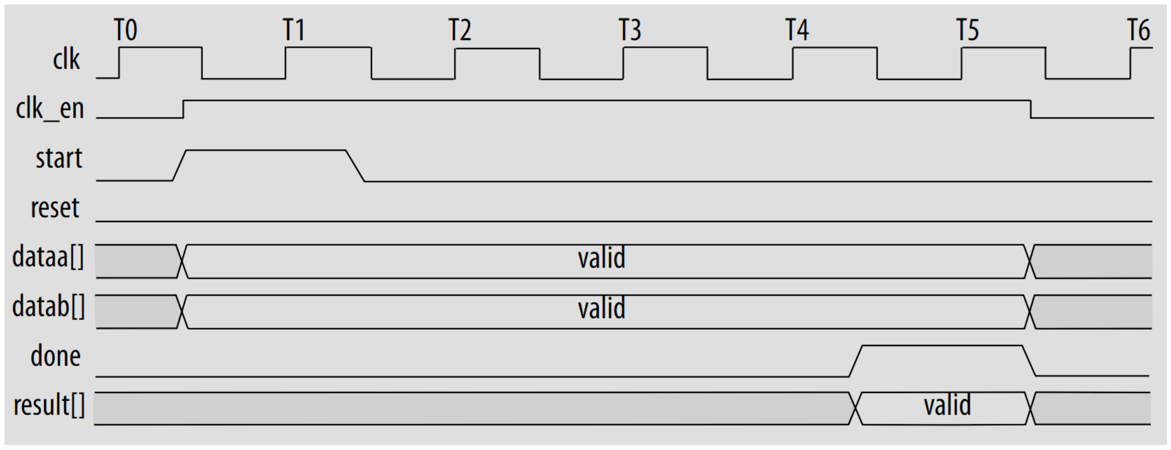


Figure 4.3: Variable Multi-Cycle CI Timing Diagram [19]

Table 4.1: Extended Field Functions

`n[1:0]`	Function	`dataa`	`datab`	`result`	clks
00	CLEAR	N/A	N/A	N/A	1
01	WRITE	ROM address	Input data	N/A	2
10	READ	Channel address	ReLU/SHIFT code	Channel output	1
11	WAIT	N/A	N/A	N/A	3

The designed custom instruction component has multiple parameterization options. The `CH_AW` specifies the address width of each of the `mac4.sv` ROMs. The `NUM_CH` parameter specifies the number of `mac4.sv` multiply-accumulate modules to be instantiated. It is recommended to set this value to the number of nodes in the largest layer of the MLP. If the device resources cannot accommodate that number of multiply-accumulate units, layer weights can be concatenated into the `mac4.sv` ROMs, but then would require multiple passes of the custom instruction invocation, reducing performance. The `NUM_CH` parameter must be less than or equal to $2^{\texttt{ROM\_AW}}$.

A testbench was written to verify the functionality of the module in the *ModelSim - Intel FPGA Starter Edition* Software. It first tests system reset, then the CLEAR operation. It then invokes the WRITE operation twice, first with input vector $x_0 = (0, 1, 2, 3)$, then with input vector $x_1 = (4, 5, 6, 7)$. THE WAIT operation is used to wait 3 clock cycles for computation to finish. While all 128 multiply-accumulate units are performing calculations, only the first two units outputs are read using the READ operation. The multiply-accumulate unit with address 0 has weight vectors

$w_{0,0} = (-5, -1, 5, -5)$ and $w_{0,1} = (-5, -1, -4, -3)$, while the unit at address 1 has weight vectors $w_{1,0} = (1, 2, 3, 6)$ and $w_{1,1} = (4, 1, -1, -10)$. The simulation output, Figure 4.4 shows the successful computation of,

$$x_0 w_{0,0}^T + x_1 w_{0,1}^T = (-6) + (-70) = -76,$$

and,

$$x_0 w_{1,0}^T + x_1 w_{1,1}^T = 26 + (-55) = -29.$$

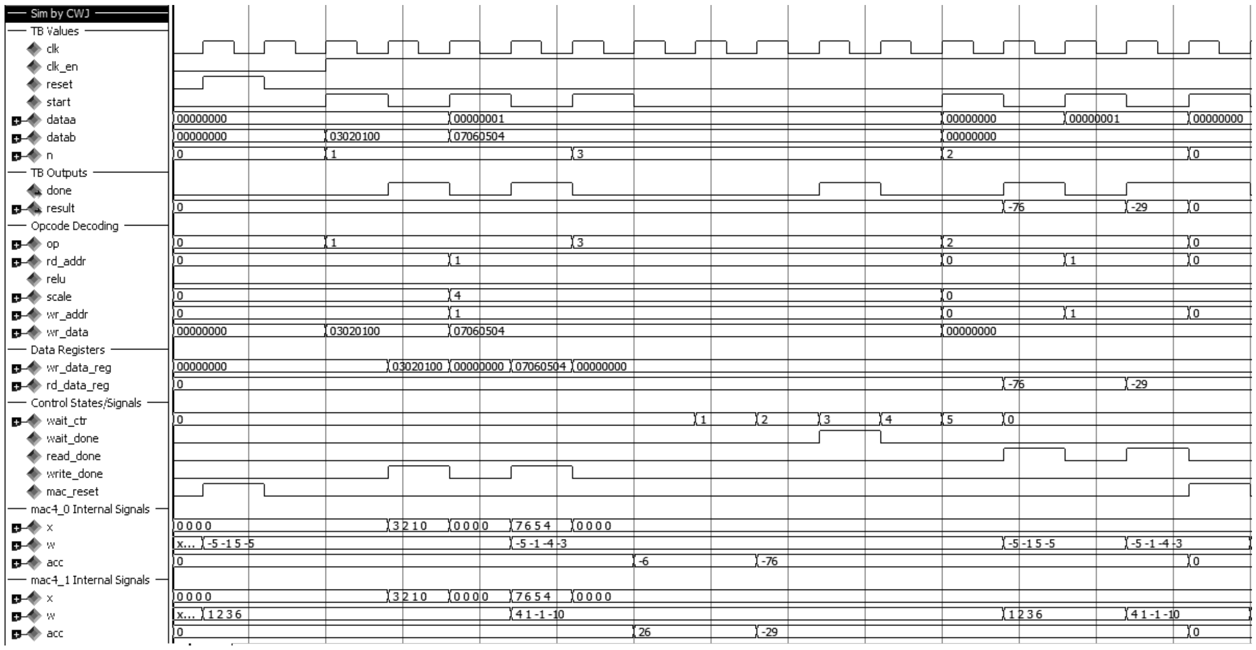


Figure 4.4: Simulation Diagram

4.4 Multiply Accumulate Unit Module

The multiply-accumulate module, `mac4.sv`, contains four 8x8-bit mixed-sign multipliers, pipeline registers, a two-level adder tree, an accumulator register, and a ROM file. It receives a data block and an address from the control unit. The 32-bit data block encodes four 8-bit unsigned integer multiplicands, and the 32-bit word present at the address in ROM encodes the four corresponding signed integer multipliers. Each of the four products are summed together through the pipelined adder tree, then the resultant sum is added to the accumulator register. The `mac4.sv` module can be prefaced by `(* multstyle = "logic" *)` or the `(* multstyle = "dsp" *)` synthesis attribute, to ensure the multipliers inferred use logic elements or DSP blocks, respectively. If a synthesis attribute is not used, the type of multiplier inferred is automatically chosen by the synthesis tool.

The module is parameterized to allow for changes in ROM size and initialization values. The `ROM_AW` parameter specifies the address width of ROM, while the `MEM_INIT_FILE` specifies the path to the hexadecimal formatted file of initialization values. Since the ROM module has a fixed 32-bit data width, it can hold $2^{\texttt{ROM\_AW}}$ data blocks, or $2^{\texttt{ROM\_AW}+2}$ 8-bit integers.

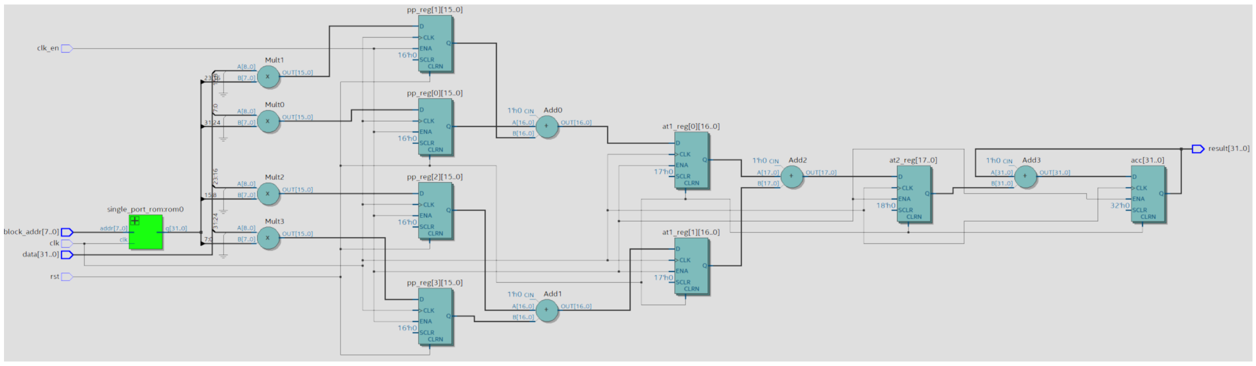
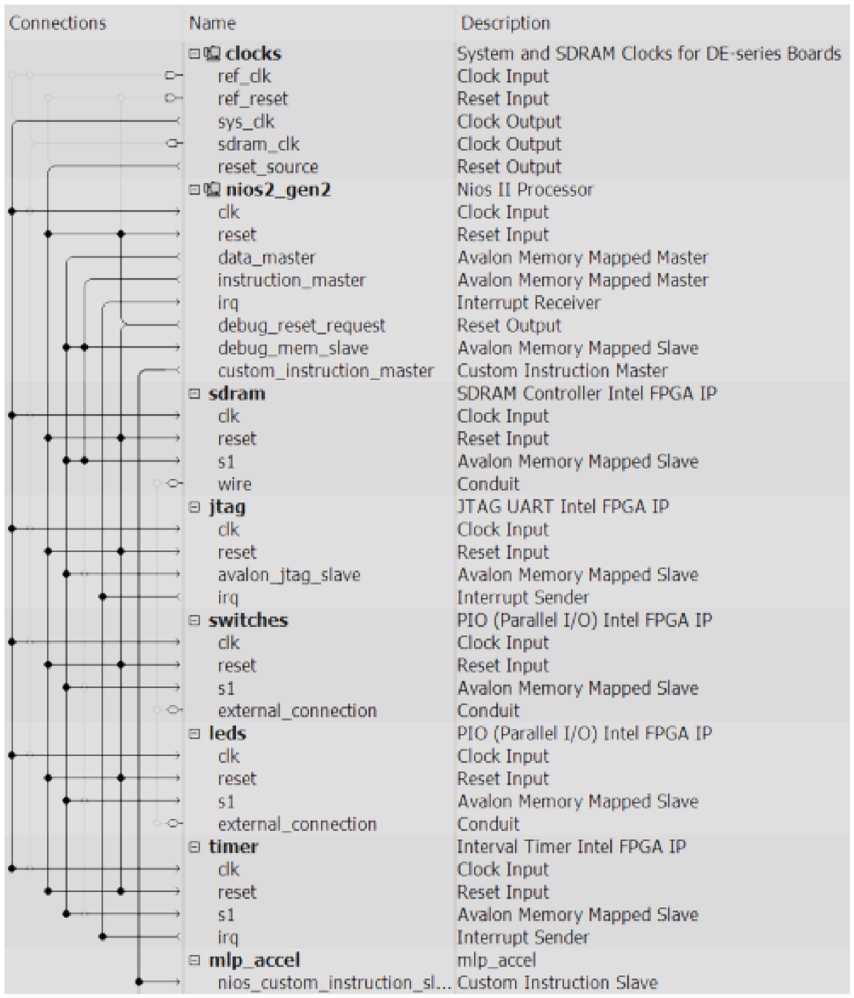


Figure 4.5: Multiply Accumulate Module

4.5 Top-Level Design

The base Nios II system was developed using the "bottom up" approach described in [20], using the same IP block parameters and address map described. The Platform Designer tool has an intuitive GUI that lets one instantiate and connect IP blocks, and it also doubles as a convenient top-level design of the system, shown in Figure 4.6. Four versions of the base Nios II system were created: "/e", "/e with CI", "/f", and "/f with CI". These share the same peripherals, address map (see Table 4.2), and clock frequency, and only differ in their Nios II core variant and inclusion of the custom instruction. The /f core used in the system designs was configured with caches and hardware multiply, shift, and rotate instructions (see Table 4.3), while the /e core retains it's default configuration. To complete the system, the "System and SDRAM Clocks for DE-series Boards", "SDRAM Controller", "JTAG UART", "PIO (Parallel I/O)", and "Interval Timer" Intel FPGA IP blocks were used.



Connections	Name	Description
	clocks	System and SDRAM Clocks for DE-series Boards
	ref_clk	Clock Input
	ref_reset	Reset Input
	sys_clk	Clock Output
	sdram_clk	Clock Output
	reset_source	Reset Output
	nios2_gen2	Nios II Processor
	clk	Clock Input
	reset	Reset Input
	data_master	Avalon Memory Mapped Master
	instruction_master	Avalon Memory Mapped Master
	irq	Interrupt Receiver
	debug_reset_request	Reset Output
	debug_mem_slave	Avalon Memory Mapped Slave
	custom_instruction_master	Custom Instruction Master
	sdram	SDRAM Controller Intel FPGA IP
	clk	Clock Input
	reset	Reset Input
	s1	Avalon Memory Mapped Slave
	wire	Conduit
	jtag	JTAG UART Intel FPGA IP
	clk	Clock Input
	reset	Reset Input
	avalon_jtag_slave	Avalon Memory Mapped Slave
	irq	Interrupt Sender
	switches	PIO (Parallel I/O) Intel FPGA IP
	clk	Clock Input
	reset	Reset Input
	s1	Avalon Memory Mapped Slave
	external_connection	Conduit
	leds	PIO (Parallel I/O) Intel FPGA IP
	clk	Clock Input
	reset	Reset Input
	s1	Avalon Memory Mapped Slave
	external_connection	Conduit
	timer	Interval Timer Intel FPGA IP
	clk	Clock Input
	reset	Reset Input
	s1	Avalon Memory Mapped Slave
	irq	Interrupt Sender
	mlp_accel	mlp_accel
	nios_custom_instruction_sl...	Custom Instruction Slave

Figure 4.6: Nios II Base System

Table 4.2: Nios II System Address Map

Component	Base	End	Interrupt No
SDRAM	0x0000_0000	0x03FF_FFFF	N/A
JTAG-UART	0xFF20_1000	0xFF20_1007	5
Switches	0xFF20_0040	0xFF20_004F	N/A
LEDs	0xFF20_0000	0xFF20_000F	N/A
Timer	0xFF20_2000	0xFF20_201F	0

Table 4.3: Nios II/f Core Parameters

Parameter	Value
Instruction Cache Size	64 Kbytes
Data Cache Size	64 Kbytes
Multiply Implementation	3 16-bit Multipliers (1 cycle)
Multiply Extended Implementation	Software
Divide Hardware	None
Shift/Rotate Implementation	Logic Elements (Pipelined, 1 cycle)

CHAPTER 5

SOFTWARE DESIGN

Implementing a artificial neural network accelerator in hardware requires a surprising amount of supporting software. First, a neural network must be defined and trained. Here, it is important to constrain the architecture of the neural network to the limits of the hardware design. The hardware accelerator only supports fully-connected layers, so other types, such as convolutional layers, must not be present. If one wants to use other types of layers, they must be implemented manually with a software routine. FPGA's have a finite number of embedded multipliers, logic elements, and block memory bits that must be considered. For this, we limit the size (number of nodes) of each layer. Additionally, we limit the total number of nodes in the design. Because of the lower speed and higher area of floating-point operations relative to fixed-point/integer operations on an FPGA, the models weights must be quantized to an 8-bit integer format to achieve high performance.

5.1 Overview

The software side of the project uses a modular approach, to better handle a variety of different work-flows and alterations. The programs, and their corresponding I/O are described abstractly in Table 5.2. A supporting python library developed, `mlp.py`, defines the MLP class, and can be parameterized by the number of hidden layers, along with the size of hidden layers and the size of the output layer. It contains a class definition and supporting functions. While the software project itself is fairly complex, a script, `start.sh` is provided to abstract away much complexity from the user. The script guides the user to install dependencies, choose options, build the project, execute programs, and ultimately produce the files needed for embedded hardware and software implementation. The execution and data flow of the project is illustrated in Figure 5.1, where a model is trained, quantized, tested, and transformed into output files.

To build the project, it is necessary to install the software dependencies listed in Table 5.1. It is recommended to install the latest version of each package. While the `start.sh` script includes an option to install and update many of these dependencies, a user should first attempt to install them manually.

Table 5.1: Software Dependencies

Dependencies	Use
APT	Package Manager
Bash	Scripting Shell Language
GNU Make	Build System
Python3	Python Interpreter
pip3	Python Package Manager
numpy	Numerical Computing Library
torch	Main PyTorch Functionality
torchvision	Datasets, Dataloaders, Data Transforms
GCC or Clang	C Compiler

Table 5.2: Software Programs

Program	Inputs	Outputs
`start.sh`	User Input	Executes Software Flow
`config.py`	User Input	Software Project Settings Text File
`train.py`	Model, Training Options	Trained Model
`quantize.py`	Model	Quantized Model
`save_weights.py`	Quantized Model	Binary Layer Weight Files
`gen_files.exe`	Binary Layer Weight Files	ROM and C-Array Initialization Files
`test_pt.py`	Model, Dataset Name, Device	Accuracy, Execution Time
`test_c.exe`	Binary Layer Weight Binaries	Accuracy, Execution Time

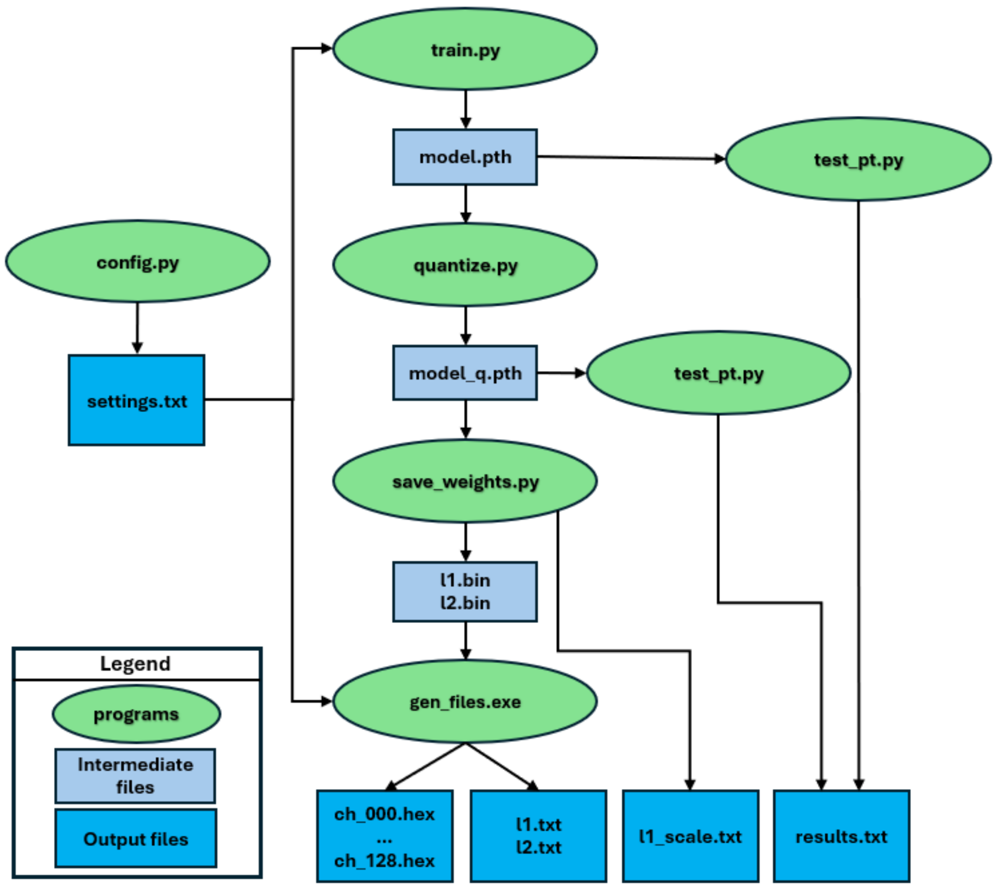


Figure 5.1: Software Project Data Flow

5.2 Software Development Tools

All software development and testing were done using the *Windows Subsystem for Linux 2* (WSL2). This allows for easy cross-platform development and deployment on Microsoft Windows, Linux/Unix, and Apple MacOS based machines. WSL2 is unique to other Linux emulators and Virtual Machines (VM) in that it seamlessly allows access to peripheral devices such as Graphics Processing Units (GPUs). Programs were written with the `Bash` scripting language, `Python` high-level objected-oriented language, and `C` low-level imperative language. Below shows the different tools needed to build and execute the software programs for each tested operating system. It is important to note that the hardware development tools, to be discussed in the next chapter, use Windows 11 without WSL2.

Table 5.3: Software Development Tools

OS Type	OS Distribution	Build System	C Compiler	Package Manager
Windows (WSL2)	Ubuntu 22.04	GNU Make	GNU GCC	APT
Linux	Ubuntu 22.04	GNU Make	GNU GCC	APT
Apple MacOS	Ventura	GNU Make	Apple Clang	Homebrew

5.3 PyTorch Library

The open-source `PyTorch` library with the high-level, interpretive `Python` language, was used to create, train and quantize the neural network models. While `PyTorch`'s interface is in `Python`, most of it is written in the `C++ libtorch` library to increase performance. It accomplishes this increase in performance by utilizing C++'s efficient parallelism mechanisms, while avoiding `Python`'s high overhead and global interpretive lock [21]. `PyTorch`'s backend is optimized for CPUs (Central Processing Unit), NVIDIA GPUs, and Apple Silicon GPUs.

Aggregate data is stored in `torch.tensor` objects [7]. These can include vectors, matrices, or higher order tensors [7]. `PyTorch` includes a variety of operations for tensors, from linear algebra operations to image processing routines. It supports automatic differentiation, which makes it much easier for programmers to implement gradient based optimization. [21]

5.4 Training

The `train.py` program utilizes `PyTorch` library functionality to train a model. It loads the MNIST dataset using `PyTorch`'s `torchvision` interface. The program detects if a system has a CUDA device (NVIDIA GPU) or MPS device (Apple Silicon GPU), and automatically uses the device to accelerate training. If no acceleration device is detected, it defaults to using the CPU. It supports a variety of optimization algorithms, but by default uses Stochastic Gradient Descent via `PyTorch`'s `torch.optim.SGD` optimizer. It supports an optional "pre-training" of the hidden layers with the larger EMNIST dataset, for a specified number of epochs. This pre-training aims to help the hidden layer better extract patterns of features from the image. When moving on to regular training, or "fine tuning", the output layer is replaced with a randomly initialized weight matrix, with ten rows for the each class in the MNIST database.

5.5 Quantization

The `quantize.py` program utilizes `PyTorch`'s FX Graph Mode Quantization workflow. It uses the "Post-Training Static Quantization" method. This is the most suited for use on embedded hardware, as other methods, like "Post-Training Dynamic Quantization", will adapt the quantization schemes for activations in real time for each input. The program uses the "per-tensor-affine" quantization scheme, which uses a single scaling factor for each layer weight matrix. The scaling factor was rounded up to the nearest power of two, to allow scaling to be performed with simple right bit shifts in hardware. Weight matrices were quantized to 8 bit signed integers, with each weight matrix assigned a floating point scale. The scaling factor was rounded up to the nearest power of 2, so scaling could be performed with simple right shifts in hardware and embedded software.

5.6 Testing

After training (and possibly quantizing) our model, it is important to test it's performance. To do this, we use the test portion of our dataset. The test routine iterates through each (image,label) pair in the test dataset, counting the number of correct and incorrect predictions, and calculating the accuracy. The `test_pt.py` program uses `PyTorch` functionality to perform the tests. `test_pt.py` can test both unquantized and quantized models, which aids in analysis in the reduction of accuracy induced by quantization. It features a command-line argument to pick a GPU (if available) or CPU for testing. On the other hand, `test_c.exe` uses the layer weight and dataset binaries to test the design. This testing program, written in the `C` language, is helpful for finding errors in the parameter extraction process. Additionally, it provides a blueprint for the implementation of the embedded testing program.

5.7 Parameter Extraction

After the model is trained, quantized, and tested, the layer weights must be converted into a form that can be used to initialize ROM's in hardware, and also initialize arrays in embedded software. The program `gen_files.exe` creates ROM files by grouping the 8-bit layer weights into groups of four, and writing their hexadecimal representations, followed by a newline character to a `.hex` file. If a layer has a number of weights not divisible by 4, the rest of the line is padded with zeros. Subsequent layer weights are concatenated to the same file in the same manner. The program generates array initialization files by simply writing the decimal representation of the layer weights to a text file, delimited by commas. The program generates an array initialization file for each layer by simply printing out a `.txt` file of comma-separated values of the elements of the layer's weight matrix in row-major order.

5.8 Embedded Software Design

The *Intel FPGA Monitor Program* IDE was used to upload the hardware design to the FPGA, fill memory locations with test data, and debug the design. It supports ANSI C programs, as well as C programs with an automatically generated BSP (Board Support Package). The IDE supports breakpoints, hardware debugging, register and memory inspection, console I/O, and disassembly viewing [22]. ANSI C projects were chosen to reduce executable size and potentially increase performance.

The embedded software uses a few techniques to aid development and testing. The preprocessor macro `#include` is used to place initialization values for large global arrays from a text file into the source file. Alternatively, an uninitialized array can be declared as `volatile`, with data loaded using the *Intel FPGA Monitor Program*'s "Load File to Memory" function.

Consideration of memory layout and cache was taken when writing the matrix-vector software routines. Since the layer weight matrices are stored in row-major order, the inner loop iterates over columns, ensuring stride-1 memory accesses, which reduces the frequency of cache misses in Nios II/f systems with data cache enabled.

The custom instruction matrix-vector routine iterates through input values with a stride of 4. It casts the address of the current element of the unsigned 8-bit integer array to a 32-bit integer pointer, then dereferences the pointer before passing it as an argument to the custom instruction. This allows the custom instruction to process four input values in a single clock cycle per function call. An integer representing the current block of four inputs is passed as the other argument, which serves as an address for the custom instruction to fetch the corresponding weights from each multiply-accumulate module's ROM. For subsequent layers, an offset is applied to the address to fetch the correct weights.

CHAPTER 6

EXPERIMENTAL DESIGN

First, a `PyTorch` model of the `MLP` class, defined in `mlp.py`, was created with the parameters in Table 6.1. The model uses a single hidden layer with 100 nodes. Layer biases were disabled, as they are often quantized to zero, and training without biases decreased the loss in accuracy from quantization. Then, it was trained using the **`train.py`** script with the options specified in Table 6.2. The “Alpha” parameter specifies the learning rate of the SGD algorithm, which is decayed every epoch by the “Gamma” parameter. The optimizer utilizes *momentum*, which is an additional parameter of the SGD optimization algorithm, that can accelerate convergence [23]. The `PyTorch` model was saved as `model.pth` for testing and quantization.

Table 6.1: MLP Model Architecture

Parameter	Value
Number of Hidden Layers:	1
Number of Input Features	784
Nodes (Hidden Layer 1):	100
Activation Function (Hidden Layer 1)	ReLU
Nodes (Output Layer):	10
Layer Bias	False
Parameter Data type:	float32

Table 6.2: MLP Training Parameters

Parameter	Value
Device	CUDA
Optimizer:	SGD
Training Batch Size	64
EMNIST Pre-Training	True
Alpha:	0.02
Gamma:	0.99
Momentum	0.9
Epochs:	10

Using test_pt.py (see Chapter 5), the unquantized trained model achieved 96.74% accuracy on the 10,000 image test set of the MNIST database. The floating-point model was then quantized with quantize.py, producing a quantized model with 8-bit signed integer layer weights, which was saved as model_q.pth. The quantized model, model_q.pth, achieved nearly the same accuracy as the unquantized model, 96.65%. The test results were saved to a text file, results.txt.

Then, the layer weights of the quantized model, model_q.pth, were converted into row-major binaries, l1.bin and l2.bin using the script save_weights.py. The ROM and array initialization files were generated from the binaries with gen_files.exe.

The DE-10 Standard development board was used for testing the hardware design. In the Platform Designer Tool, the mlp_accel custom instruction was configured with parameters ROM_AW = 8, CH_AW = 7, and NUM_CH = 100. The ROM_FILE_PATH parameter was set to the directory of the generated ROM files. This was repeated for the Nios II "/e with CI" and "/f with CI" systems. The system clock frequency was set to 120 MHz for all systems. The hardware designs were compiled with Quartus Prime 18.1 Standard for the Cyclone V 5CSXFC6D6F31C6 device, and the compilation reports, including compilation time, were recorded. For each system, the compilation mode was set to "Performance (Aggressive - increases runtime and area)," with all other synthesis and fitter options holding their default settings.

Execution time for the Nios II implementations were measured with the "Interval Timer Intel FPGA IP" core, and corresponding embedded software routine.

CHAPTER 7

RESULTS

After successful compilation of each of the Nios II systems, the device resources and timing analysis reports were recorded. The "Slow 85C" Fmax measurements are provided by the timing analyzer as estimates of the systems maximum clock frequency. As shown in Table 7.1, the systems with the custom instruction implemented received relatively little Fmax performance degradation, due to the pipelined implementation of the custom instruction. Because of the insufficient number of embedded multipliers in the Cyclone V device, some of the multipliers for the custom instruction were implemented with logic elements. This explains the higher number of ALMs, and all of the devices 112 DSP blocks utilized for the systems with the custom instruction. The "f" systems utilize three DSP blocks for their hardware multiply instruction.

Table 7.1: Nios II System Resource Utilization

Nios II System	ALMs	Regs	M10K Bits	DSPs	Fmax Slow 85C	Compilation Time
/e	1,207	1,805	11,392	0	134.70 MHz	2min, 4s
/e with CI	11,215	18,360	830,592	112	128.21 MHz	7min, 48s
/f	2,014	2,922	1,136,512	3	136.87 MHz	3min, 0s
/f with CI	12,242	19,706	1,955,712	112	125.58 MHz	9min, 32s

For both Nios II/e and Nios II/f systems, speedups of over 308x were achieved, providing orders of magnitude quicker inference performance. Nios II/e systems achieved a 359x speedup (see Table 7.2), and the Nios/f systems acheived 359x (see Table 7.3). The slowest system, "/e", can only process 6 images per second, while the fastest, "/f with CI", can process over 28,000 images a second! For reference, typical video formats range from just 24-60 images per second. The output vectors, $\hat{y}$, were identical for each system, so there was no loss of accuracy when using the custom instruction.

Table 7.2: Nios II/e System Single Inference Performance

Nios II System	Clock Cycles	Fmax	Inference Time	Relative Speedup
/e	22,758,740	134.70 MHz	168,958.72 μs	1.00
/e with CI	60,339	128.21 MHz	470.63 μs	359.01

Table 7.3: Nios II/f System Single Inference Performance

Nios II System	Clock Cycles	Fmax	Inference Time	Relative Speedup
/f	1,489,062	136.87 MHz	10,879.38 μs	1.00
/f with CI	4,430	125.58 MHz	35.27 μs	308.40

CHAPTER 8

CONCLUSION

The custom instruction accelerator showed groundbreaking speedups for performing image classification with Artificial Neural Networks (ANNs) on FPGAs. The extremely low latencies achieved allow the use of FPGAs for real-time inference in embedded or data-center applications, as the systems were shown to be able to perform orders of magnitude more inferences per second than typical video frame-rates.

A Multi-Layer Perceptron (MLP) model was created using the PyTorch framework and trained from scratch to perform image classification on the MNIST handwritten character database. The model achieved an impressive 96.74% accuracy on the test set, and lost only a negligible amount of accuracy after being quantized to an 8-bit integer format. A parameterized Nios II embedded soft processor custom instruction was developed that accelerates matrix-vector multiplications necessary for computation of MLPs. Using a system generation software framework, the custom instruction was configured to accelerate the trained, quantized model. Nios II systems with custom instructions were deployed on a Cyclone V FPGA, and over 347x speedups were recorded, with inferences performed in as little as 36 μs.

The reconfigurable hardware design and system accelerator software allow this work to be adapted to a variety of different applications, MLP architectures, and FPGA devices with minimal effort or knowledge from a user.

This work will be tested with MLPs with more, and bigger layers, and on more complex problems, like the EMNIST dataset. Future work can be done to support more ANN architectures, such as Convolutional Neural Networks (CNNs), Recurrent Neural Networks (RNNs), and Deep Neural Networks (DNNs). Additionally, the work can be adapted to create stand-alone accelerator components that can interface with SoC devices with Hard Processor Systems (HPS). It can also be integrated into workstation/server applications with PCIe based FPGA accelerator cards, potentially replacing GPUs as inference accelerators for workstations and servers. ANN accelerator custom instructions could be generated for the newer, RISCV ISA, Nios V soft processor.

This work will be released as an open-source project on the author's *github* account, *https://github.com/cw-johnson*, so that others can deploy, study, or build off of the project.

APPENDIX A

ANN ACCELERATOR MODULE

```
// Author: Christopher Johnson
// Date: 03/19/24
// Revision 1.4
// Filename: lin_layer_8bit.sv
// Description: ANN Accelerator Module

module lin_layer_8bit
#(parameter ROM_AW = 8, parameter NUM_CH = 128, parameter CH_AW = 7, parameter
    ROM_FILE_PATH = "./comb_rom_data/")
(
   input [31:0] dataa,
   input [31:0] datab,
   output reg [31:0] result,
   input clk,
   input clk_en,
   input reset,
   input start,
   output reg done,
   input [1:0] n
);

   // Opcode Parameters
   localparam CLEAR = 2'd0;
   localparam WRITE = 2'd1;
   localparam READ = 2'd2;
   localparam WAIT = 2'd3;

   localparam WAIT_CYCLES = 3'd3;

   // Opcode Decoding
   wire [1:0] op = n[1:0];

   // Operand Decoding
   // WRITE
   wire [ROM_AW-1:0]  wr_addr = dataa[ROM_AW-1:0];
   wire [31:0]     wr_data = datab[31:0];
   // READ
   wire [CH_AW-1:0]  rd_addr = dataa[CH_AW-1:0];
   wire       relu = datab[4];
   wire [3:0]    scale  = datab[3:0];


   // Connection Wires
   wire [31:0] ch_outputs[NUM_CH-1:0];
```

```
// MAC Unit Reset
reg mac_reset;

// Registers
reg [31:0] wr_data_reg;
reg signed [31:0] rd_data_reg;
reg read_done;
wire wait_done;
reg [3:0] wait_ctr;
reg write_done;

// Initialize Registers
initial begin
   wr_data_reg <= '0;
   rd_data_reg <= '0;
   read_done <= '0;
   wait_ctr <= '0;
   write_done <= '0;
end

// Instantiate MAC Units
genvar i;
generate
   for (i = 0; i < NUM_CH; i++) begin : lp_gen_macs
      localparam [7:0] index_ones = "0" + (i % 10);
      localparam [7:0] index_tens = "0" + ((i/10) % 10);
      localparam [7:0] index_huns = "0" + ((i/100) % 10);
      mac4 #(.ROM_AW(ROM_AW), .MEM_INIT_FILE({ROM_FILE_PATH, "ch_", index_huns,
          index_tens, index_ones, ".hex"})) m0
         (
         .clk(clk),
         .clk_en(clk_en),
         .rst(mac_reset),
         .data(wr_data_reg),
         .block_addr(wr_addr),
         .result(ch_outputs[i])
         );
   end
endgenerate

// "result" Multiplexer
always_comb begin
   if (op == READ) begin
      if ((relu) && (rd_data_reg[31] == 1'b1)) begin
         result <= '0;
      end else begin
         result <= rd_data_reg >>> scale;
      end
   end else begin
      result <= '0;
   end
end
```

```
// "wait_done" multiplexer
assign wait_done = (wait_ctr == WAIT_CYCLES) ? 1'b1 : 1'b0;

// "done" Multiplexer
always_comb begin
   if (op == READ) begin
      done <= read_done;
   end else if (op == WAIT) begin
      done <= wait_done;
   end else if (op == WRITE) begin
      done <= write_done;
   end else begin
      done <= start;
   end
end

// "mac_reset" Multiplexer
always_comb begin
   if (reset) begin
      mac_reset <= reset;
   end else if ((clk_en) && (start) && (op==CLEAR)) begin
      mac_reset <= start;
   end else begin
      mac_reset <= 1'b0;
   end
end

// WRITE Registers behavior
always @ (posedge clk or posedge reset) begin
   if (reset) begin
      wr_data_reg <= '0;
      write_done <= 1'b0;
   end else if (clk_en) begin
      if ((start) && (op==WRITE)) begin
         wr_data_reg <= wr_data;
         write_done <= 1'b1;
      end else begin
         wr_data_reg <= '0;
         write_done <= 1'b0;
      end
   end
end

// READ Registers behavior
always @ (posedge clk or posedge reset) begin
   if (reset) begin
      rd_data_reg <= '0;
      read_done <= 1'b0;
   end else if (clk_en) begin
      if ((start) && (op==READ)) begin
         if (rd_addr < NUM_CH) begin
            rd_data_reg <= ch_outputs[rd_addr];
            end else begin
               rd_data_reg <= '0;
            end
            read_done <= 1'b1;
         end else begin
            read_done <= 1'b0;
         end
      end
   end

   // WAIT Registers Behavior
   always @ (posedge clk or posedge reset) begin
      if (reset) begin
         wait_ctr <= '0;
      end else if (clk_en) begin
         if ((start) && (op==WAIT)) begin
            wait_ctr <= '0;
         end else if (op==WAIT) begin
            if (wait_ctr < 3'd7) begin
               wait_ctr <= wait_ctr + 1'b1;
            end
         end else begin
            wait_ctr <= '0;
         end
      end
   end

endmodule
```

APPENDIX B

MULTIPLY ACCUMULATE UNIT MODULE

```
// Author: Christopher Johnson
// Date: 03/19/24
// Revision 1.2
// Filename: mac4.sv
// Description: Single Pipelined Multiply Accumulate Unit with ROM
//(* multstyle = "logic" *)
//(* multstyle = "dsp" *)
module mac4
#(parameter ROM_AW = 8, parameter MEM_INIT_FILE = "./comb_rom_data/ch_000.hex")
(
   input clk,
   input clk_en,
   input rst,
   input [31:0] data,
   input [ROM_AW-1:0] block_addr,
   output [31:0] result
);

   //Wire Declarations
   wire [31:0] rom_data;
   wire signed [8:0] x[3:0];
   wire signed [7:0] w[3:0];
   wire signed [15:0] pp[3:0];
   wire signed [16:0] at1[1:0];
   wire signed [17:0] at2;

   //Register Declarations
   reg signed [15:0] pp_reg[3:0];
   reg signed [16:0] at1_reg[1:0];
   reg signed [17:0] at2_reg;
   reg signed [31:0] acc;

   //Operand Decoding and Multiplications
   genvar i;
   generate
      for (i=0;i<4;i++) begin : lpp
         assign x[i] = {1'b0,data[8*i+7 : 8*i]};
         assign w[i] = rom_data[8*i+7 : 8*i];
         assign pp[i] = x[i] * w[3-i];
      end
   endgenerate

   //Adder Tree
   assign at1[0] = pp_reg[0] + pp_reg[1];
   assign at1[1] = pp_reg[2] + pp_reg[3];
   assign at2 = at1_reg[0] + at1_reg[1];
   assign result = acc;

   initial begin
         pp_reg <= '{default : '0};
         at1_reg <= '{default : '0};
         at2_reg <= '0;
         acc <= '0;
   end

   // Pipeline Register Behavior
   always @ (posedge clk, posedge rst) begin
      if (rst) begin //Clear All Pipeline Registers
         pp_reg <= '{default : '0};
         at1_reg <= '{default : '0};
         at2_reg <= '0;
         acc <= '0;
      end else if (clk_en) begin
         pp_reg <= pp;
         at1_reg <= at1;
         at2_reg <= at2;
         acc <= acc + at2_reg;
      end
   end

   //Instantiate ROM
   single_port_rom #(.DATA_WIDTH(32), .ADDR_WIDTH(ROM_AW),
       .MEM_INIT_FILE(MEM_INIT_FILE)) rom0 (block_addr, clk, rom_data);

endmodule


// Quartus Prime Verilog Template
// Single Port ROM
module single_port_rom
#(parameter DATA_WIDTH=32, parameter ADDR_WIDTH=8, parameter
    MEM_INIT_FILE="./comb_rom_data/ch_000.hex")
(
   input [(ADDR_WIDTH-1):0] addr,
   input clk,
   output reg [(DATA_WIDTH-1):0] q
);

   // Declare the ROM variable
   reg [DATA_WIDTH-1:0] rom[2**ADDR_WIDTH-1:0];

   // Initialize the ROM with $readmemb
   initial begin
      $readmemh(MEM_INIT_FILE,rom);
   end

   always @ (posedge clk) begin
```

```
      q <= rom[addr];
   end
endmodule
```

REFERENCES

[1] OpenAI, : J. Achiam, *et al.*, *GPT-4 technical report*, 2023. arXiv: 2303.08774 [cs.CL].

[2] *Introduction to the Intel Nios II soft processor*, ver 18.1, Intel, Santa Clara, California, 2019. [Online]. Available: https://ftp.intel.com/Public/Pub/fpgaup/pub/Teaching_Materials/current/Tutorials/Nios2_introduction.pdf.

[3] L. Deng, *The MNIST database of handwritten digit images for machine learning research*, 2012. DOI: 10.1109/MSP.2012.2211477.

[4] S. Haykin, *Neural Networks: A Comprehensive Foundation (3rd Edition)*. USA: Prentice-Hall, Inc., 2007, ISBN: 0131471392.

[5] W. S. McCulloch and W. Pitts, "A logical calculus of the ideas immanent in nervous activity," in *Neurocomputing: Foundations of Research*. Cambridge, MA, USA: MIT Press, 1988, pp. 15–27, ISBN: 0262010976.

[6] F. Rosenblatt, "The perceptron - a perceiving and recognizing automaton," Cornell Aeronautical Laboratory, Ithaca, New York, Tech. Rep. 85-460-1, Jan. 1957.

[7] A. Zhang, Z. C. Lipton, M. Li, and A. J. Smola, *Dive into deep learning*, 2023. arXiv: 2106.11342 [cs.LG].

[8] A. Quarteroni, R. Sacco, and F. Saleri, *Numerical Mathematics (Texts in Applied Mathematics)*, 2nd ed. Springer Berlin, Heidelberg, 2006, ISBN: 978-3-540-34658-6.

[9] L. Bottou, C. Cortes, J. Denker, *et al.*, "Comparison of classifier methods: A case study in handwritten digit recognition," in *Proceedings of the 12th IAPR International Conference on Pattern Recognition, Vol. 3 - Conference C: Signal Processing (Cat. No.94CH3440-5)*, vol. 2, 1994, 77–82 vol.2. DOI: 10.1109/ICPR.1994.576879.

[10] G. Cohen, S. Afshar, J. Tapson, and A. van Schaik, *EMNIST: An extension of MNIST to handwritten letters*, 2017. arXiv: 1702.05373 [cs.CV].

[11] M. Nagel, M. Fournarakis, R. A. Amjad, Y. Bondarenko, M. van Baalen, and T. Blankevoort, "A white paper on neural network quantization," *CoRR*, vol. abs/2106.08295, 2021. arXiv: 2106.08295. [Online]. Available: https://arxiv.org/abs/2106.08295.

[12] *Cyclone V device handbook, volume 1: Devices interfaces and integration*, ver 2023.10.18, Intel/Altera, San Jose, California, 2023. [Online]. Available: https://cdrdv2.intel.com/v1/dl/getContent/666995?fileName=cv_5v2-683375-666995.pdf.

[13] *Cyclone V hard processor system technical reference manual*, ver 2023.08.28, Intel/Altera, San Jose, California, 2023. [Online]. Available: `https://www.intel.com/content/www/us/en/docs/programmable/683126/21-2/hard-processor-system-technical-reference.html`.

[14] *Embedded memory user guide*, ver 2021.09.17, Intel, Santa Clara, California, 2021. [Online]. Available: `https://cdrdv2.intel.com/v1/dl/getContent/667041?fileName=ug_ram_rom-683240-667041.pdf`.

[15] *DE10-standard user manual*, ver 1.06, Terasic, HsinChu, Taiwan, 2018. [Online]. Available: `https://www.terasic.com.tw/cgi-bin/page/archive.pl?Language=English&CategoryNo=165&No=1081&PartNo=4#contents`.

[16] *Introduction to the platform designer tool*, ver 18.1, Intel, Santa Clara, California, 2019. [Online]. Available: `https://ftp.intel.com/Public/Pub/fpgaup/pub/Teaching_Materials/current/Tutorials/Introduction_to_the_Qsys_Tool.pdf`.

[17] *Avalon interface specifications*, ver 18.1, Intel, Santa Clara, California, 2019. [Online]. Available: `https://www.intel.com/content/www/us/en/docs/programmable/683091/18-1/introduction.html`.

[18] *Nios II processor reference guide*, ver 2023.08.28, Intel, Santa Clara, California, 2023. [Online]. Available: `https://www.intel.com/content/www/us/en/docs/programmable/683836/current/eol.html`.

[19] *Nios II custom instruction user guide*, ver 2020.04.27, Intel, Santa Clara, California, 2020. [Online]. Available: `https://www.intel.com/content/www/us/en/docs/programmable/683242/current/nios-ii-custom-instruction-overview.html`.

[20] U. Meyer-Baese, *Embedded Microprocessor System Design using FPGAs*, 1st ed. Springer Cham, 2021, ISBN: 978-3-030-50532-5.

[21] A. Paszke, S. Gross, F. Massa, *et al.*, *Pytorch: An imperative style, high-performance deep learning library*, 2019. arXiv: `1912.01703 [cs.LG]`.

[22] *Intel FPGA monitor program tutorial for nios II*, ver 18.1, Intel, Santa Clara, California, 2019. [Online]. Available: `https://ftp.intel.com/Public/Pub/fpgaup/pub/Teaching_Materials/current/Tutorials/Intel_FPGA_Monitor_Program_NiosII.pdf`.

[23] I. Sutskever, J. Martens, G. Dahl, and G. Hinton, "On the importance of initialization and momentum in deep learning," in *Proceedings of the 30th International Conference on Machine Learning*, S. Dasgupta and D. McAllester, Eds., ser. Proceedings of Machine Learning Research, vol. 28, Atlanta, Georgia, USA: PMLR, 17–19 Jun 2013, pp. 1139–1147. [Online]. Available: `https://proceedings.mlr.press/v28/sutskever13.html`.

BIOGRAPHICAL SKETCH

Christopher Wade Johnson was born in Little Rock, Arkansas, on January 27, 1998. After graduating from high school in Naples, Florida in 2016, he briefly attended University of California, Santa Cruz. He moved back to Florida to complete his Bachelor of Science degree in Computer Engineering at Florida State University. He returned to FSU to pursue a Masters of Science in Electrical Engineering. He is passionate about digital hardware design, cryptography, high-performance computing, and artificial intelligence.