People's Democratic Republic of Algeria
Ministry of Higher Education and Scientific Research

University M'Hamed BOUGARA – Boumerdès

Institute of Electrical and Electronic Engineering
Department of Electronics

Project Report Presented in Partial Fulfilment of

the Requirements of the Degree of

'Master'

In Computer Engineering

Title:

Deep Neural Network Acceleration using Intel NIOS-II Custom Instructions

Presented By:

- **BOUDRAA Nadjib**
- **MIHOUBI Djenet Yasmine**

Supervisor:

Dr. MAACHE Ahmed

Registration Number:...../2024

Dedication

I would like to dedicate this humble work to my dear parents, for their unconditional encouragement and support and for being there since day one. I would express my gratitude to all my family members for their encouragement.

I also dedicate this project to my good friends Younes, Abdelaziz, Djilali, Souhaib, Abdelghani, and all the others for being with me in this 5 years journey. I aditionally dedicate it to my friend Iyad for being beside me for more than 10 years.

N.BOUDRAA

Dedication

To my Mom Lamouri Fadila
Your love and sacrifices have shaped my journey, guiding me with unwavering support. In every moment, your presence has been my strength. Grateful beyond words for your endless devotion.

To my Grandpa Lamouri Youcef
Your love was a father's embrace, your guidance a compass in life's storms. In your absence, memories of your kindness comfort me. Thank you for being more than a grandfather.

To my Sister Lamouri Amina
Your presence is a guiding light, illuminating my path with love and understanding. Through thick and thin, your support never wavers, You are more than a sister to me .

To my Uncle Boucheneb Hocine
You've played a significant role akin to that of a father figure in my life. I am deeply grateful for your steadfast guidance and support.

To my Uncle Alim Kamel
You have been a constant source of encouragement. Your generosity knows no bounds. Thank you for being there for me .

To my Lovelies
Yassine, Djoumana, Djawed, Rayane, Moundher, Mouatez .

To My best friend Hamadou Manel
In every laugh, every tear, and every shared moment, your presence has been a guiding light in my life. Your support and understanding have been my anchor through stormy seas. Thank you for being my rock, my confidant, and my constant companion

To My friends
Rania , Nadine , Loubna , Imene , Oumaima .
Thank you for five incredible years of friendship and support. You've made this journey unforgettable.

Mihoubi Djenet Yasmine

Acknowledgment

First and foremost, we must praise Allah, the Ever-Thankful and Most Merciful, for His help and blessings, and for granting us the ability and patience to complete this humble work.

We would like to express our sincere gratitude to everyone who contributed to the completion of this project, especially our teacher and supervisor, Dr. Ahmed Maache, for his invaluable support and guidance throughout the challenges and obstacles encountered.

We are also deeply grateful to our families, particularly our parents, for their generous and unwavering support throughout our lives and especially during our education.

Lastly, our heartfelt thanks go to all our teachers at the institute, who imparted the essential knowledge and skills needed to become good engineers and guided us to a successful conclusion of this unforgettable journey. We also appreciate all the staff members and employees at the institute.

Abstract

The rapid advancement of Artificial Intelligence (AI) has led to its integration into various fields, including embedded systems, which present unique challenges due to constraints in storage, power consumption, and the need for real-time execution. To optimize AI performance in these environments, we propose a hardware acceleration system. This system incorporates a floating point unit and lookup tables for Sigmoid and Leaky-ReLU activation functions, both designed using HDL and implemented on a Field Programmable Gate Array (FPGA) with the Nios II soft processor and its custom instructions.

We tested this system on a Deep Neural Network (DNN) written in C and trained it multiple times with varying numbers of layers. The results were remarkable, with some networks experiencing performance improvements of over 60%. However, as the model's complexity increased with additional layers, the acceleration benefit decreased, dropping to around 10% in the most complex scenarios.

Despite this reduction in acceleration for highly complex models, our system remains reliable and efficient, demonstrating its potential for critical real-time embedded AI applications.

List of Figures

List of Tables

Listings

List of Acronyms

AI Artificial Intelligence

ALU Arithmic and Logic Unit

ASIC Application-Specific Integrated Circuit

BSP Board Support Package

CPU Central Processing Unit

DNN Deep Neural Network

DSP Digital Signal Processing

DMIPS Dhrystone Million Instructions Per Second

DE10 board Development and Education board Cyclone V

FFNN Feedforward Neural Network

FPH Floating Point Hardware

FPGA Field-Programmable Gate Array

GPU Graphics Processing Unit

HDL Hardware Description Language

IoT Internet of Things

IP Intellectual Property

IEEE Institute of Electrical and Electronics Engineers

I/O Input/Output

IDE Integrated Development Environment

LUT Look-Up Table

MMU Memory Management Unit

MPU Memory Protection Unit

NN Neural Network

API	Aplication Programming Interface
PLL	Phase-Locked Loop
ReLU	Rectified Linear Unit
SoPC	System on a Programmable Chip
SDRAM	Synchronous Dynamic Random-Access Memory
TPU	Tensor Processing Unit
UART	Universal Asynchronous Receiver-Transmitter
VHDL	VHSIC Hardware Description Language (VHSIC: Very High-Speed Integrated Circuit)

Contents

Chapter 1

General Introduction

1.1 Overview

The project is fundamentally focused on the design and implementation of a dedicated neural network accelerator specifically engineered to operate within embedded systems. Embedded systems, characterized by their constrained power and space resources, pose unique challenges for deploying complex algorithms such as neural networks efficiently.

To overcome these challenges, the project adopts a unique approach: a custom hardware architecture carefully designed to enhance the execution of neural network computations within these limited environments. This architecture is customized to boost performance while using resources efficiently, ensuring effective operation within the constraints of embedded systems.

A key component of this custom hardware architecture is the utilization of the Nios II processor, a versatile and customizable embedded processor core. To further enhance its capabilities for neural network tasks, the processor is augmented with specialized custom instructions. These instructions are specifically designed to accelerate common neural network operations, enabling faster and more efficient computation.

The expected outcomes at the end of the project include a functional prototype capable of enhancing the neural network execution time while keeping a high accuracy,aiming to advance the capabilities of embedded AI on edge computing devices.

1.2 Motivation

In recent years, the field of artificial intelligence (AI) has experienced significant growth, leading to a multitude of practical applications across various domains. Among these applications, the integration of AI into embedded systems and edge devices has garnered particular attention.

Researchers have dedicated considerable effort to enhance the computational capabilities of edge devices to accelerate neural network execution. This approach is the primary focus of this project.

Utilizing FPGAs, specifically including Nios custom instructions, offers a notable advantage due to the flexibility it provides in designing and customizing hardware blocks. This flexibility enables the development of tailored solutions capable of efficiently addressing the specific challenges associated with accelerating neural networks within embedded systems.

1.3 Objectives

The project aims to achieve several key goals through the utilization of FPGA technology to enhance the speed of AI models. Firstly, it seeks to significantly reduce inference latency, enabling real-time processing of AI tasks in critical applications.

Secondly, the objective is to enhance the scalability of AI systems by offloading computational tasks from traditional processors onto specialized FPGA hardware, thus accommodating larger and more complex models without sacrificing performance.

Furthermore, optimizing resource utilization is a crucial objective, ensuring that computational resources are efficiently allocated within the FPGA architecture to maximize performance and minimize overhead. Lastly, the project aims to democratize AI by making high-performance inference accessible to a wider range of devices and applications.

1.4 Report Organization

The report is structured into four chapters, each serving a distinct purpose:

Chapter 1 offers an introductory overview of the project's scope, goals, and the organizational structure of the report.

Chapter 2 delves into the background of deep learning, with a particular focus on feedforward neural networks (FFNN), as well as Embedded AI. Additionally, it provides insights into the FPGA board utilized and the custom instructions of the Nios II processor.

Chapter 3 provides a detailed description of the model employed in the project, along with an in-depth exploration of the designed acceleration system. This includes a comprehensive breakdown of its various components and their respective functionalities.

Chapter 4 engages in a comprehensive discussion of the results obtained both before and after implementing the designed accelerator. It further includes a comparative analysis of these results, offering valuable insights into the efficacy and performance of the acceleration system.

Chapter 2

Theoretical Background

2.1 Introduction

The integration of AI capabilities directly into hardware to enhance efficiency and performance is known as hardware acceleration in AI embedding. It involves the use of FPGAs, which provide customizable and reconfigurable circuits that achieve high-speed execution of AI algorithms. This chapter aims to provide a fundamental understanding of the principles and tools necessary for implementation phase.

2.2 Artificial Intelligence

AI has been viewed from a variety of perspectives. In 1985, Haugeland defined AI as "the exciting new effort to make computers think... machines with minds, in the full and literal sense," while Poole in 1998 defined AI as "Computational Intelligence is the study of the design of intelligent agents"[1].Regardless to those different approaches, AI now encompasses a significant portion of our lives.Its remarkable development over the previous decade, and its widespread application has increased efficiency and created benefits not only in the science and engineering fields, but also in the economy and society.

2.3 Feed Forward Neural Network

FFNN are the basic deep learning models. They have a unidirectional flow of information on a layer-by-layer basis with no loops or feedback connections as demonstrated in Figure 2.1. The network consists of an input layer of source neurons, and an output layer that provides the predicted feature.Finally, the hidden layers which serve as the neural network's computational engine; each hidden layer takes the weighted sum of the outputs, applies an activation function, and then passes the result on to the following layer [2].

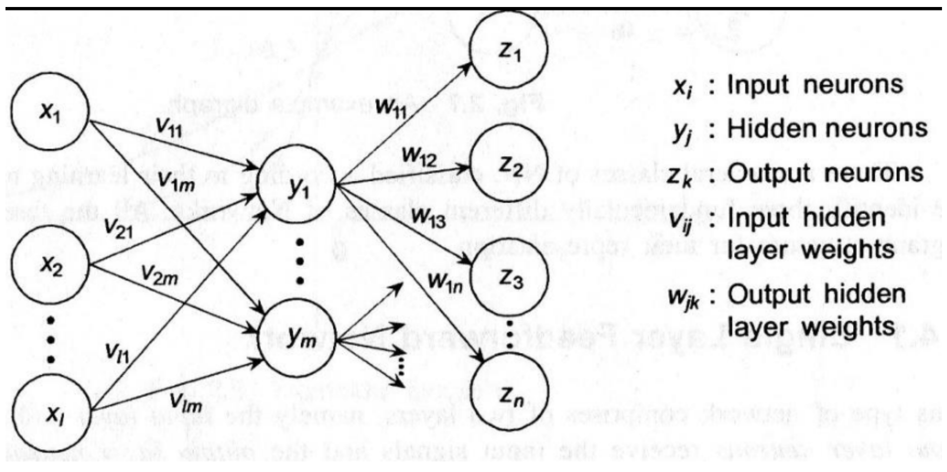


Figure 2.1: Feed Forward Neural Network

2.3.1 Forward Propagation

The forward propagation phase initiates the neural network process by receiving inputs and forwarding them through the network. At each hidden layer, the inputs' weighted sum is computed and passed through an activation function, introducing non-linearity. This process continues until the output layer, where predictions are made[3].

2.3.2 Backward Propagation

The backpropagation phase follows the forward propagation phase in the FFNN. Here, the predicted network output is compared to the expected output, and an error is computed using a specified function such as mean square error. This error is then propagated backward through the network, one layer at a time. The weights of the network are adjusted based on their contribution to the error, often using a gradient descent optimization algorithm[3].

2.4 Activation Functions

Activation functions are a fundamental component of neural network architectures, playing an important role in the introduction of non-linear properties into neuron outputs. This non-linearity enables the modeling of complex mappings between inputs and outputs.Activation functions are responsible for determining the activation status of neurons by computing weighted sum with bias terms as demonstrated in Equation 2.1.

$$\sum_{i=1}^{n}(x_i * w_i) + b \tag{2.1}$$

The absence of activation functions would limit neural networks to recording only linear relationships between input and output variables.

2.4.1 Leaky-ReLu

The rectified linear unit ReLu presented by Equation 2.2 is often the standard choice for activation functions due to its simplicity and effectiveness[2].

$$f(x) = \begin{cases} x, & \text{if } x \geq 0 \\ 0, & \text{if } x < 0 \end{cases} \tag{2.2}$$

However,The ReLU activation function suffers from drawbacks like dead neurons,where inputs below zero are set to zero, causing neurons to become inactive[4]. To address this issue we evaluate the leaky ReLu which is much the same as ReLu but includes a constant scale factor alpha as shown in Equation 2.3.

$$g(x) = \begin{cases} x, & \text{if } x \geq 0 \\ \alpha x, & \text{if } x < 0 \end{cases} \tag{2.3}$$

This factor ensures that inputs below zero are scaled by alpha,maintaining a small, non-zero gradient when the neuron is saturated or inactive [4] as demonstrated in Figure 2.2.

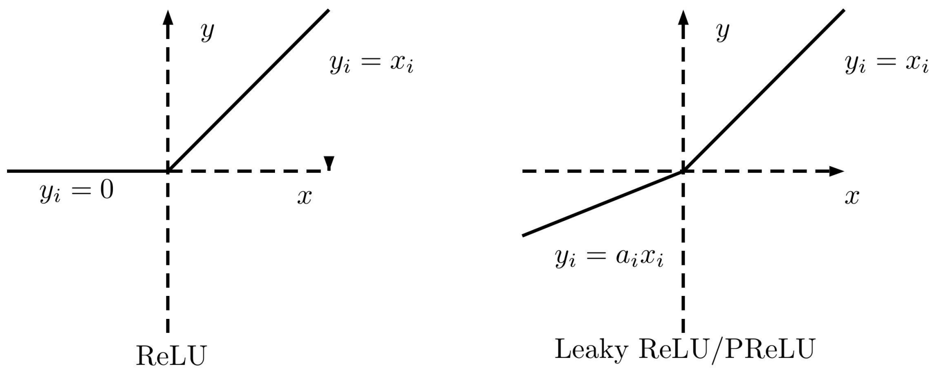


Figure 2.2: ReLu Activation Functions

2.4.2 Sigmoid

The sigmoid activation function has a historical significance as it was one of the initial activation functions used in neural networks. It can be represented by Equation 2.4.

$$\sigma(x) = \frac{1}{1+e^{-x}} \tag{2.4}$$

The purpose of the sigmoid activation function is to map a continuous real number to a value within the range of (0,1) as presented in Figure 2.3. This helps in maintaining a consistent and bounded input for the next layer, which in turn stabilizes the weights of the neural network[2]. The symmetrical nature of the sigmoid function around zero is a characteristic that has made it popular in many applications. Additionally, its ability to produce outputs close to zero or one while remaining differentiable makes it suitable for the backpropagation learning algorithm, which is crucial in neural networks. By enabling smooth gradient adjustments of network parameters during training, sigmoid function allows for effective learning of complex patterns and relationships in data.[5]

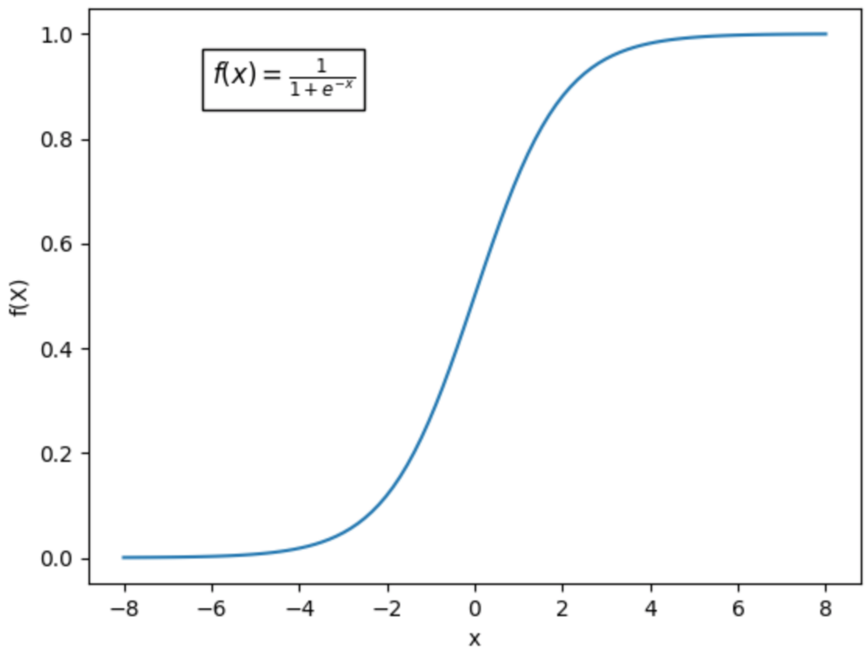


Figure 2.3: Sigmoid Activation Function

2.5 Embedded systems

An embedded system is a physical setup that combines hardware and software in order to serve specific applications. These systems are designed to operate reliably and efficiently, often with real-time constraints.

Embedded systems are found in a wide range of domains, including consumer electronics, automotive systems, industrial machinery, and medical devices.Key components of embedded systems include micro-controllers or microprocessors, memory, input and output devices, communication interfaces, power supply, real-time clocks, and software applications tailored to their intended functions[6].

Figure 2.4 illustrates the general architecture of an embedded system.

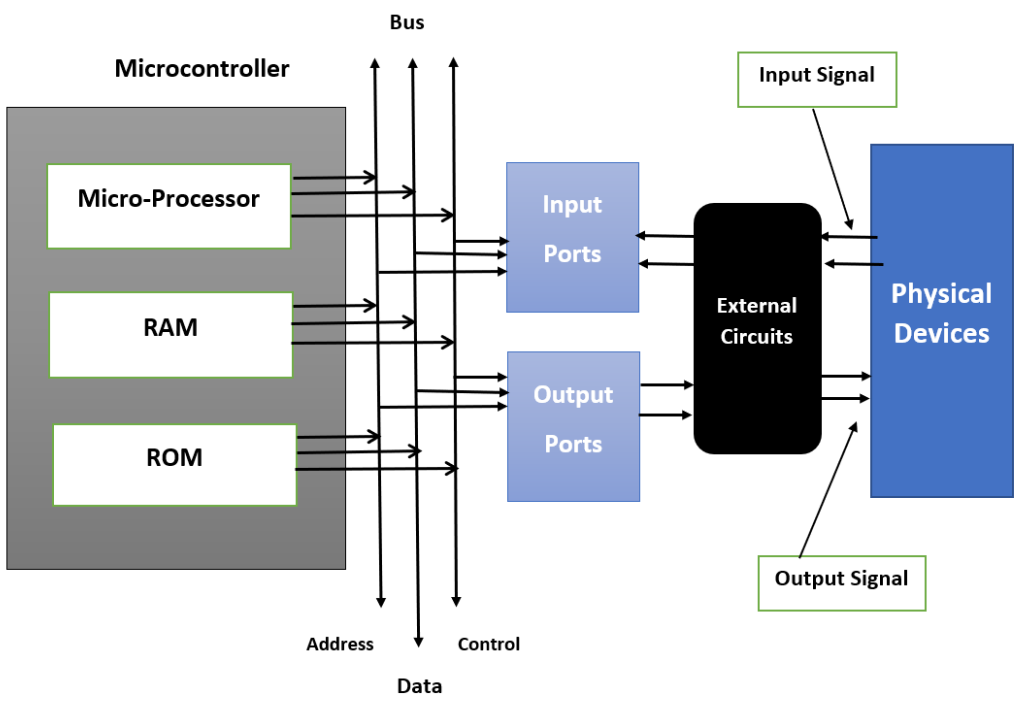


Figure 2.4: Embedded System General Architecture

2.6 Embedded AI

Embedded AI or Embedded Artificial Intelligence is a powerful technology that allows connected devices, systems, and objects to make decisions and perform complex tasks without relying on a steady internet connection or human assistance. By integrating AI capabilities such as machine learning and deep learning directly into the hardware and software, devices can perform data processing and decision-making without constantly needing to send data to the cloud[7].

2.6.1 Key Components of Embedded AI

Embedded AI relies on a complex network of interconnected components, including specialized hardware accelerators, software frameworks, and advanced algorithms. These components work together to enable devices to learn from data, adjust to different environments, and efficiently perform complex tasks. Dedicated hardware accelerators like GPUs, TPUs, and FPGAs handle the computationally intensive tasks, while software frameworks such as TensorFlow and PyTorch provide the essential tools for developing and deploying AI models. Advanced algorithms, such as deep learning and reinforcement learning, drive intelligent capabilities like object recognition, natural language processing, and autonomous navigation. By combining these components, embedded AI empowers devices to deliver sophisticated AI functionalities[8].

2.6.2 Hardware Acceleration

Hardware acceleration in embedded AI plays a crucial role in optimizing the performance of AI algorithms on devices that have limited computational resources. By offloading computationally demanding tasks from the CPU to specialized hardware accelerators, this approach enhances the efficiency and speed of AI inference processes, enabling real-time decision-making in embedded systems. Embedded AI systems utilize various types of hardware accelerators, each with its own advantages and trade-offs. GPUs excel in parallel processing, TPUs are specialized for accelerating tensor operations in neural networks, FPGAs provide flexibility and reconfigurability for diverse AI workloads, and ASICs offer high performance and power efficiency for specific AI applications. The adoption of hardware acceleration brings multiple benefits to embedded AI systems, including faster processing speed, efficient execution of AI algorithms, reduced power consumption, and extended battery life for portable devices. Furthermore, the parallel processing capabilities of accelerators enable scalable solutions for managing large-scale AI workloads, thereby augmenting the capabilities of embedded AI systems[9].

2.7 Field Programmable Gate Arrays

Field programmable gate arrays (FPGAs) are digital integrated circuits containing configurable logic blocks and interconnects, offering design engineers the flexibility to program them for diverse tasks. Initially introduced in the mid-1980s, FPGAs were mainly employed for basic functions like glue logic which is a simple logic circuit that is used to connect complex logic circuits together,and moderate state machines. However, with advancements in the early 1990s leading to larger and more complex FPGAs, They gained popularity in telecommunications and networking industries. By the late 1990s, their usage expanded significantly into consumer electronics, automotive, and industrial sectors[10].

In the early 2000s, the availability of high-performance FPGAs with millions of gates marked a milestone. These advanced FPGAs showcased features such as integrated microprocessor cores and high-speed input/output interfaces. Today, FPGAs are indispensable for a wide array of applications, including communication devices, software-defined radios, radar, image processing, and digital signal processing tasks. They are also integral components in system-on-chip designs, seamlessly blending hardware and software functionalities[10].

In terms of programming FPGAs, a hardware description language (HDL) is employed to design and configure the logic blocks and interconnects within an FPGA to perform a specific task. This programming approach differs from typical high-level programming languages. The two most commonly used HDLs for FPGA programming are VHDL (VHSIC Hardware Description Language) and Verilog. These languages provide a set of constructs and syntax specifically designed for describing and simulating digital circuits[11].

2.7.1 Cyclone V

The Cyclone V FPGA, released by Intel (formerly Altera), offers significant advancements over its predecessors. With improved performance and efficiency. The Cyclone V features a higher logic density and lower power consumption compared to older Cyclone series FPGAs. Its enhanced capabilities make it ideal for a wider range of applications, from telecommunications to embedded systems, providing greater flexibility and scalability for designers. Additionally, Cyclone V FPGAs incorporate advanced features such as integrated transceivers and hardened floating-point DSP blocks, further enhancing their suitability for demanding tasks[12, 13].

2.7.2 The DE10-Standard board

The DE10 Standard Board, developed by Terasic, is a feature-rich development kit centered around the Intel Cyclone V FPGA. It provides a wide range of features for various circuit implementations, from basic designs to multimedia projects. With components like SDRAM, user input interfaces, LEDs, displays, and audio/video capabilities as demonstrated in Figure 2.5, it offers a versatile platform for development and experimentation[14].

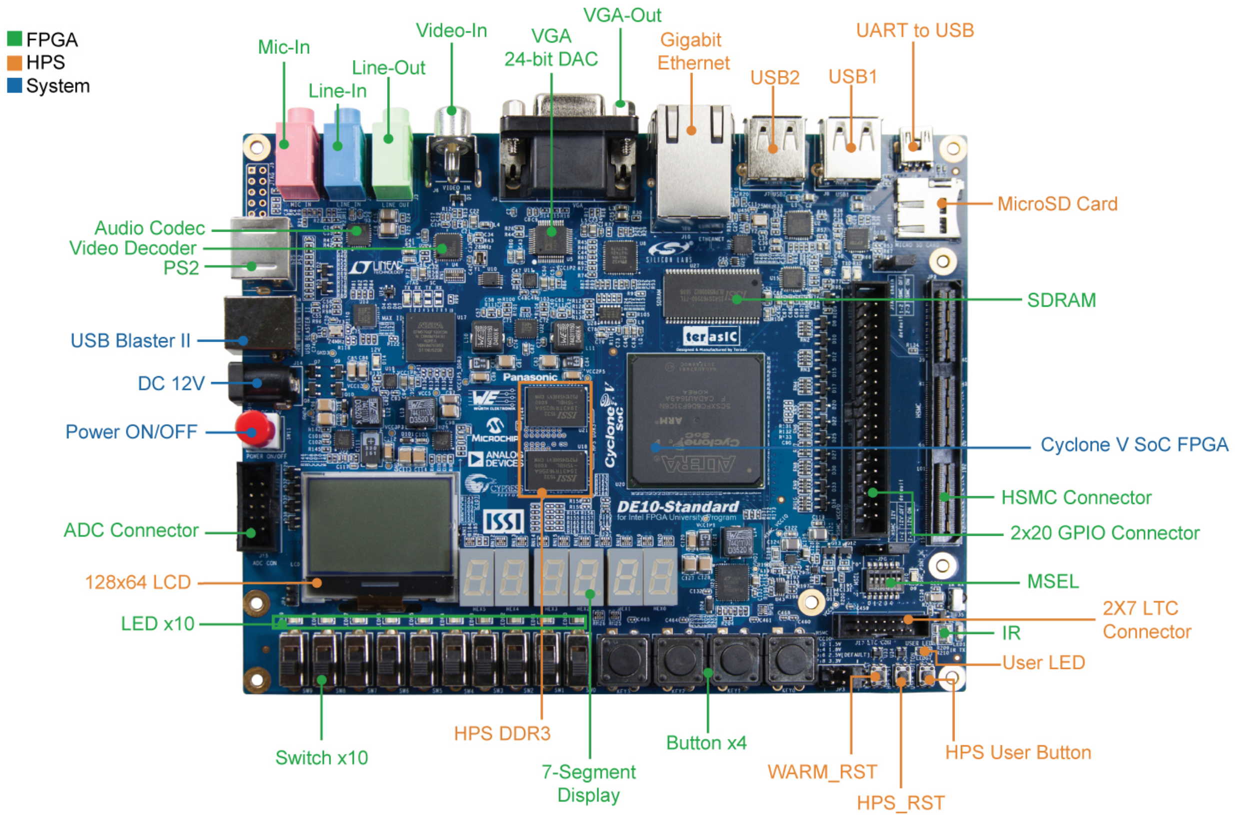


Figure 2.5: DE-10 Standard Board

2.7.3 Quartus Prime

Quartus Prime is an FPGA design software suite by Intel, providing a platform for developing, compiling, and debugging FPGA designs. It offers a range of features including synthesis, placement, routing, and simulation, enabling efficient and reliable FPGA development workflows.

Quartus Prime Standard supports various Intel FPGA families, empowering designers to create innovative and high-performance digital systems[15].

2.7.4 Platform Designer

Platform Designer is a feature integrated into the Intel Quartus Prime software that serves as a system integration tool. Its main purpose is to simplify the process of defining and integrating customized IP components (IP cores) into your design. By allowing the packaging and integration of custom IP components with Intel and third-party IP components, Platform Designer promotes design reuse. It automates the creation of interconnect logic based on the high-level connectivity specifications provided, eliminating the need for writing HDL code for system-level connections, which is prone to errors and time-consuming. One of the notable features of Platform Designer is hierarchical isolation, which separates the system interconnect and IP components. This allows for independent parameter modification of individual IP components without regenerating the entire system or other IP components within it. Similarly, changes in the system connectivity do not require the regeneration of any IP components[16].

2.7.5 Nios-II

The Nios II processor is a soft-core RISC processor with a 32-bit instruction set, data path, and address space. It features 32 general-purpose registers, 32 interrupt sources, and support for external interrupt controllers. It has multiplication and division capabilities, as well as optional floating-point instructions. The processor has a barrel shifter and interfaces for on-chip and off-chip peripherals and memories. It includes a debug module and offers optional memory management and protection units. It is supported by the GNU C/C++ tool chain and Nios II software development tools. Nios II processor systems are like microcontrollers, with the processor, peripherals, and memory all integrated on a single chip. They have a consistent instruction set and programming model and can achieve performance up to 250 DMIPS[17]. Figure 2.6 shows a general Block Diagram of Nios-II.

The functional units specified in the Nios II architecture include[17]:

1. Register file
2. Arithmetic logic unit (ALU)
3. Interface to custom instruction logic
4. Exception controller
5. Internal or external interrupt controller
6. Instruction bus
7. Data bus
8. Memory management unit (MMU)
9. Memory protection unit (MPU)
10. Instruction and data cache memories
11. Tightly-coupled memory interfaces for instructions and data
12. JTAG debug module

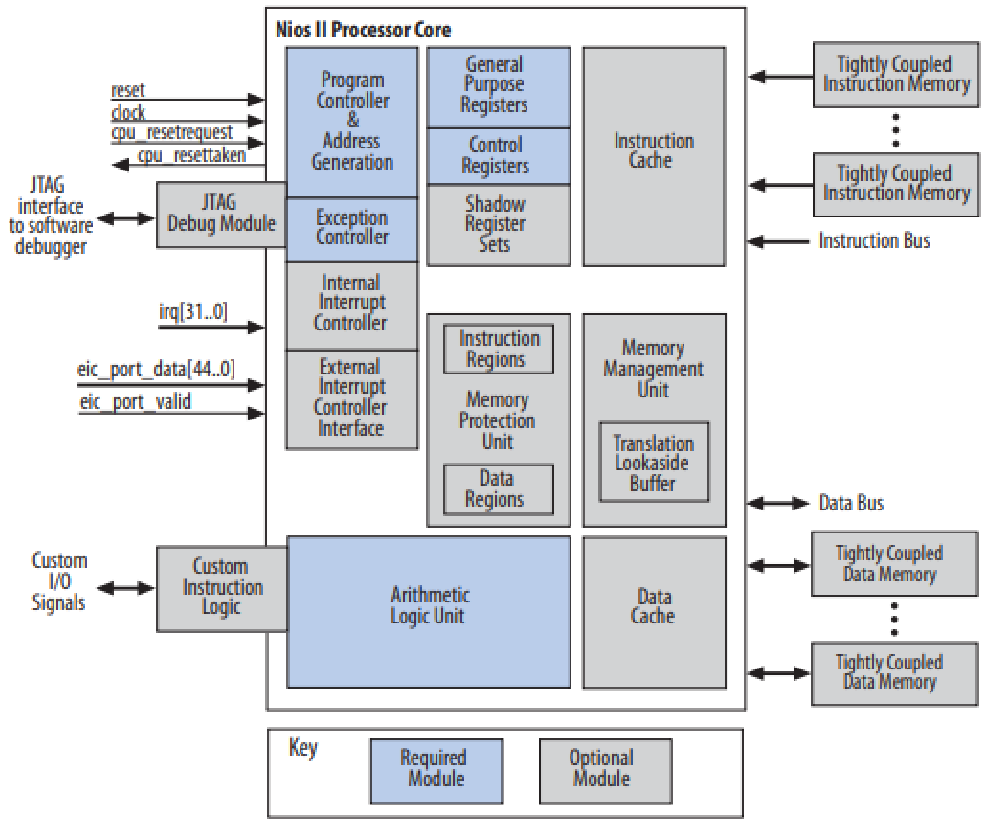


Figure 2.6: Nios-II Core Block Diagram

2.7.5.1 Nios-II/f vs Nios-II/e

The Nios II/e economy core is specifically designed to have the smallest possible core size. It operates by executing a maximum of one instruction per six clock cycles, allows for full 32-bit addressing, and can access up to 4 GB of external address space. It enables the inclusion of custom instructions and supports the use of the JTAG debug module. However, it lacks hardware support for potential unimplemented instructions and does not feature an instruction or data cache. Additionally, it does not perform branch prediction. In cases where instructions are not supported, the Nios II/e core emulates them through software[17].

However,the Nios II/f fast core is designed for high execution performance but has a larger core size. Its key features include separate optional instruction and data caches, support for an MMU and MPU for memory management, a 6-stage pipeline for efficient execution, dynamic or static branch prediction, optional hardware multiply/divide/shift operations, support for custom instructions, and JTAG debug module support. It also offers features like external interrupt control, shadow register sets for improved latency, tightly-coupled memory, and optional ECC support[17].

2.7.6 Nios-II Custom Instructions

Custom instructions in the Nios II processor allow for tailoring the processor to meet specific application requirements. They enable the acceleration of time-critical software algorithms by converting them to custom hardware logic blocks, thereby providing an easy way to experiment with hardware-software tradeoffs. The custom instruction logic connects directly to the processor's ALU as demonstrated in Figure 2.7, and like native instructions, can take values from up to two source registers and optionally write back a result to a destination register. Custom instructions can significantly improve system performance, often achieving performance gains from 10 to 100 times that of software-based operations. These instructions can be integrated into the hardware and tested with software on reprogrammable Intel FPGAs. From a software perspective, custom instructions appear as machine-generated assembly macros or C functions, eliminating the need to understand assembly language[18].

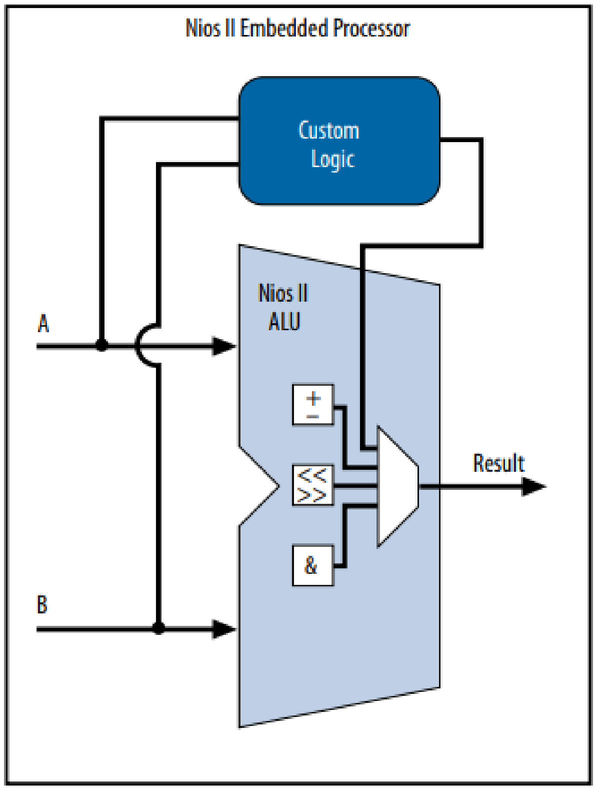


Figure 2.7: Custom Instruction Block Diagram

Custom instructions can be categorized into four types: combinational instructions, multi-cycle instructions, extended instructions, and internal register-file instructions.Figure 2.8 demonstrates that to transition from one type of custom instruction to another, only specific hardware parameters need to be adjusted.

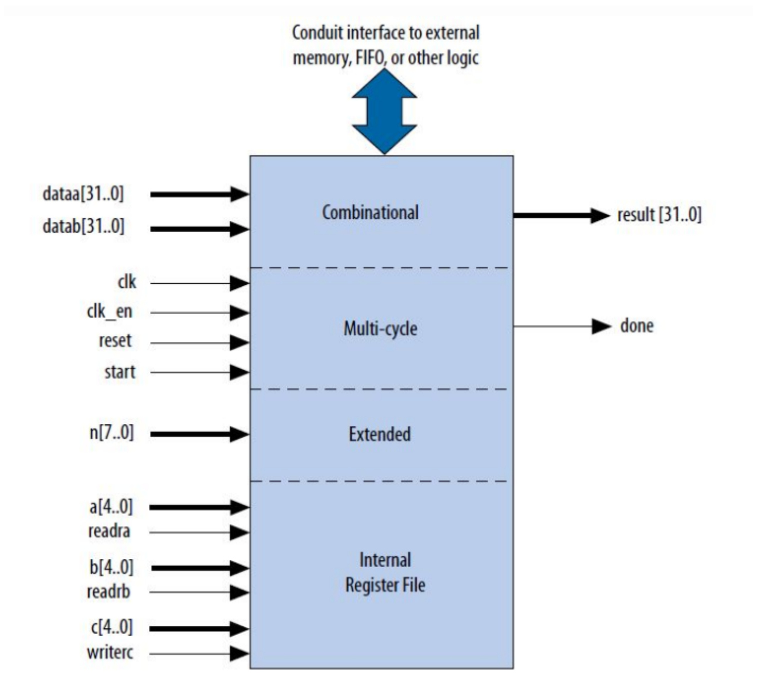


Figure 2.8: Custom Instruction's Types

2.8 Conclusion

To sum up, this chapter has introduced the concept of hardware acceleration in AI embedding, which involves integrating AI capabilities directly into hardware to boost efficiency and performance. Focusing on the utilization of FPGAs for their customizable and reconfigurable circuitry, the chapter aims to equip readers with the foundational knowledge and tools essential for the implementation phase of hardware-accelerated AI. In the subsequent sections, we will delve deeper into the principles, methodologies, and practical applications of this innovative approach.

Chapter 3

Design and Implementation

3.1 Introduction

This chapter details the design and implementation of the hardware system utilized for the neural network application, specifically focusing on the integration of the various custom hardware blocks to enhance the computational efficiency of the neural network.

3.2 Neural Network Architecture

The chosen Network is a fully connected Feed forward Network written in C language for the use in Micro-controllers. The Network is generated from a library that contains all the functions and codes needed for training, testing and predicting.

The neural network architecture consists of multiple layers, each containing a specified number of neurons, with weights and biases connecting the neurons between adjacent layers. Various activation functions introduce non-linearity into the model. During initialization, the network is set up with zero depth, and layers are dynamically added, allocating memory for neurons, weights, biases, and activations. Forward propagation calculates the output by applying activation functions to the weighted sums of inputs. During training, backpropagation is used to compute weight adjustments based on the error between predicted and target values, and these adjustments are applied to update the weights. Predictions are made by performing forward propagation on input data .The functions used are shown in Table 3.1. The model can be saved to a file and loaded later, and versioning ensures compatibility and tracks updates[19].

While the original training code used to store the weights of the network as ASCII file of floating point values; it was modified to store and read the weights of the model from a floating point array, since the Nios-II does not have the ability to operate on files during the execution mode.

The model was trained and tested several times on the computer,the structure, the number of layers and their activation functions where modified and tuned until the desired result was achieved.The final Network structure includes one input layer without an activation function, one output layer with Sigmoid activation function and varying numbers of hidden layers as we will see in the coming sections.

Table 3.1: Neural Network API Functions

Function	Description
InitializeNN()	Initialize neural network by allocating memory, setting Depth to 0, and all the pointers to NULL
ForwardPropagation()	Calculate neuron values in each layer using weighted sum and activation function
TrainNN(inputs, targets, learning_rate)	Train the neural network using input-output pairs and learning rate
PredictNN(inputs)	Predict output for given inputs using the trained neural network
LoadNNFromFile(path)	Load neural network parameters and weights from a model file
SaveNNToFile(nn, path)	Save neural network parameters and weights to a model file
GetNNVersion()	Return the version of the neural network library

3.3 Overall System Design

The hardware system of this project consists of on-chip components. As depicted in the block diagram in Figure 3.1, the on-chip hardware system contains the SoPC and non-SoPC parts of the system implemented using Platform Designer (formerly Qsys).The SoPC part includes the Nios-II/f processor due to its good features, SDRAM controller, phase-locked loop (PLL), JTAG interface, I/O peripherals, floating-point hardware, and VHDL LUT custom blocks.While,the non-SoPC part includes a 64Mb SDRAM .

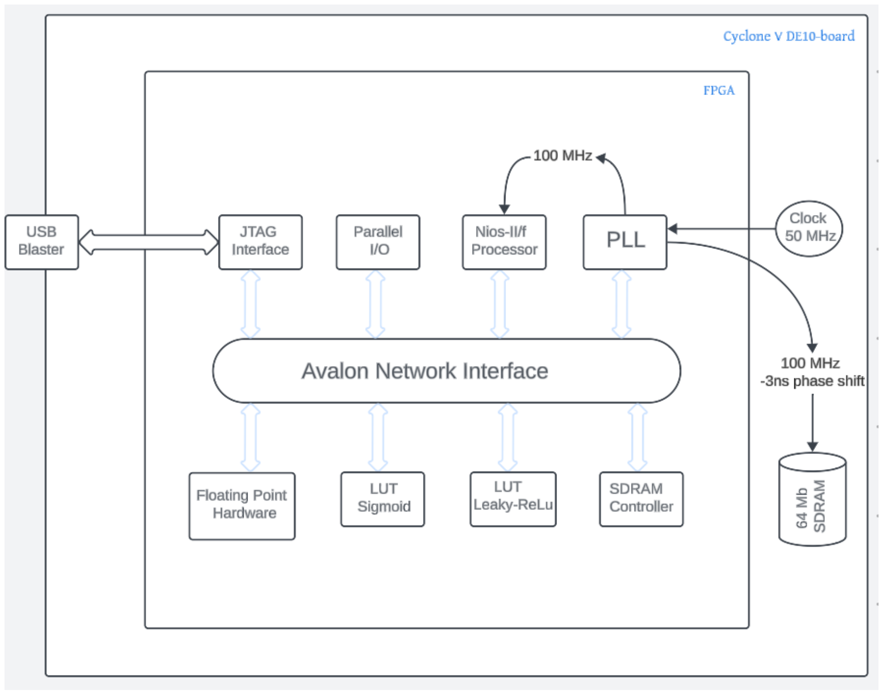
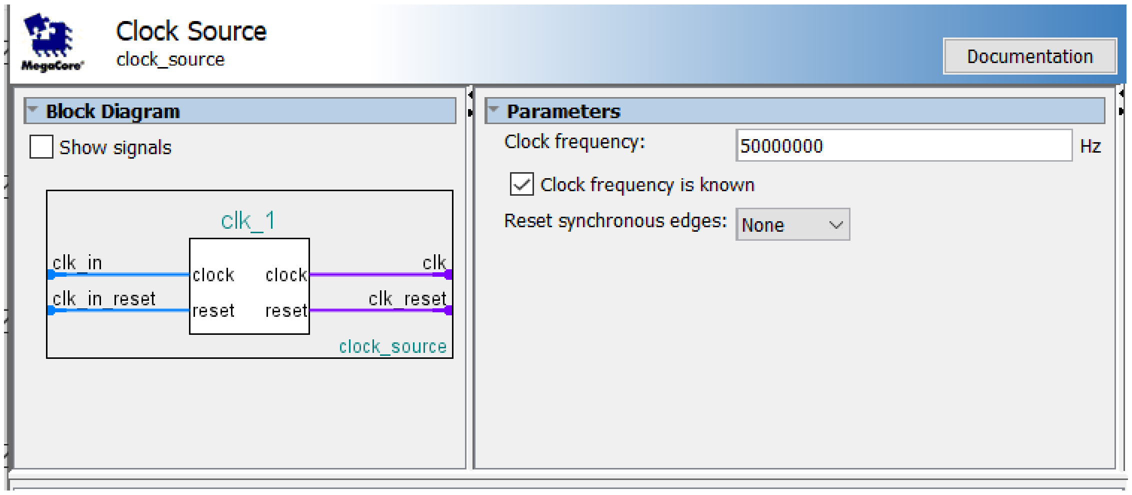


Figure 3.1: Overall System Design

3.4 SoPC System Implementation

The system has been implemented on a Cyclone V 5CSXFC6D6F31C6 FPGA, which is available on the DE10-Standard board. The SoPC system has been designed using the Platform Designer tool of the Quartus Prime Standard IDE. Different components have been selected and correctly interconnected to create the desired system configuration as following :

- Clock Source



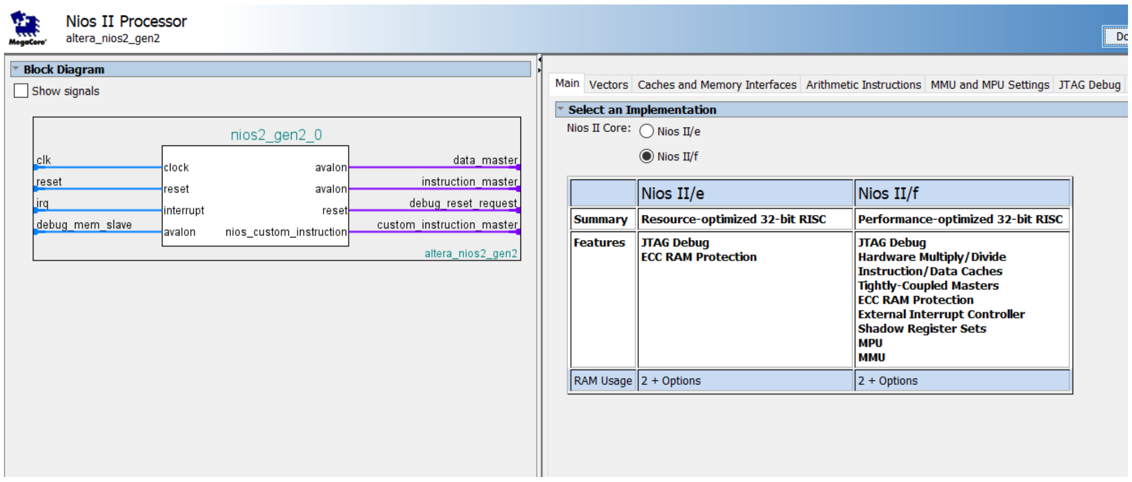
Clock frequency: 50000000 Hz

[✓] Clock frequency is known

Reset synchronous edges: None

Figure 3.2: Clock Source

- Nios-II/f Processor

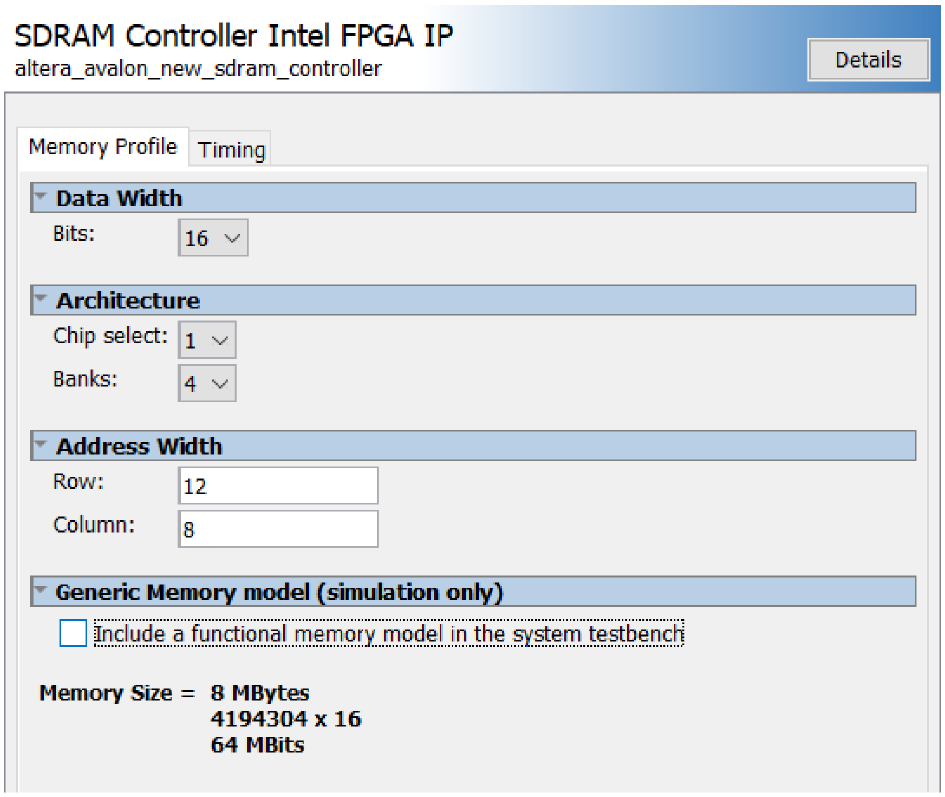


Nios II Core: Nios II/e [✓ Nios II/f]

	Nios II/e	Nios II/f
Summary	**Resource-optimized 32-bit RISC**	**Performance-optimized 32-bit RISC**
Features	**JTAG Debug** **ECC RAM Protection**	**JTAG Debug** **Hardware Multiply/Divide** **Instruction/Data Caches** **Tightly-Coupled Masters** **ECC RAM Protection** **External Interrupt Controller** **Shadow Register Sets** **MPU** **MMU**
RAM Usage	2 + Options	2 + Options

Figure 3.3: Nios-II/f

- SDRAM Controller: manages the interface and data transfer between the FPGA and SDRAM, ensuring efficient memory access and timing compliance.


SDRAM Controller Intel FPGA IP
altera_avalon_new_sdram_controller

Data Width
Bits: 16

Architecture
Chip select: 1
Banks: 4

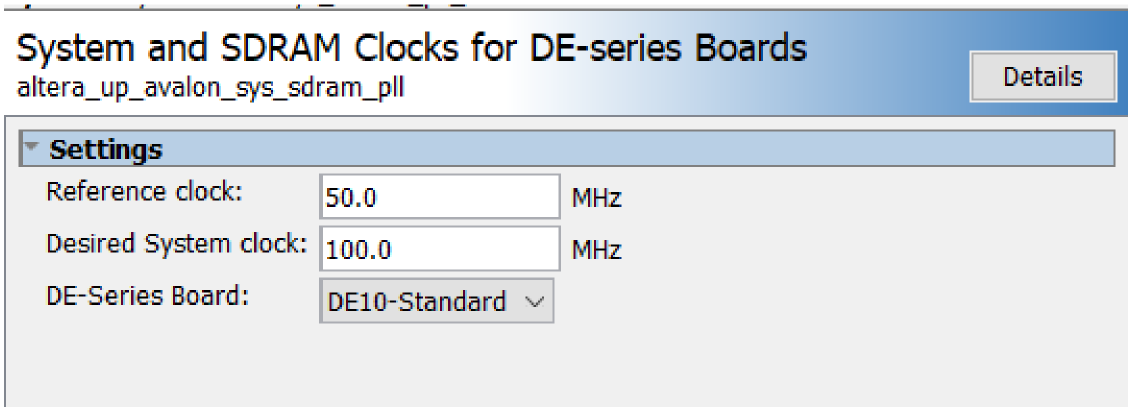
Address Width
Row: 12
Column: 8

Generic Memory model (simulation only)
☐ Include a functional memory model in the system testbench

Memory Size = 8 MBytes
4194304 x 16
64 MBits

Figure 3.4: SDRAM Controller

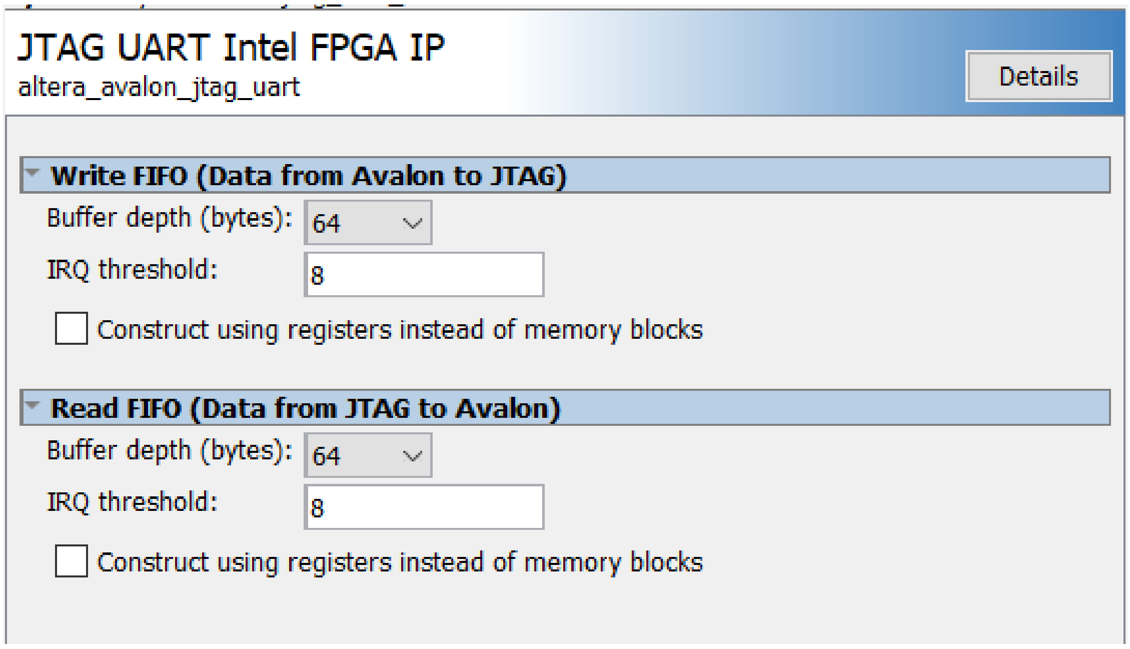
- PLL : generates and stabilizes the necessary clock signals for synchronous operation and timing alignment between components.


System and SDRAM Clocks for DE-series Boards
altera_up_avalon_sys_sdram_pll

Settings
Reference clock: 50.0 MHz
Desired System clock: 100.0 MHz
DE-Series Board: DE10-Standard

Figure 3.5: PLL

- JTAG UART that facilitates communication between the Nios processor and a host computer for debugging and data exchange



JTAG UART Intel FPGA IP
altera_avalon_jtag_uart

Write FIFO (Data from Avalon to JTAG)

Buffer depth (bytes): 64

IRQ threshold: 8

☐ Construct using registers instead of memory blocks

Read FIFO (Data from JTAG to Avalon)

Buffer depth (bytes): 64

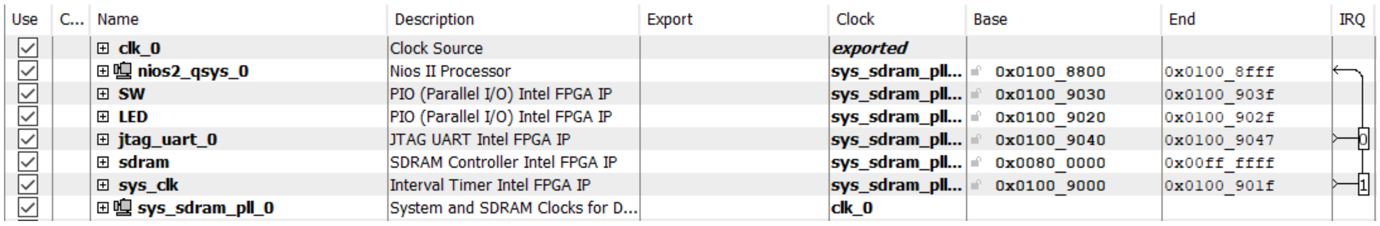
IRQ threshold: 8

☐ Construct using registers instead of memory blocks

Figure 3.6: JTAG UART

- Parllel I/O : mostly LEDs an Switches that have been used to test the functionality of the system at the beginning of the project.

Figure 3.7 shows the overall SoPC system.



Use	C...	Name	Description	Export	Clock	Base	End	IRQ
✓		clk_0	Clock Source		*exported*			
✓		nios2_qsys_0	Nios II Processor		sys_sdram_pll...	0x0100_8800	0x0100_8fff	
✓		SW	PIO (Parallel I/O) Intel FPGA IP		sys_sdram_pll...	0x0100_9030	0x0100_903f	
✓		LED	PIO (Parallel I/O) Intel FPGA IP		sys_sdram_pll...	0x0100_9020	0x0100_902f	
✓		jtag_uart_0	JTAG UART Intel FPGA IP		sys_sdram_pll...	0x0100_9040	0x0100_9047	0
✓		sdram	SDRAM Controller Intel FPGA IP		sys_sdram_pll...	0x0080_0000	0x00ff_ffff	
✓		sys_clk	Interval Timer Intel FPGA IP		sys_sdram_pll...	0x0100_9000	0x0100_901f	1
✓		sys_sdram_pll_0	System and SDRAM Clocks for D...		clk_0			

Figure 3.7: Overall System Design

3.4.1 System Block Diagram

The HDL file of the system was generated then and it was possible to instantiate it as a block diagram as shown in Figure 3.8

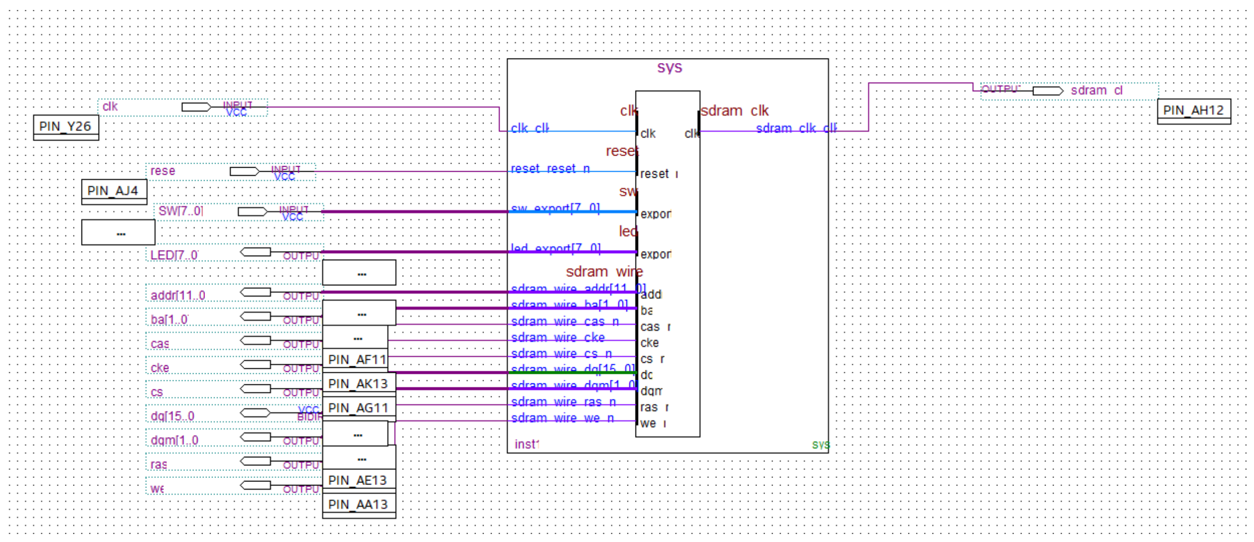


Figure 3.8: SoPC Block Diagram File

3.5 PLL System Block

The Phase-Locked Loop (PLL) is crucial for managing timing and synchronization between the system clock and the SDRAM clock. It can generate multiple clock frequencies from a single reference clock and adjust the phase of specific clock signals to either lead or lag the system clock by a set amount, typically a few nanoseconds, to compensate for clock misalignment. In our designs, ensuring the SDRAM clock signal leads the Nios II system clock by approximately 3 nanoseconds is essential for proper operation.

3.6 Altera NIOS II JTAG Debug Module

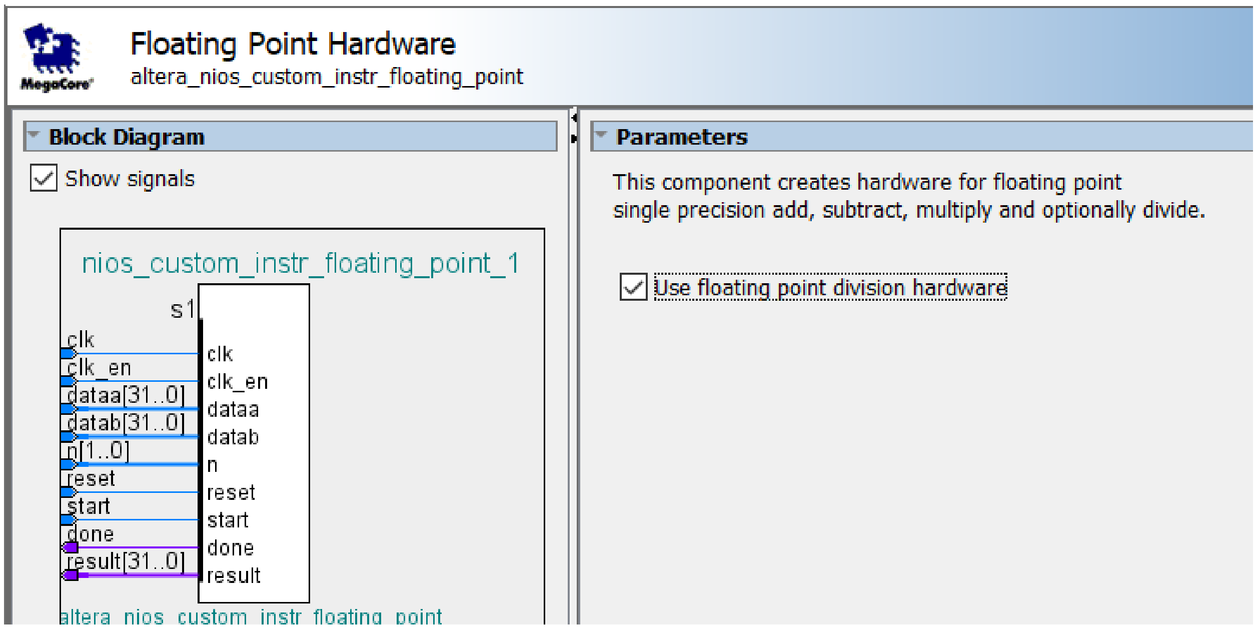
The Nios II architecture supports a JTAG debug module that enables on-chip emulation capabilities, allowing for remote processor control from a host PC. Software debugging tools operating on the PC interface with the JTAG debug module to offer functionalities such as:

- Program download to memory.
- Execution control, including start and stop commands.
- Breakpoint and watchpoint configuration.
- Inspection of registers and memory.
- Real-time execution trace data collection.

3.7 Floating point Unit

To improve the efficiency of floating-point operations, we utilized Altera's specialized floating-point hardware , specifically FPH1.This dedicated hardware module is a considered as a custom instruction designed to handle single-precision floating-point operations like addition, subtraction, and multiplication while following the IEEE Std 754-1985 standards.Thus the software emulation for the floating point operations was outperformed.

By integrating this hardware solution, we achieved a notable acceleration in floating-point computations within the neural network. This optimization greatly enhances the overall efficiency and performance of the neural network's computations.



This component creates hardware for floating point single precision add, subtract, multiply and optionally divide.

[✓] Use floating point division hardware

Figure 3.9: Floating Point Hardware

3.8 Activation function Custom instructions

The activation functions within the neural network posed a significant time-consuming challenge due to their reliance on floating-point computations, particularly when utilizing software emulation, which proved excessively slow. To address this issue, we proposed replacing the software-based computations with hardware-based solutions.Thus, Lookup tables were then implemented to replace the software activation functions.

In this context, a lookup table (LUT) is a data structure used to expedite the computation of activation functions within the neural network. It stores precomputed values for these functions, allowing for rapid retrieval of output values based on input indices. By implementing lookup tables, the neural network can efficiently replace software-based activation functions, significantly enhancing computational speed and performance.

3.8.1 Leaky-ReLu Custom Instruction

The Leaky ReLU Look-Up Table (LUT) is implemented as a variable multi-cycle custom instruction, designed to be both flexible and efficient. It operates within the range of -3 to 3 based on the weights values we obtained from the training and uses 16 carefully calculated sample float values. These values are obtained through Python code and are assigned specific outputs based on the input range. For inputs within the range of -3 to 0, a specific output is determined, while for inputs within the range of 0 to 3, the output is simply the input itself. This specialized instruction employs an IF-THEN-ELSE chain IF-THEN-ELSE chain as shown in the pseudo-code below.

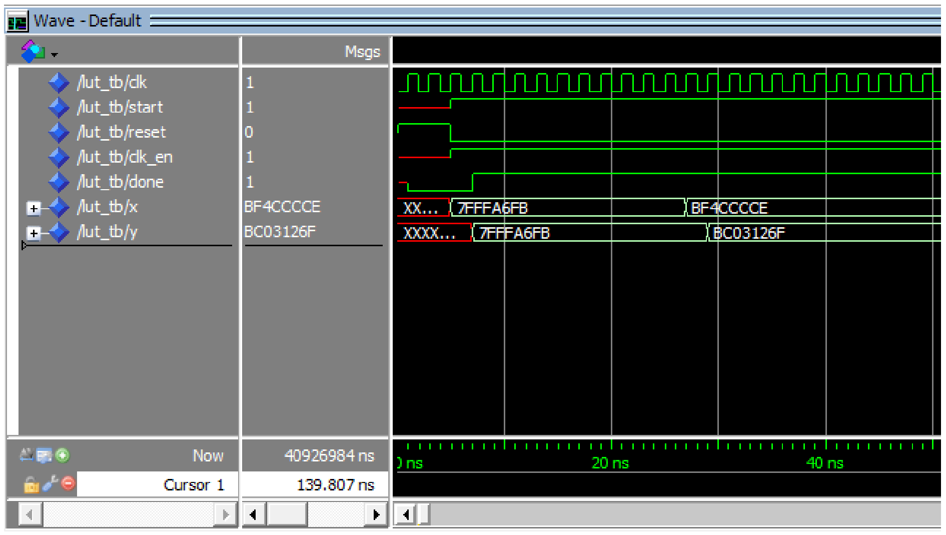
Algorithm 1 Leaky-ReLU LUT pseudo-code

```
1:  procedure LUT_OPERATION(clk)
2:      if rising_edge(clk) then
3:          if reset then
4:              done ← 0
5:          else if clk_en and start then
6:              if $x$ is within specified intervals then
7:                  if $x \geq 0$ and $x \leq 3$ then
8:                      $y \leftarrow x$
9:                  else if $x \geq -3$ and $x < 0$ then
10:                     if $x$ is in Interval 1 then
11:                         $y \leftarrow$ Value 1
12:                     else if $x$ is in Interval 2 then
13:                         $y \leftarrow$ Value 2
14:                     else if $x$ is in Interval 3 then
15:                         $y \leftarrow$ Value 3
16:                     else
17:                         $y \leftarrow$ Default Value
18:                     end if
19:                 end if
20:             end if
21:             done ← 1
22:         end if
23:     end if
24: end procedure
```

Before integrating the Custom Instructions block into the System Design, its functionality was verified by simulating it using the Model-sim Altera simulator. The simulation was conducted using a test bench, Figure 3.10 and Listing 3.1 demonstrate the test bench and the results of the simulation .

```
library ieee;
use ieee.std_logic_1164.all;

entity lut_tb is
end entity lut_tb;

architecture test of lut_tb is
begin
    clk <= not clk after 1 ns;
    reset <= '1', '0' after 5 ns;

    dut: entity work.lut
    port map (
        clk => clk,
        start => start,
        done => done,
        clk_en => clk_en,
        reset => reset,
        x => x,
        y => y
    );

    stimulus: process
    begin
        wait until (reset = '0');
        x <= x"7FFFA6FB";

        clk_en <= '1';
        start <= '1';
        wait until (done = '1');
        wait for 20 ns;
        start <= '0';

        x <= x"BF4CCCE";
        start <= '1';
        clk_en <= '1';
        wait until (done = '1');

        report("end simulation");
    end process stimulus;

end architecture;
```

Listing 3.1: VHDL Testbench Code for Leaky-ReLU



```
/lut_tb/clk      1
/lut_tb/start    1
/lut_tb/reset    0
/lut_tb/clk_en   1
/lut_tb/done     1
/lut_tb/x        BF4CCCCE     XX...   7FFFA6FB   BF4CCCCE
/lut_tb/y        BC03126F     XXXX... 7FFFA6FB   BC03126F

Now       40926984 ns
Cursor 1  139.807 ns
```

Figure 3.10: Leaky-ReLu Simulation

3.8.2 Sigmoid Custom Instruction

Similar to the Leaky-ReLU mentioned earlier, the Sigmoid LUT is implemented as a variable multi-cycle custom instruction. This instruction operates within the range of -3 to 3 and uses 16 precisely calculated sample float values. These values are derived using Python code and are mapped to specific outputs based on the input range. This custom instruction utilizes an IF-THEN-ELSE chain, as demonstrated in the pseudocode below.

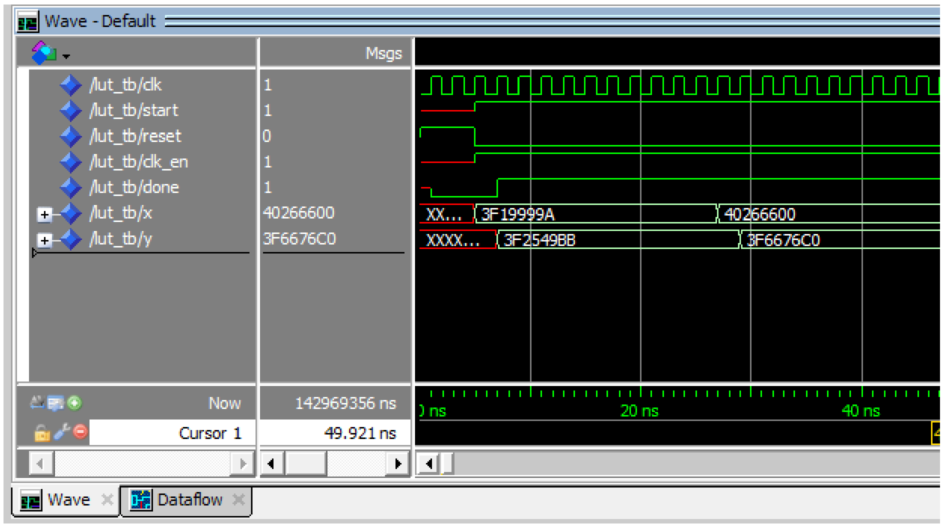
Algorithm 2 Sigmoid LUT pseudo-code

```
1:  procedure LUT_OPERATION(clk)
2:      if rising_edge(clk) then
3:          if reset = '1' then
4:              done ← '0'
5:          else if clk_en = '1' and start = '1' then
6:              if $x$ is in Interval 1 then
7:                  $y$ ← Value 1
8:              else if $x$ is in Interval 2 then
9:                  $y$ ← Value 2
10:             else if $x$ is in Interval 3 then
11:                 $y$ ← Value 3
12:             else
13:                 $y$ ← Default Value
14:             end if
15:             done ← '1'
16:         end if
17:     end if
18: end procedure
```

Before integrating the Custom Instructions block into the System Design, its functionality was verified by simulating it using the Model-sim Altera simulator. The simulation was conducted using a test bench, Figure 3.11 and Listing 3.2 demonstrate the test bench and the results of the simulation .

```
library ieee;
use ieee.std_logic_1164.all;
entity lut_tb is
end entity lut_tb;

architecture test of lut_tb is
begin
    clk <= not clk after 1 ns;
    reset <= '1', '0' after 5 ns;

    dut: entity work.lut_sig
    port map (
        clk => clk,
        start => start,
        done => done,
        clk_en => clk_en,
        reset => reset,
        x => x,
        y => y
    );

    stimulus: process
    begin
        wait until (reset = '0');
        x <= x"3F19999A";

        clk_en <= '1';
        start <= '1';
        wait until (done = '1');
        wait for 20 ns;
        start <= '0';

        x <= x"40266600";
        start <= '1';
        clk_en <= '1';
        wait until (done = '1');

        report("end simulation");
    end process stimulus;

end architecture;
```

Listing 3.2: VHDL Testbench Code for Sigmoid

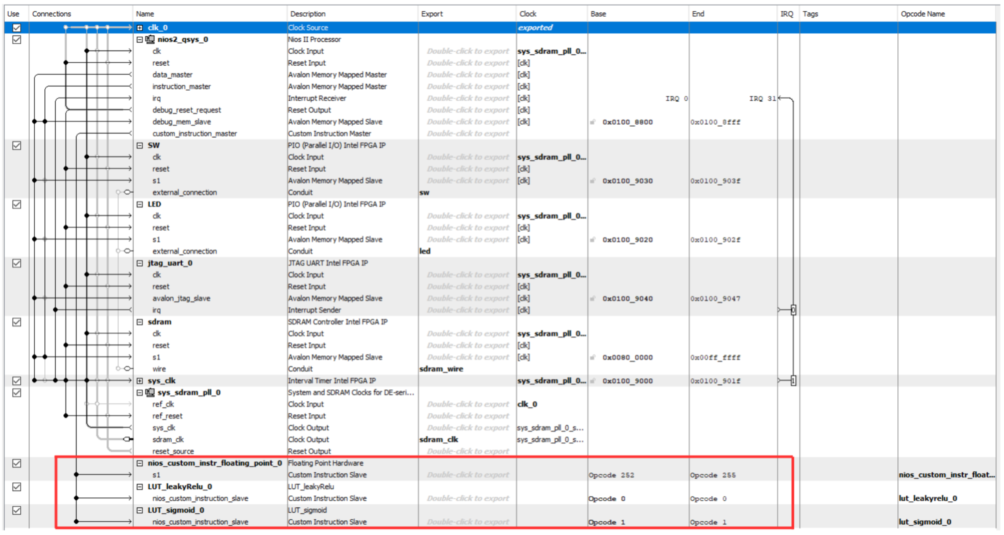


```
/lut_tb/clk      1
/lut_tb/start    1
/lut_tb/reset    0
/lut_tb/clk_en   1
/lut_tb/done     1
/lut_tb/x        40266600     XX...   3F19999A   40266600
/lut_tb/y        3F6676C0     XXXX... 3F2549BB   3F6676C0
Now       142969356 ns
Cursor 1  49.921 ns
```

Figure 3.11: Sigmoid Simulation

3.9 Final System Integration

After Carefully designing and explaining each individual component, we now present the fully integrated system.Figure 3.12 provides a representation of the overall architecture, showcasing the interconnections and interactions between the various modules.The Floating Point Hardware custom instruction was introduced with opcodes 252 to 255, each representing a specific operation: 255 for division, 254 for subtraction, 253 for addition, and 252 for multiplication. Furthermore, two additional lookup tables were implemented as custom instructions, with opcode 0 assigned to the leaky-ReLU custom instruction and opcode 1 assigned to the Sigmoid custom instruction.It is important to note that the Nios II processor is capable of supporting up to 256 unique custom instructions via custom opcodes.



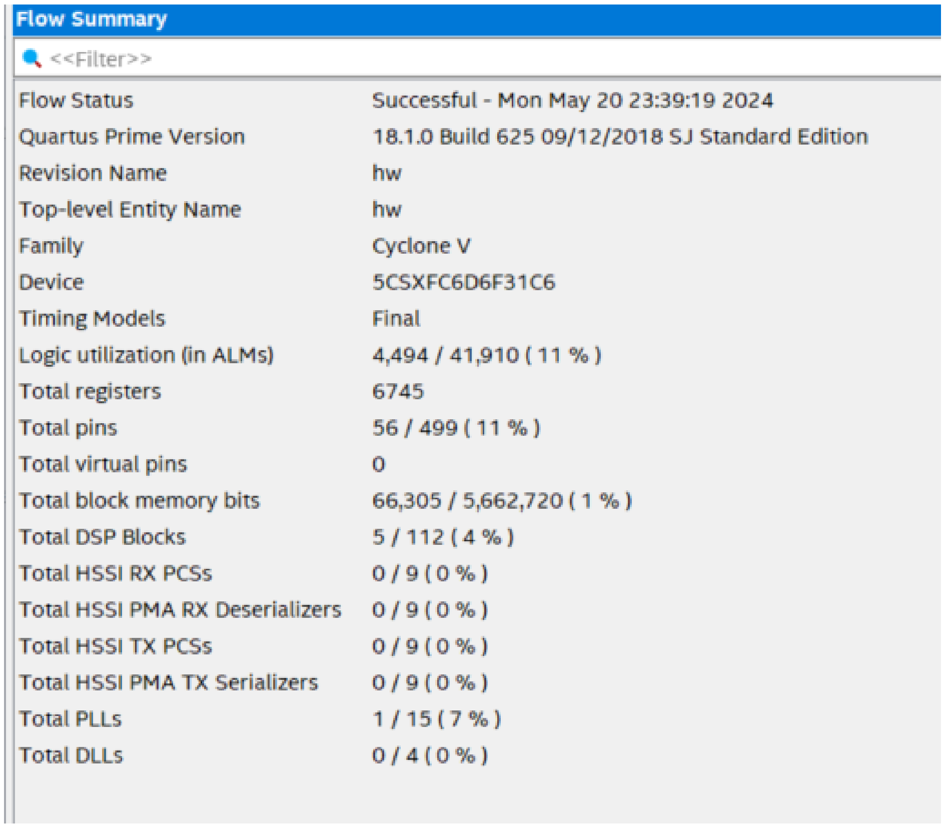
Use	Connections	Name	Description	Export	Clock	Base	End	IRQ	Tags	Opcode Name
✓		⊞ **clk_0**	Clock Source		*exported*					
✓		⊟ **nios2_qsys_0**	Nios II Processor							
		clk	Clock Input	Double-click to export	**sys_sdram_pll_0...**					
		reset	Reset Input	Double-click to export	[clk]					
		data_master	Avalon Memory Mapped Master	Double-click to export	[clk]					
		instruction_master	Avalon Memory Mapped Master	Double-click to export	[clk]					
		irq	Interrupt Receiver	Double-click to export	[clk]	IRQ 0	IRQ 31			
		debug_reset_request	Reset Output	Double-click to export	[clk]					
		debug_mem_slave	Avalon Memory Mapped Slave	Double-click to export	[clk]	0x0100_8800	0x0100_8fff			
		custom_instruction_master	Custom Instruction Master	Double-click to export						
✓		⊟ **SW**	PIO (Parallel I/O) Intel FPGA IP							
		clk	Clock Input	Double-click to export	**sys_sdram_pll_0...**					
		reset	Reset Input	Double-click to export	[clk]					
		s1	Avalon Memory Mapped Slave	Double-click to export	[clk]	0x0100_9030	0x0100_903f			
		external_connection	Conduit	**sw**						
✓		⊟ **LED**	PIO (Parallel I/O) Intel FPGA IP							
		clk	Clock Input	Double-click to export	**sys_sdram_pll_0...**					
		reset	Reset Input	Double-click to export	[clk]					
		s1	Avalon Memory Mapped Slave	Double-click to export	[clk]	0x0100_9020	0x0100_902f			
		external_connection	Conduit	**led**						
✓		⊟ **jtag_uart_0**	JTAG UART Intel FPGA IP							
		clk	Clock Input	Double-click to export	**sys_sdram_pll_0...**					
		reset	Reset Input	Double-click to export	[clk]					
		avalon_jtag_slave	Avalon Memory Mapped Slave	Double-click to export	[clk]	0x0100_9040	0x0100_9047			
		irq	Interrupt Sender	Double-click to export	[clk]			0		
✓		⊟ **sdram**	SDRAM Controller Intel FPGA IP							
		clk	Clock Input	Double-click to export	**sys_sdram_pll_0...**					
		reset	Reset Input	Double-click to export	[clk]					
		s1	Avalon Memory Mapped Slave	Double-click to export	[clk]	0x0080_0000	0x00ff_ffff			
		wire	Conduit	**sdram_wire**						
✓		⊞ **sys_clk**	Interval Timer Intel FPGA IP		**sys_sdram_pll_0...**	0x0100_9000	0x0100_901f	1		
✓		⊟ **sys_sdram_pll_0**	System and SDRAM Clocks for DE-seri...							
		ref_clk	Clock Input	Double-click to export	**clk_0**					
		ref_reset	Reset Input	Double-click to export						
		sys_clk	Clock Output	Double-click to export	sys_sdram_pll_0_s...					
		sdram_clk	Clock Output	**sdram_clk**	sys_sdram_pll_0_s...					
		reset_source	Reset Output	Double-click to export						
✓		⊟ **nios_custom_instr_floating_point_0**	Floating Point Hardware							
		s1	Custom Instruction Slave	Double-click to export		Opcode 252	Opcode 255			**nios_custom_instr_float...**
✓		⊟ **LUT_leakyRelu_0**	LUT_leakyRelu							
		nios_custom_instruction_slave	Custom Instruction Slave	Double-click to export		Opcode 0	Opcode 0			**lut_leakyrelu_0**
✓		⊟ **LUT_sigmoid_0**	LUT_sigmoid							
		nios_custom_instruction_slave	Custom Instruction Slave	Double-click to export		Opcode 1	Opcode 1			**lut_sigmoid_0**

Figure 3.12: Final System Interconnections

3.9.1 Top Level File Compilation

The compilation of the whole system was successful and the summary of the compilation is shown in figure 4.. The entire system uses 11% of the total logic elements, 6745 registers, 11% of the total pins , 1% of the total memory bits and one PLL.

The top-level file containing the SoPC system is illustrated in Figure 3.13



Flow Summary

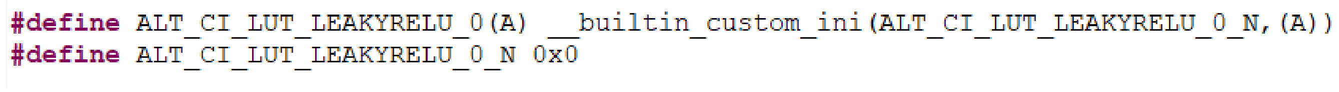
<<Filter>>

Flow Status	Successful - Mon May 20 23:39:19 2024
Quartus Prime Version	18.1.0 Build 625 09/12/2018 SJ Standard Edition
Revision Name	hw
Top-level Entity Name	hw
Family	Cyclone V
Device	5CSXFC6D6F31C6
Timing Models	Final
Logic utilization (in ALMs)	4,494 / 41,910 (11 %)
Total registers	6745
Total pins	56 / 499 (11 %)
Total virtual pins	0
Total block memory bits	66,305 / 5,662,720 (1 %)
Total DSP Blocks	5 / 112 (4 %)
Total HSSI RX PCSs	0 / 9 (0 %)
Total HSSI PMA RX Deserializers	0 / 9 (0 %)
Total HSSI TX PCSs	0 / 9 (0 %)
Total HSSI PMA TX Serializers	0 / 9 (0 %)
Total PLLs	1 / 15 (7 %)
Total DLLs	0 / 4 (0 %)

Figure 3.13: Compilation Report

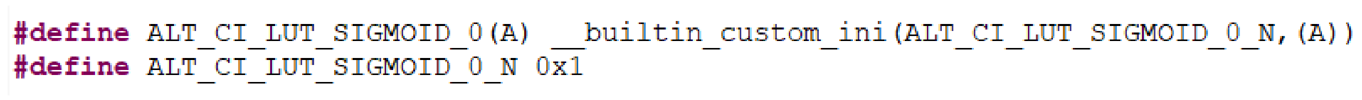
3.10 Custom Instructions Software Implementation

After integrating the custom instruction blocks and generating the HDL files, and subsequently compiling the project, the built-in functions and assembly MACROS for leaky-ReLU and sigmoid LUTs can be accessed from the header file ”system.h” within the BSP (Board Support Package) file of the project as shown in Figures 3.14 and 3.15



```
#define ALT_CI_LUT_LEAKYRELU_0(A) __builtin_custom_ini(ALT_CI_LUT_LEAKYRELU_0_N,(A))
#define ALT_CI_LUT_LEAKYRELU_0_N 0x0
```

Figure 3.14: Leaky-ReLu built-in Function and Assembly Macro



```
#define ALT_CI_LUT_SIGMOID_0(A) __builtin_custom_ini(ALT_CI_LUT_SIGMOID_0_N,(A))
#define ALT_CI_LUT_SIGMOID_0_N 0x1
```

Figure 3.15: Sigmoid built-in Function and Assembly Macro

3.11 Hexadecimal and Floating-Point Conversion Functions

After incorporating the built-in functions into the neural network code, we observed that our LUTs were encoded with hexadecimal values. However, the built-in functions required floating-point inputs. To address this mismatch, we added two conversion functions: one to convert floating-point values to hexadecimal before executing the built-in function, and another to convert the resulting hexadecimal values back to floating-point. This ensured seamless integration and accurate execution of the neural network operations.The functions are shown in Listings 3.3 and 3.4 .

```c
// Function to convert float to hexadecimal representation
uint32_t float_to_hex(float value) {

  uint32_t hex_value;
  memcpy(&hex_value, &value, sizeof(float));

  return hex_value;
}

// Function to convert hexadecimal representation to float
float hex_to_float(uint32_t hex_value) {

  float result;
  memcpy(&result, &hex_value, sizeof(float));

  return result;
}
```

Listing 3.3: FP to Hexadecimal Conversion

```c
// Function to convert hexadecimal representation to float
float hex_to_float(uint32_t hex_value) {
  float result;

  memcpy(&result, &hex_value, sizeof(float));
  return result;
}
```

Listing 3.4: Hexadecimal to FP Conversion

Listing 3.5 illustrates an example of how these two conversion functions are used.

```
printf("Hello from Nios II!\n");
int main() {

  float a = -1.43;

  float rl;

  // Convert float to hexadecimal
  uint32 tc float to hex(a);
  printf("Hexadecimal representation of a: %08x\n", c);

  // Call the custom instruction function with hexadecimal input
  uint32 t result = ALT_CI_LUT_LEAKYR_0 (c);

  // Convert hexadecimal result back to float
  rl hex to float (resultl);

  // Print the result
  printf("Result of custom instruction for a: %08x\n", result1);

  printf("r1 %f\n", r1);

}  return 0;
```

Listing 3.5: Example of Using Conversion Functions

3.12 Conclusion

In summary, this chapter has outlined the development and setup of the hardware system for our neural network application. It focuses on how we've integrated custom hardware blocks to make the neural network more efficient. Moving forward, we'll explore the details of these integrations and discuss their impact on enhancing the performance of our system.

Chapter 4

Results and Discussion

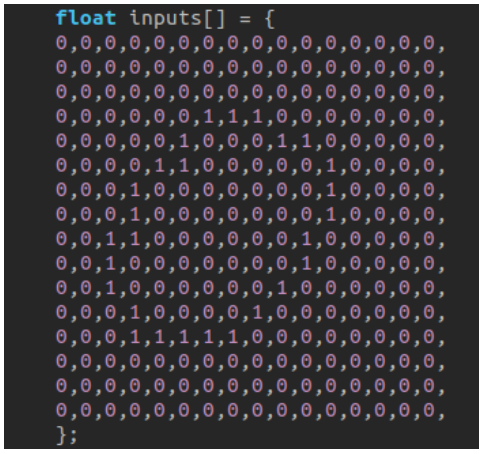
4.1 Introduction

In this chapter, we focus on the results obtained from hardware acceleration. This involves analyzing the impact of utilizing hardware resources to improve the model's performance, efficiency, and overall functionality. Our results reveal the significant benefits of hardware acceleration in optimizing the model's performance and contributing to a more efficient system.

4.2 NN Execution Using different Hardware Blocks

In this phase, we trained two neural networks with different configurations - one having 4 layers and the other having 11 layers. These networks were executed on the Nios-II processor. Afterwards, the Nios-II was enhanced with the FPH1 hardware block, and the execution of the networks was repeated. Finally, the networks were run with the FPH1 and activation functions LUTs.

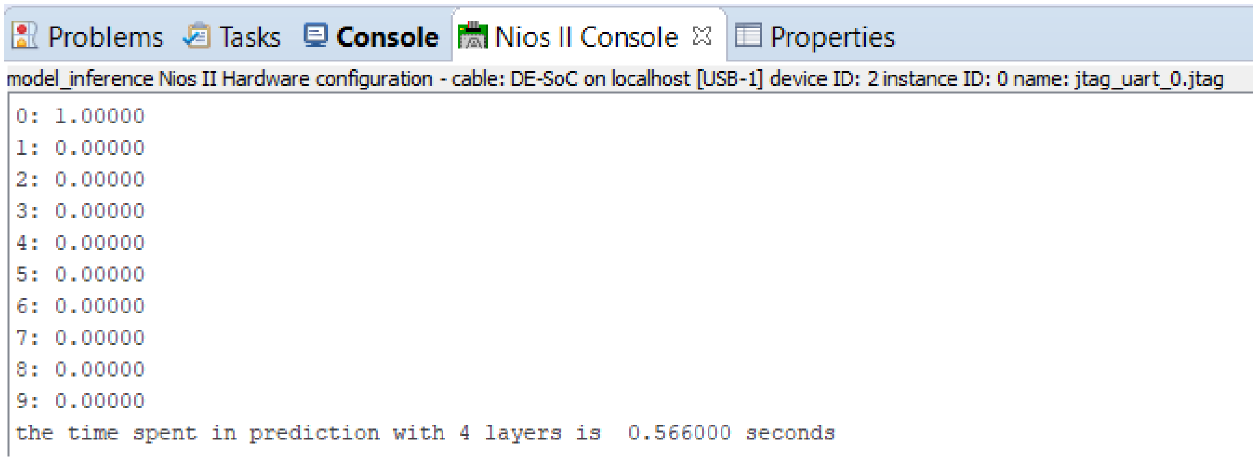
Every prediction in this chapter used an input array of size 256, representing a handwritten '0' digit as shown in Figure 4.1. To prioritize execution time over prediction result accuracy, the focus was primarily on the model's processing speed.



```
float inputs[] = {
0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,
0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,
0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,
0,0,0,0,0,0,1,1,1,0,0,0,0,0,0,0,
0,0,0,0,0,1,0,0,0,1,1,0,0,0,0,0,
0,0,0,0,1,1,0,0,0,0,0,1,0,0,0,0,
0,0,0,1,0,0,0,0,0,0,0,1,0,0,0,0,
0,0,0,1,0,0,0,0,0,0,0,1,0,0,0,0,
0,0,1,1,0,0,0,0,0,0,1,0,0,0,0,0,
0,0,1,0,0,0,0,0,0,0,1,0,0,0,0,0,
0,0,1,0,0,0,0,0,0,1,0,0,0,0,0,0,
0,0,0,1,0,0,0,0,1,0,0,0,0,0,0,0,
0,0,0,1,1,1,1,1,0,0,0,0,0,0,0,0,
0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,
0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,
0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,
};
```

Figure 4.1: Prediction Input Data

Performance results and predictions, of the 4-layer neural network on the Nios-II processor without any additional hardware blocks, are illustrated in Figure 4.2.

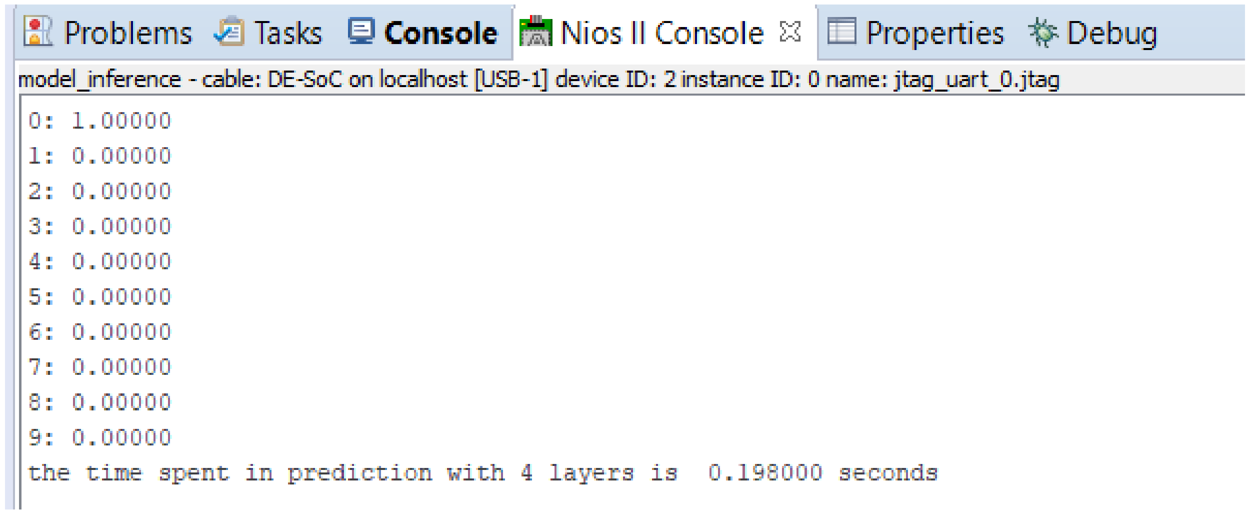


model_inference Nios II Hardware configuration - cable: DE-SoC on localhost [USB-1] device ID: 2 instance ID: 0 name: jtag_uart_0.jtag

```
0: 1.00000
1: 0.00000
2: 0.00000
3: 0.00000
4: 0.00000
5: 0.00000
6: 0.00000
7: 0.00000
8: 0.00000
9: 0.00000
the time spent in prediction with 4 layers is  0.566000 seconds
```

Figure 4.2: 4 Layers NN execution with Nios-II

When the same neural network was executed after incorporating the FPH1 custom instruction, the results and prediction times were shown in Figure 4.3.

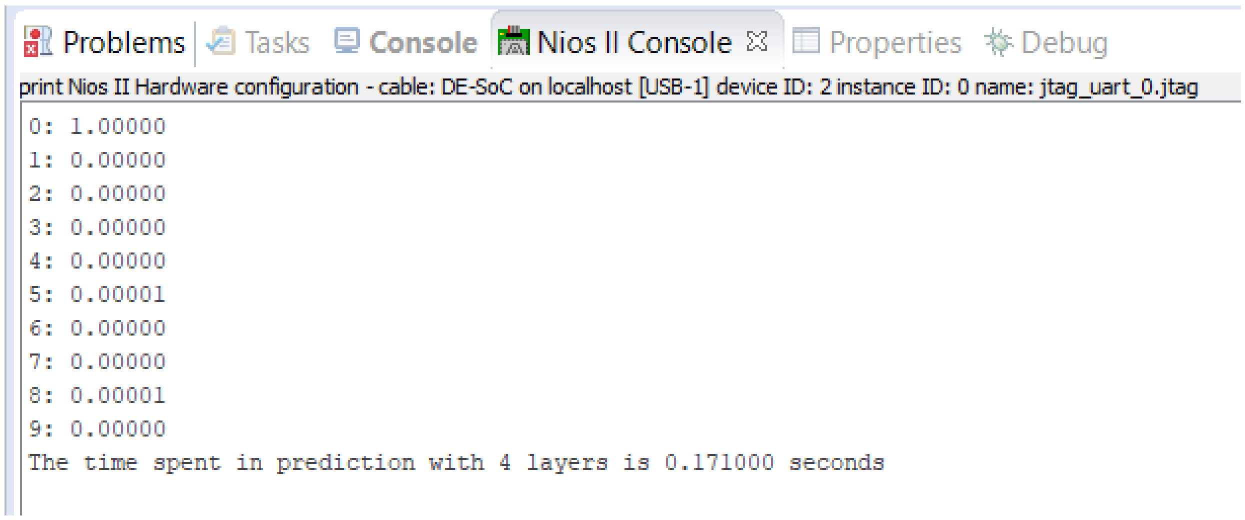


model_inference - cable: DE-SoC on localhost [USB-1] device ID: 2 instance ID: 0 name: jtag_uart_0.jtag

```
0: 1.00000
1: 0.00000
2: 0.00000
3: 0.00000
4: 0.00000
5: 0.00000
6: 0.00000
7: 0.00000
8: 0.00000
9: 0.00000
the time spent in prediction with 4 layers is  0.198000 seconds
```

Figure 4.3: 4 Layers NN execution with FPH1 Block

As expected; adding the FPH block reduces the operations' execution time by 65.02%.



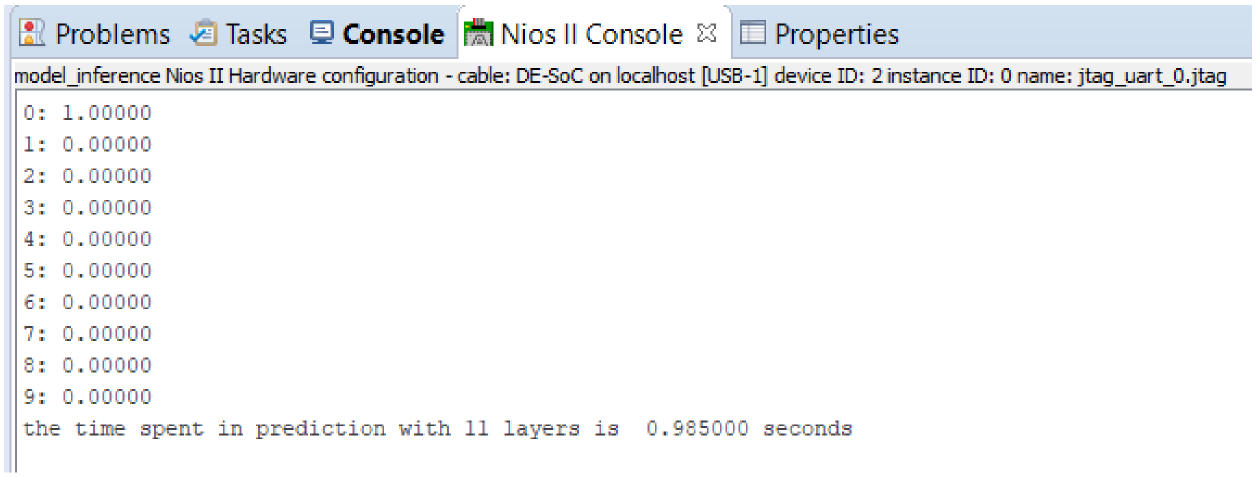
print Nios II Hardware configuration - cable: DE-SoC on localhost [USB-1] device ID: 2 instance ID: 0 name: jtag_uart_0.jtag

```
0: 1.00000
1: 0.00000
2: 0.00000
3: 0.00000
4: 0.00000
5: 0.00001
6: 0.00000
7: 0.00000
8: 0.00001
9: 0.00000
The time spent in prediction with 4 layers is 0.171000 seconds
```

Figure 4.4: 4 Layers NN execution with FPH1 and LUTs Custom Instructions

The final test for this neural network employed both FPH1 and LUTs custom instructions, with the results depicted in Figure 4.4. The LUTs enhanced the model's acceleration by 69.79%, since the CPU did not perform any calculations for the activation functions,and all operations were performed on hardware.

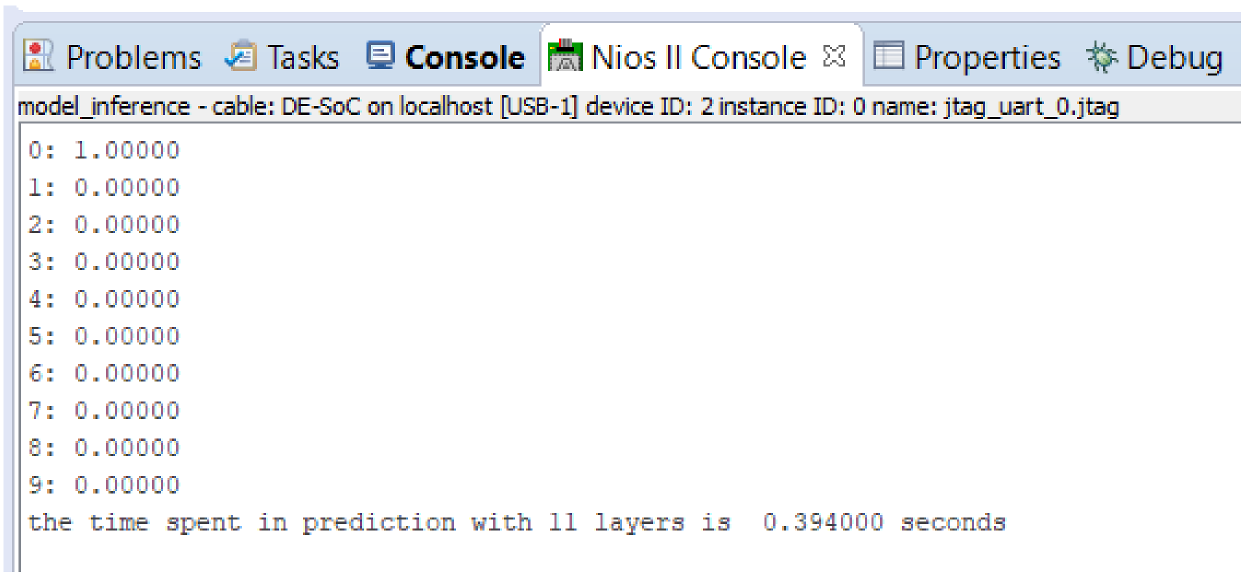
Repeating the same tests to the other NN resulted the following:



model_inference Nios II Hardware configuration - cable: DE-SoC on localhost [USB-1] device ID: 2 instance ID: 0 name: jtag_uart_0.jtag

```
0: 1.00000
1: 0.00000
2: 0.00000
3: 0.00000
4: 0.00000
5: 0.00000
6: 0.00000
7: 0.00000
8: 0.00000
9: 0.00000
the time spent in prediction with 11 layers is  0.985000 seconds
```

Figure 4.5: 11 Layers NN execution with Nios-II

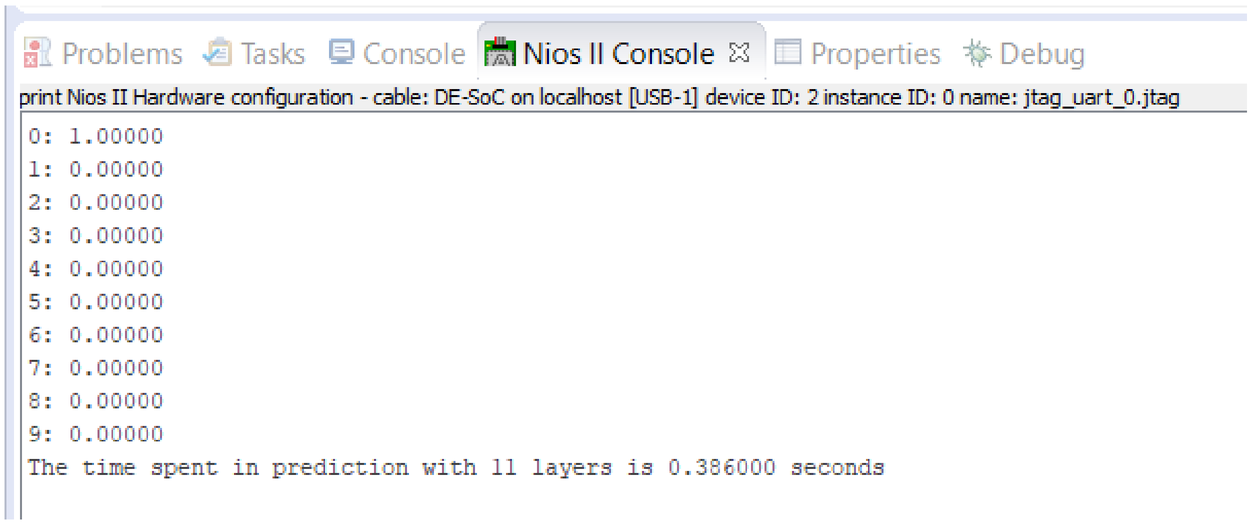


model_inference - cable: DE-SoC on localhost [USB-1] device ID: 2 instance ID: 0 name: jtag_uart_0.jtag

```
0: 1.00000
1: 0.00000
2: 0.00000
3: 0.00000
4: 0.00000
5: 0.00000
6: 0.00000
7: 0.00000
8: 0.00000
9: 0.00000
the time spent in prediction with 11 layers is  0.394000 seconds
```

Figure 4.6: 11 Layers NN execution with FPH1 Block

As shown in Figures 4.5 and 4.6, the implementation of the FPH1 block resulted in a significant performance improvement. The execution time was reduced from 0.985 seconds to 0.394 seconds, marking a 60% increase in speed. This optimization highlights the efficiency of using specialized hardware accelerators in neural network computations. By offloading intensive tasks to the FPH1, the overall processing time was greatly decreased, showcasing the potential for considerable enhancements in execution efficiency.



print Nios II Hardware configuration - cable: DE-SoC on localhost [USB-1] device ID: 2 instance ID: 0 name: jtag_uart_0.jtag

```
0: 1.00000
1: 0.00000
2: 0.00000
3: 0.00000
4: 0.00000
5: 0.00000
6: 0.00000
7: 0.00000
8: 0.00000
9: 0.00000
The time spent in prediction with 11 layers is 0.386000 seconds
```

Figure 4.7: 11 Layers NN execution with FPH1 and LUTs Custom Instructions

The LUTs had a minimal impact on execution time, reducing it by only 0.008 seconds compared to using the FPH1 alone, as shown in Figure 4.7.

In comparison to the first execution with 4 layers, the performance improvement in the 11-layer configuration was 4.77%. This indicates that while the LUTs provided a noticeable boost in the simpler, 4-layer network, their impact was less significant in the more complex, 11-layer network.

Both accelerations achieved for the 4-layer and 11-layer configurations were satisfactory, demonstrating the overall effectiveness of using hardware accelerators.However, the difference in results between the 4-layer and 11-layer executions raises the question of whether the number of layers affects the hardware acceleration of the neural network.

4.3 Acceleration versus the Number of Layers

In this section, we attempted to answer the previous question by generating graphs that depict execution times with and without the acceleration hardware blocks for various numbers of layers.

The process began with training 18 neural networks spanning from 3 to 20 layers, and the outcomes are showcased in Figure 4.8.

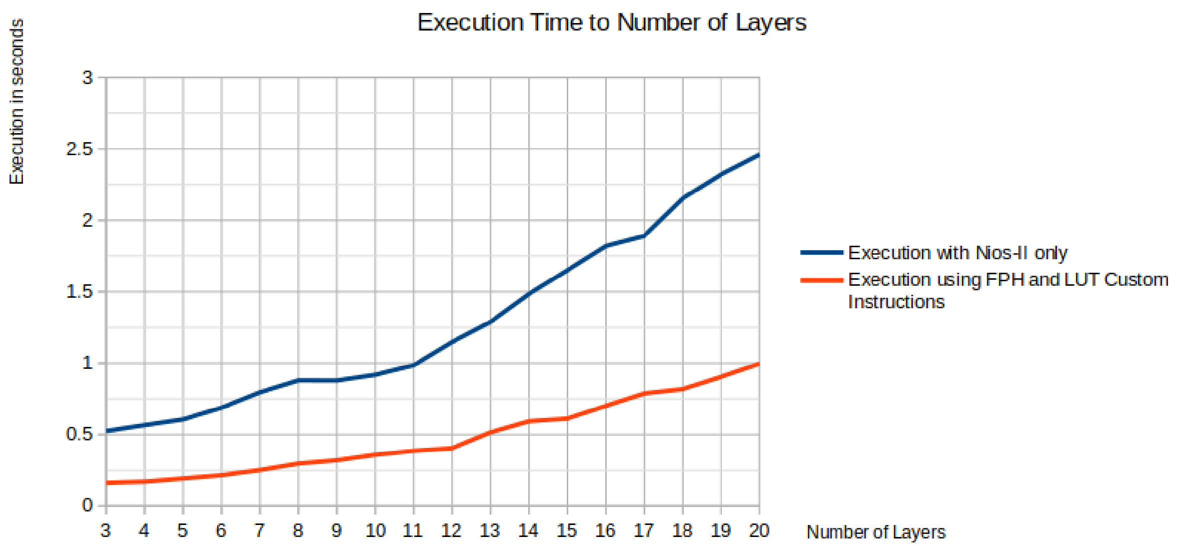


Figure 4.8: NNs of 3 to 20 layers acceleration

The graphs in Figure 4.8 indicate that the acceleration stays relatively consistent as the number of layers changes.

The average acceleration is 63.55%, with the best results around 69% achieved by networks with 3 and 4 layers. The lowest result was 58.3% in a 17-layer network, which is still an impressive outcome.

In order to have better comparison, We tried to train the NN with larger number of layers.However, we faced the issue of vanishing gradients, where the gradients needed for updating the network became extremely small or disappear as they are backpropagated from the output layers to the initial layers.

As a solution, we employed For-Loops in the previously trained NNs to increase the number of layers. The primary aim was to raise the number of floating-point operations and observe its effect on acceleration.

Figure 4.9 and 4.10 presents the acceleration results.

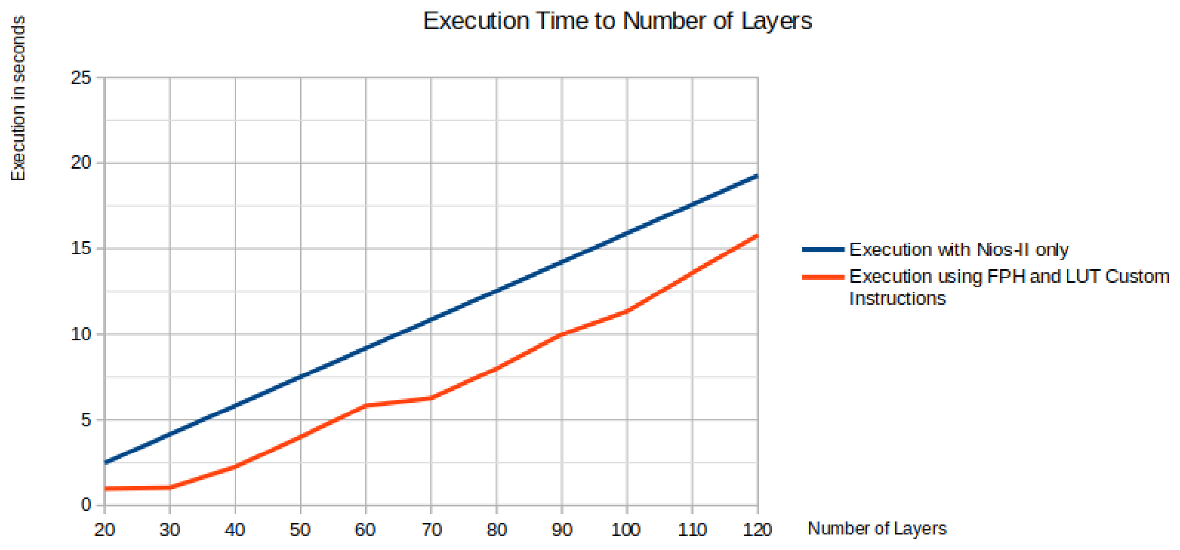


Figure 4.9: NNs of 20 to 120 layers acceleration

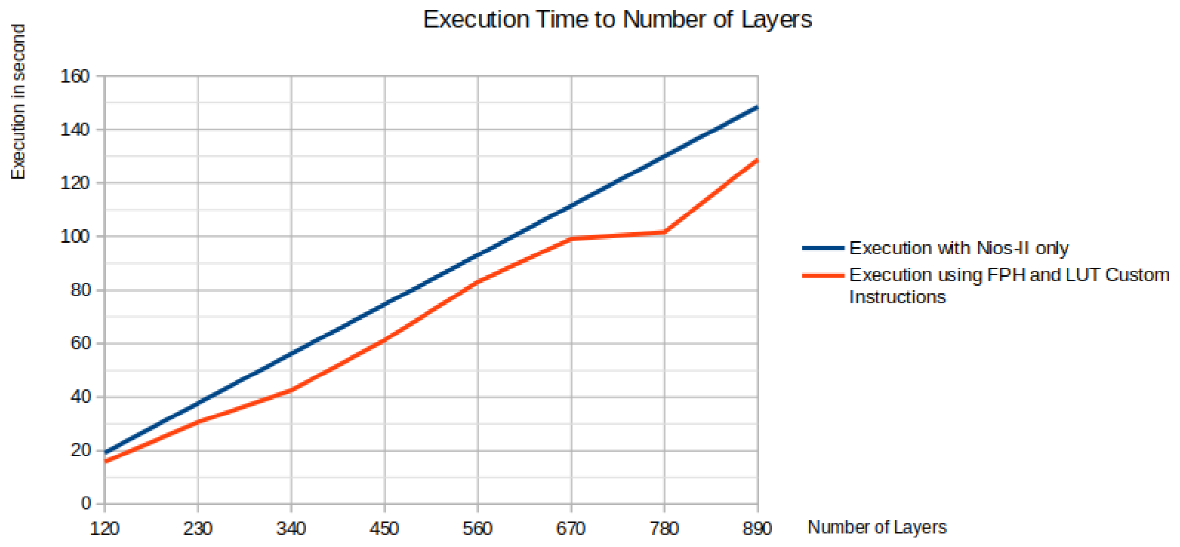


Figure 4.10: NNs of 120 to 890 layers acceleration

Referring to Figure 4.9 it is evident that the accelerated and original execution graphs are closer together compared to Figure 4.8. The average acceleration was 40.72%, which is 22% lower than the first set of tests. However, this set included the highest acceleration observed in our experiment, with 74.61% acceleration at 30 layers.

For Figure 4.10 , the difference between the execution times of NIOS-II and NIOS-II with custom instructions is minimal. The largest gap was observed in the neural network with 340 layers, achieving a 24.34% acceleration, while the neural network with 560 layers showed only a 10.75% improvement.

Comparing the average speedup, which is 17% for these networks, it was relatively low, likely due to the increased number of software functions used inside the loop, reducing the impact of our system.

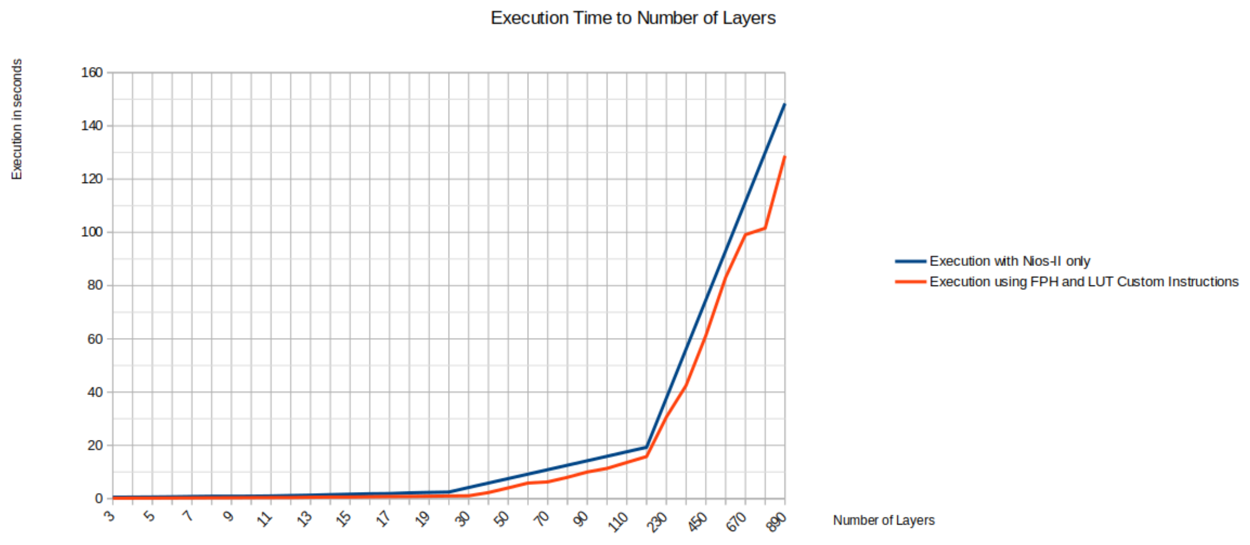


Figure 4.11: Overall Acceleration graph

Figure 4.11 combines all the previous results into a single graph.

Despite the decline in acceleration performance with the addition of more layers to the neural network, the system remains practical for several reasons:

- For embedded AI applications, the models typically employed are not overly large or complex. This is partly due to considerations such as energy consumption, available storage capacity, and the straightforward nature of their applications.
- Even when utilizing a larger model, the system's acceleration capabilities remain effective. This is evident in the significant reduction in execution time, as demonstrated by the 890-layer model, where execution time was decreased by 20 seconds.

4.4 Conclusion

At the conclusion of this chapter, having gathered all the results from our tests, we affirm that the accelerators we designed are beneficial for executing neural networks on embedded systems.

General Conclusion

In conclusion, our exploration into hardware acceleration for neural network execution on embedded systems has yielded promising results and valuable insights. Through the design and implementation of custom hardware accelerators, specifically engineered to operate within the constraints of embedded environments, We've effectively handled the unique challenges of deploying complex AI algorithms in resource constrained settings.

The primary objective of our project was to enhance the speed and efficiency of AI models on embedded systems, and our findings demonstrate significant progress towards this goal. By including FPGA technology and custom instructions for the Nios II processor, we have achieved notable reductions in inference latency and execution time, Opening the door for real-time processing of AI tasks in critical applications.

Our study highlights the effectiveness of hardware acceleration in optimizing neural network performance while minimizing resource usage. Through careful design and customization of hardware blocks,we achieved significant improvements in execution efficiency despite the constraints of embedded systems. This underscores the importance of tailored solutions for AI applications, where customized hardware architecture can lead to significant performance enhancements compared to traditional software-based approaches.

Furthermore, our investigation into the impact of network complexity on hardware acceleration has provided valuable insights into the scalability of our approach. As the network becomes more complex, the performance improvements decrease, our accelerators remain practical and effective for a wide range of embedded AI applications.

Our findings suggest exciting possibilities for future research in hardware acceleration for embedded AI. We could explore additional optimization techniques like model quantization or hardware-aware training algorithms to improve efficiency and scalability further.

As we look back on our research, it's crucial to recognize certain constraints. Although our study has given us valuable insights and shown encouraging outcomes, there are areas where we could delve deeper. One limitation is the focus on a specific type of neural network and a single FPGA board, which may not capture the full range of possibilities or constraints in different contexts. Additionally, our investigation into the impact of network complexity on hardware acceleration only scratched the surface, and deeper analysis could reveal more detailed relationships.

In conclusion, our study reaffirms the potential of hardware acceleration as a practical solution for deploying AI algorithms on embedded systems. By utilizing the power of custom hardware accelerators, we have demonstrated significant improvements in execution speed, efficiency, and scalability, laying the foundation for the next generation of intelligent embedded devices.

Future Work

In this chapter, we explore potential avenues for further research and development based on the findings presented in the preceding chapters. Although our designed System has reduced execution time for NN ,several improvements can be done. As we look ahead, we consider the following areas for future exploration:

- **Optimization and Compression:** One critical area for future exploration involves optimizing and compressing the neural network model. While this work primarily focuses on hardware development, it does not delve into techniques for reducing resource usage. Investigating methods such as weight quantization, pruning, and sparsity can lead to more efficient models. By compressing the model weights, we can achieve better performance in terms of memory footprint and computational efficiency.
- **Generalization to Other Models:** The current testing and results comparison are limited to a specific model—the handwritten digits classification deep neural network (DNN). However, to validate the broader applicability of our approach, we need to extend our evaluation to other neural network architectures. Different models may exhibit varying behaviors, especially when faced with diverse input datasets. Therefore, exploring additional model types and assessing their performance is essential for robustness and generalization.
- **Implementation on Alternative FPGA Platforms:** While our work focuses on the development of neural network accelerators for a specific FPGA board, it is essential to explore the feasibility of porting these accelerators to other FPGA platforms. Different FPGA families have varying architectures, resource availability, and performance characteristics. By adapting our designs to alternative boards, we can assess their scalability and versatility.

By incorporating the proposed future work, we will anticipate broader coverage of use cases, and move closer to realizing efficient and versatile neural network accelerators for embedded systems.

Bibliography

[1] Stuart J. Russell and Peter Norvig. *Artificial Intelligence A Modern Approach*. Prentice Hall, Englewood Cliffs, New Jersey, 1995.

[2] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. `http://www.deeplearningbook.org`. MIT Press, 2016.

[3] MICHAEL NEGNEVITSKY. *Artificial Intelligence A Guide to Intelligent Systems*. 2nd ed. Pearson Education Limited,Edinburgh Gate,England, 2005.

[4] Andrew L. Maas, Awni Y. Hannun, and Andrew Y. Ng. "Rectifier Nonlinearities Improve Neural Network Acoustic Models". In: (2013), pp. 2–3.

[5] YONGBIN YU 1 et al. "RMAF: Relu-Memristor-Like Activation Function for Deep Learning". In: (2020).

[6] Carl Hamacher et al. *Computer Organization and Embedded Systems*. sixth edition. McGraw-Hill, 2012.

[7] Sadman Sakib Khan. "Understanding Embedded AI (EAI): What it is and How it Works". In: (2023). https://polygontechnology.io/embedded-ai-explained/.

[8] Nadeski. *Bringing Machine Learning to Embedded Systems.* Dallas, Texas: Texas Instruments, 2019.

[9] Z. Zhang and J. Li. "A Review of Artificial Intelligence in Embedded Systems". In: *Micromachines* (2023). https://doi.org/10.3390/mi14050897.

[10] Clive Maxfield. *The Design Warrior's Guide to FPGAs*. Elsevier,Newnes, 2004.

[11] Nazeih Botros. . *HDL with Digital Design: VHDL and Verilog*. David Pallai MERCURY LEARNING and INFORMATION, 2015.

[12] *Cyclone® V Device Overview*. 2018.

[13] *Altera Cyclone FPGA families comparison table*. URL: `https://akpc806a.wordpress.com/2016/10/27/altera-cyclone-fpga-families-comparison-table/`.

[14] *DE10-Standard User Manual*. 2018.

[15] *Intel® Quartus® Prime Standard Edition User Guide*. 2019.

[16] *Platform Designer User Guide*. 2018.

[17] *Nios® II Processor Reference Guide*. 2023.

[18] *Nios II Custom Instruction User Guide*. 2020.

[19] *Neural Network*. URL: `https://github.com/synthintai/nn` (visited on 06/03/2024).