ISTANBUL TECHNICAL UNIVERSITY ★ GRADUATE SCHOOL

ARTIFICIAL NEURAL NETWORK BASED ELECTRICAL MACHINE FAULT CLASSIFICATION ON FPGA

M.Sc. THESIS

Mert Yaşar AYDIN

Department of Electronics and Communication Engineering

Electronics Engineering Programme

DECEMBER 2024

ISTANBUL TECHNICAL UNIVERSITY ★ GRADUATE SCHOOL

ARTIFICIAL NEURAL NETWORK BASED ELECTRICAL MACHINE FAULT CLASSIFICATION ON FPGA

M.Sc. THESIS

Mert Yaşar AYDIN

(504211216)

Department of Electronics and Communication Engineering

Electronics Engineering Programme

Thesis Advisor: Prof. Dr. Sıddıka Berna ÖRS YALÇIN

DECEMBER 2024

İSTANBUL TEKNİK ÜNİVERSİTESİ ★ FEN BİLİMLERİ ENSTİTÜSÜ

FPGA ÜZERİNDE YAPAY SİNİR AĞI TABANLI ELEKTRIK MAKİNESİ ARIZA SINIFLANDIRMASI

YÜKSEK LİSANS TEZİ

Mert Yaşar AYDIN

(504211216)

Elektronik ve Haberleşme Mühendisliği Anabilim Dalı

Elektronik Mühendisliği Programı

Tez Danışmanı : Prof. Dr. Sıddıka Berna ÖRS YALÇIN

DECEMBER 2024

Mert Yaşar AYDIN, a **M.Sc.** student of **ITU Graduate School of Science Engineering and Technology** student ID **504211216,** successfully defended the thesis entitled **"ARTIFICIAL NEUREAL NETWORK BASED ELECTRICAL MACHINE FAULT CLASSIFICATION ON FPGA",** which he prepared after fulfilling the requirements specified in the associated legislations, before the jury whose signatures are below.

Thesis Advisor:	**Prof. Dr. Sıddıka Berna ÖRS YALÇIN** Istanbul Technical University	**…….**
Jury Members:	**Prof. Dr. Serdar İPLİKÇİ** Pamukkale University	**……**
	Asst. Prof Dr. İlkay ÖKSÜZ Istanbul Technical University	**……**

Date of Submission : 23.11.2024

Date of Defense : 17.12.2024

FOREWORD

When I was a child, I had dreams about becoming a scientist and inventing things that will ease the human life and advancing technology. I believe his thesis makes my dreams real as it contains unmentioned, untouched areas in the technology.

Thanks to Artificial Intelligence, engineers find new fields of application of their knowledge. Because it provides a different solution method for the problems that we have to deal. In this thesis, I found opportunity observe the power of the artificial intelligence at first hand. I saw that the tasks that are hard to solve with classic methods become really easy when the AI is applied. Also, I found opportunity master on VHDL language thanks to this study and my venerable advisor Prof. Dr. Sıddıka Berna ÖRS YALÇIN.

Additionally, I would like to thank my institution that I work in to support me both financially and morally.

And, I would like to thank Onur ŞAHBAZ and Yusuf Engin TETİK who are my colleagues also my team leader and group leader to believe and support me to succeed in this study.

And, I would like to thank my family who are my mother Emine AYDIN, Father Yasin AYDIN and my brothers Furkan Deniz AYDIN and Batuhan Cuma AYDIN to become a great family.

And finally, I would like to extend my most special thanks to İmran Hanım, the kindest person in the world, who has been like a mother to me, providing both financial and emotional support and standing by me during my most difficult times for the past three years.

December 2024 Mert Yaşar AYDIN

(Electrical and Electronic Engineer)

TABLE OF CONTENTS

Page

ABBREVIATIONS

1D : One-Dimensional
CNN : Convolutional Neural Network
AI : Artificial Intelligence
ANN : Artificial Neural Network
SVM : Support Vector Machine
FPGA : Field Programmable Gate Array
VHDL : VHSIC Hardware Description Language
FFT : Fast Fourier Transform
MB : Mega Byte
DSP : Digital Signal Processor
MFS : Machinery Fault Simulator
ABVT : Alignment-Balance-Vibration
MAFAULDA: Machinery Fault Database
CSV : Comma-Separated Values
DMA : Direct Memory Access
CDMA : Central Direct Memory Access
FIFO : Firs In First Out
IP : Intellectual Property
AXI : Advanced eXtensible Interface
RAM : Random Access Memory
DDR : Double Data Rate
FF : Flip Flop
LUT : Look Up Table

SYMBOLS

W **:** Watt
J **:** Joule

LIST OF TABLES

Page

LIST OF FIGURES

Page

ARTIFICIAL NEUREAL NETWORK BASED ELECTRICAL MACHINE FAULT CLASSIFICATION ON FPGA

SUMMARY

Electrical machines and drives play a critical role in modern society, spanning industrial equipment to renewable energy production. The reliability and efficiency of these systems depend on continuous performance monitoring and the early detection of potential issues. Condition monitoring systems are essential for evaluating the performance of machines and detecting potential faults early, thereby reducing unexpected downtimes and maintenance costs.

Traditional fault detection methods include vibration analysis, thermography, and oil analysis. Vibration analysis is used to identify mechanical issues, while thermography can detect overheating that may indicate electrical or mechanical problems. The integration of artificial intelligence (AI) and machine learning techniques has significantly enhanced fault detection capabilities. AI techniques such as artificial neural networks (ANN) and support vector machines (SVM) can process large datasets to predict potential faults in advance.

This thesis aims to develop and implement an ANN-based fault diagnosis system for electrical machines on an FPGA platform. The system will leverage the high-speed processing and parallel computing capabilities of FPGAs to achieve real-time fault detection and diagnosis. Additionally, this method will be tested with different datasets to evaluate its generalizability. The thesis comprises two main sections: deep learning and FPGA applications. The first section describes the dataset used, data processing, and the development of the CNN architecture. The second section discusses the implementation of the CNN model on FPGA using VHDL. Experimental results are presented in the final section.

In the deep learning section, a CNN-based fault detection model has been developed for ensuring the reliable operation of electrical machines. The dataset used consists of vibration signals obtained from the MAFAULDA database. The data were used to train and test the CNN model, with performance evaluated based on accuracy, precision, recall, and F1 score. The ability of CNNs to learn local patterns and features makes them particularly effective for fault detection.

The FPGA application section covers the implementation of the CNN-based fault detection system on FPGA. Convolution_1D, Convolution_1D_Middle, Convolution_1D_No_MP, and Dense layers were developed using VHDL. Real-time fault detection was performed using the Nexys A7 development board. The parallel processing capabilities of the FPGA allowed for high-speed computations, making the system suitable for real-time applications. This implementation efficiently handles large volumes of data, ensures low latency, and maintains high accuracy.

When developing a CNN-based fault detection system using VHDL, optimizing the layers and processes is crucial. The Convolution_1D, Convolution_1D_No_MP, and

Convolution_1D_Middle layers perform 1D convolution operations on the input data using predefined kernels. The Convolution_1D layer extracts fundamental features from raw sensor data, while the Convolution_1D_No_MP layer maintains higher resolution for detailed analysis. The Convolution_1D_Middle layer further refines the features, ensuring high accuracy in fault detection. The Dense layer processes these features into the final classification result. The TOP_CNN module integrates all layers, managing the data flow and producing the final fault classification result.

In conclusion, a CNN-based fault detection system has been developed and implemented on FPGA. The system significantly enhances the reliability and efficiency of electrical machines. Future research will involve testing this method with broader datasets and different application areas to further evaluate and improve its effectiveness. This work contributes to the implementation of proactive maintenance strategies in industrial processes, ensuring operational excellence of machinery.

FPGA'DA YAPAY SİNİR AĞ TABANLI ELEKTRİKLİ MAKİNA ARIZA SINIFLANDIRMASI

ÖZET

Elektrik makineleri ve sürücüleri, modern teknolojinin temel yapı taşlarından biri olarak endüstriyel üretim, enerji dönüşümü ve otomasyon gibi çok çeşitli alanlarda vazgeçilmez bir rol oynamaktadır. Bu sistemler, enerjinin mekanik harekete dönüştürülmesi ve kontrol edilmesi gibi hayati işlevleri yerine getirirken, üretim süreçlerinin verimliliği ve güvenilirliği açısından büyük bir öneme sahiptir. Özellikle yenilenebilir enerji kaynaklarının kullanımı arttıkça, elektrik makinelerinin performansı ve dayanıklılığı, hem ekonomik hem de çevresel sürdürülebilirlik açısından kritik hale gelmiştir. Bu bağlamda, elektrik makinelerinin güvenilir çalışması ve arızalarının önlenmesi, endüstriyel operasyonların kesintisiz devam etmesi için kaçınılmaz bir gerekliliktir.

Elektrik makinelerinde meydana gelebilecek arızalar, sadece makinelerin çalışma ömrünü kısaltmakla kalmaz, aynı zamanda üretim süreçlerini kesintiye uğratarak ciddi ekonomik kayıplara yol açabilir. Özellikle ani arızalar, duruş sürelerinin uzamasına neden olabilir ve bu durum hem maliyetli onarımlar gerektirir hem de operasyonel riskleri artırır. Bu nedenle, elektrik makinelerinin durumu hakkında sürekli bilgi sağlayan, performanslarını izleyen ve olası sorunları henüz ortaya çıkmadan tespit eden sistemlerin geliştirilmesi büyük bir önem taşır. Bu tür durum izleme sistemleri, makinelerin genel sağlığını analiz ederek proaktif bakım planlaması yapılmasına olanak tanır. Bu da hem bakım maliyetlerinin azaltılmasını sağlar hem de makinelerin operasyonel sürekliliğini destekler.

Durum izleme sistemleri, makinelerin çalışma durumlarını sürekli izleyerek performanslarını değerlendiren gelişmiş teknolojik altyapılardır. Bu sistemler, titreşim analizi, termografi ve yağ analizi gibi geleneksel yöntemlerden yararlanır. Titreşim analizi, elektrik makinelerinde mekanik problemleri tespit etmek için en yaygın kullanılan yöntemlerden biridir. Bu yöntem, rulman aşınmaları, hizalama hataları veya dengesizlik gibi mekanik sorunların erken teşhisinde oldukça etkilidir. Termografi ise, makinelerde aşırı ısınma veya anormal sıcaklık değişimleri gibi elektriksel ve mekanik problemleri tespit etmek için kullanılan bir diğer popüler tekniktir. Bununla birlikte, yağ analizi, makine bileşenlerinin aşınma durumu ve yağ kalitesinin değerlendirilmesi açısından değerli bilgiler sunar. Ancak, bu geleneksel yöntemlerin veri işleme kapasitesi ve hassasiyeti sınırlıdır; dolayısıyla karmaşık arıza durumlarında veya büyük veri setlerinin analizinde yetersiz kalabilirler.

Son yıllarda, yapay zeka (AI) ve makine öğrenimi (ML) teknikleri, durum izleme ve arıza tespit sistemlerinde yeni bir dönemin kapılarını açmıştır. Yapay sinir ağları (ANN) ve destek vektör makineleri (SVM) gibi ileri düzey algoritmalar, büyük veri setlerini analiz ederek makinelerin çalışma durumlarını daha hassas bir şekilde değerlendirebilmekte ve potansiyel arızaları henüz ortaya çıkmadan tahmin edebilmektedir. Özellikle, AI tabanlı sistemlerin hız, doğruluk ve otomasyon

yetenekleri, geleneksel yöntemlere kıyasla çok daha üstün bir performans sunar. Bu sistemler, geniş veri setlerini işleyerek, makinelerde meydana gelebilecek arızaları belirleme ve sınıflandırma sürecinde benzersiz bir esneklik ve doğruluk sağlar. Ayrıca, gerçek zamanlı analiz yetenekleri sayesinde, arızaların tespiti ve giderilmesi süreci hızlandırılarak, endüstriyel süreçlerin verimliliği artırılır.

Bu tez, yapay sinir ağı (ANN) tabanlı bir arıza teşhis sisteminin FPGA (Field Programmable Gate Array) üzerinde geliştirilmesini ve uygulanmasını hedeflemektedir. FPGA teknolojisi, yüksek hızda veri işleme ve paralel hesaplama yetenekleriyle gerçek zamanlı uygulamalar için ideal bir platform olarak öne çıkmaktadır. FPGA'ların düşük güç tüketimi, ölçeklenebilirlik ve esneklik gibi özellikleri, yapay zeka tabanlı sistemlerin donanım platformlarında uygulanabilirliğini artırmaktadır. Bu çalışmada geliştirilen sistem, elektrik makinelerindeki farklı arıza türlerini tespit etmek ve sınıflandırmak için optimize edilmiş bir yapıya sahiptir. Sistem, yüksek doğruluk oranları ve düşük gecikme süreleriyle özellikle gerçek zamanlı uygulamalar için etkili bir çözüm sunmaktadır.

Tez çalışması iki ana bölümden oluşmaktadır: derin öğrenme geliştirme ve FPGA uygulama. İlk bölümde, elektrik makinelerindeki arızaların teşhisi için bir Convolutional Neural Network (CNN) modeli geliştirilmiştir. CNN modeli, MAFAULDA veri tabanından alınan titreşim sinyallerini kullanarak eğitilmiş ve test edilmiştir. Veri seti, normalizasyon ve Hızlı Fourier Dönüşümü (FFT) gibi ön işleme teknikleriyle optimize edilmiştir. Model, normal çalışma, hizalama hatası ve dengesizlik gibi farklı arıza kategorileri arasında başarılı bir şekilde sınıflandırma yapabilmiştir. Modelin performansı, doğruluk, kesinlik, geri çağırma ve F1 skoru gibi metriklerle değerlendirilmiş ve %96.5 doğruluk oranı elde edilmiştir. Bu sonuçlar, modelin yüksek doğruluk ve genelleme kapasitesini kanıtlamaktadır.

FPGA uygulama bölümü, CNN modelinin donanım üzerinde uygulanmasına odaklanmıştır. Nexys A7 geliştirme kartı kullanılarak, CNN modeline ait farklı katmanlar VHDL ile tasarlanmıştır. Bu katmanlar arasında Convolution_1D, Convolution_1D_No_MP, Convolution_1D_Middle ve Dense yer almaktadır. Convolution_1D katmanı, temel özellikleri çıkarırken; Convolution_1D_No_MP ve Convolution_1D_Middle katmanları, daha ayrıntılı analizler için optimize edilmiştir. Dense katmanı, modelin son çıktısını üreterek sınıflandırma işlemini tamamlamaktadır. FPGA üzerinde yapılan bu uygulama, yüksek işlem hızları ve düşük gecikme süreleri sayesinde gerçek zamanlı arıza tespitini mümkün kılmaktadır. Ayrıca, sistemin paralel işlem yetenekleri, büyük veri hacimlerinin hızlı bir şekilde işlenmesini sağlamaktadır.

Bu tez çalışmasında, geliştirilen yapay sinir ağı tabanlı sistemin deneysel sonuçları, hem simülasyon hem de donanım testleri ile doğrulanmıştır. FPGA üzerinde uygulanan sistem, yüksek doğruluk oranları ve hızlı veri işleme yetenekleri ile elektrik makinelerinde arıza tespiti için etkili bir çözüm olduğunu göstermiştir. Sistem, sadece elektrik makinelerinin güvenilirliğini artırmakla kalmamış, aynı zamanda bakım süreçlerini optimize ederek ekonomik kazanç sağlamıştır.

Sonuç olarak, bu tez, yapay zeka ve FPGA teknolojilerinin bir araya getirilmesiyle, elektrik makinelerinde arıza tespiti için yenilikçi bir çözüm sunmaktadır. Geliştirilen sistem, makinelerin verimliliğini artırmak ve arıza risklerini azaltmak için etkili bir araçtır. Gelecekte, bu yöntem daha geniş veri setleri ve farklı endüstriyel uygulamalar üzerinde test edilerek daha da geliştirilebilir. Ayrıca, bu yaklaşımın robotik,

yenilenebilir enerji ve akıllı şehir sistemleri gibi alanlarda uygulanabilirliği araştırılarak daha geniş bir etki alanı oluşturulabilir. Bu çalışma, endüstriyel bakım süreçlerinde proaktif stratejilerin benimsenmesine katkıda bulunarak, makinelerin operasyonel mükemmeliyetini sağlamaya yönelik önemli bir adım atmıştır.

1. INTRODUCTION

Electrical machines and drives play a pivotal role in our contemporary world, powering the technological innovations that have significantly influenced our society. These systems are crucial in a multitude of applications, including industrial equipment, transportation, and renewable energy production. They lead the pursuit of sustainability and efficiency, profoundly transforming various industries. The significance of electrical machines and drives in today's society is clear, as their effects are evident across many fields. Condition monitoring of electrical machines is crucial for the reliability and efficiency of modern industrial systems. These machines are critical components of production processes, and their failures can lead to significant costs and production losses. Condition monitoring continuously tracks the performance of machines, allowing potential problems to be detected early. This helps minimize unplanned downtime and manage maintenance activities more effectively.

Techniques such as vibration analysis, thermography, and oil analysis are particularly useful for assessing the condition of electrical machines and identifying signs of failure at early stages. Condition monitoring not only aims to prevent failures but also offers significant advantages in terms of increasing energy efficiency and extending the lifespan of machines. With this approach, maintenance costs are reduced, and machines can operate at optimal performance.

Condition monitoring provides a competitive advantage for industrial enterprises. By preventing unexpected machine failures, production processes can continue without interruption. "Condition monitoring makes significant contributions to efficiency and reliability by monitoring the operational status of machines." Additionally, when these systems are integrated with data analytics and artificial intelligence, proactive maintenance strategies can be developed, ensuring even higher levels of operational excellence. Therefore, it is crucial to develop and improve the current condition monitoring systems. Especially, Electrical machine fault detection and classification task is important for industrial applications.

In this study, Development and implementation of CNN based electrical machine fault detection method is introduced. For that purpose, proposed 1D CNN model will be trained with the data that is obtained from a mechanical fault simulator on the python environment by using “KERAS” library. The physical quantity that is used for this application is vibration signals. This dataset is available as online and it is called “MAFAULDA”. After the training and testing of the CNN model, weights of the model is extracted from model as 32-bit Floating number kernels. Then, this model will be implemented on FPGA for realization of condition monitoring purposes.

1.1 Literature Review

1.1.1 Electrical machine fault detection

Fault detection in electrical machines is critical for enhancing reliability and efficiency in industrial applications. Advances in this field have gained significant momentum with the use of machine learning and deep learning techniques. This literature review comprehensively examines the application of these techniques and current trends in this domain.

Peesapati et al. [1] conducted a literature review focused on advanced methods for condition monitoring and early diagnosis in wind turbines. The study elaborates on advanced monitoring techniques aimed at continuously tracking the operational status of turbine components to intervene before failures occur. Integration of data analysis and machine learning techniques plays a crucial role in enhancing the efficiency and lifespan of turbines.

Carvalho et al. [2] presented a comprehensive review on autonomous fault detection in electrical machines using deep learning techniques. This study highlights methods that improve the accuracy of fault detection and diagnosis through deep learning models. Various deep learning architectures and their optimization for practical applications are discussed.

Singh and Yadav [3] explored the application of machine learning techniques in predictive maintenance. The study compares the performance of different machine learning algorithms to determine the most effective methods for industrial applications. Additionally, challenges in implementing these techniques and strategies to overcome them are addressed.

Kim et al. [4] provided an in-depth review of fault diagnosis in electrical machines using deep learning-based methods. The study emphasizes the effectiveness of deep learning techniques such as neural networks and convolutional neural networks (CNNs) in fault diagnosis. Training deep learning models with large datasets and their application in real-time scenarios are discussed

Kudelina et al. [5] examined methods for condition monitoring and fault detection in electrical machines. This study discusses various sensor technologies and data processing techniques to enhance machine performance and reliability. The role of data analysis and machine learning algorithms in this field is highlighted.

Akbar et al. [6] reviewed advanced techniques used for fault diagnosis in electrical machines. The study focuses on the application of machine learning and deep learning algorithms in this domain. Optimized models for different fault types and their performance in industrial applications are discussed.

Gultekin and Bazzi [7] investigated fault detection and diagnosis techniques for AC motor drives. The study highlights advancements achieved through the integration of machine learning and signal processing techniques. Optimization and performance enhancement of these techniques for practical applications are examined.

Magar et al. [8] conducted a comprehensive study on deep learning-based bearing fault classification. The study examines the effectiveness and accuracy of deep learning models in detecting and classifying bearing faults. The performance of the FaultNet model is particularly emphasized.

Tang et al. [9] introduced Omni-Scale CNN models for time series classification. This study focuses on improving the accuracy and speed of time series data classification using different kernel sizes. Omni-Scale CNNs demonstrate high performance when working with large datasets.

Singh and Gupta [10] reviewed recent advancements in data-driven fault diagnosis for electrical machines. This study emphasizes the role of machine learning and big data analytics in fault diagnosis. Practical applications and performance evaluation of these techniques in industrial settings are discussed.

Palit [24] presents an FPGA-based fault detection and classification system developed for the defense industry, focusing on an electromechanical system operating under various health conditions. Vibration and current data were collected and analyzed

using machine learning models, particularly Support Vector Machines (SVM). The most successful model was implemented on an FPGA platform, achieving a classification accuracy of 98.076% across six distinct conditions. This study demonstrates the potential of FPGA-based systems for real-time performance and high-accuracy fault detection in complex environments.

1.1.2 1D Convolutional neural networks

Fault detection in electrical machines is critical for enhancing reliability and efficiency in industrial applications. Advances in this field have gained significant momentum with the use of machine learning and deep learning techniques, particularly 1D Convolutional Neural Networks (CNNs). These networks are well-suited for processing time-series data, which is commonly used in fault diagnosis. This literature review comprehensively examines the application of 1D CNNs in fault detection and diagnosis, highlighting current trends and innovations in this domain.

Taherkhani et al. [11] presented a study on a deep convolutional neural network (CNN) designed for time series classification with intermediate targets. This approach significantly improves classification accuracy by incorporating intermediate learning targets, which enhance the training process and overall model performance. By focusing on intermediate targets, the model can better capture the nuances in the data, leading to more accurate and reliable classifications. This method is particularly useful in applications where high precision is required, and it demonstrates the potential of advanced machine learning techniques in improving time series classification tasks.

Gu et al. [12] investigated a motor on-line fault diagnosis method based on 1D-CNN and multi-sensor information. Their research highlights the effectiveness of integrating multiple sensor data with 1D-CNN models to accurately diagnose faults in real-time, thus improving the reliability and operational efficiency of electric motors. The study shows that combining data from various sensors provides a more comprehensive understanding of the motor's condition, enabling early detection of potential issues. This method not only enhances fault detection accuracy but also contributes to the development of more robust and reliable diagnostic systems for industrial applications.

Ertarğın et al. [13] explored a deep learning approach for motor fault detection using mobile accelerometer data. This study emphasizes the practicality of using readily available mobile sensors to collect data, which is then processed using deep learning

models to detect motor faults, providing a cost-effective solution for fault diagnosis. The use of mobile accelerometers makes it possible to implement fault detection systems in a wide range of environments without the need for expensive and specialized equipment. This approach not only reduces costs but also simplifies the deployment of fault detection systems, making it accessible to smaller businesses and remote locations.

Morenas et al. [14] examined the edge application of machine learning techniques for fault diagnosis in electrical machines. Their research focuses on deploying machine learning algorithms at the edge, which allows for real-time processing and immediate fault detection, reducing the dependency on centralized computing resources. This approach is particularly beneficial in scenarios where latency and bandwidth are critical factors. By processing data locally at the edge, the system can provide quicker responses and maintain high performance even in environments with limited connectivity. This study highlights the growing trend of edge computing and its potential to revolutionize fault diagnosis in industrial settings (Morenas et al., 2023).

Abdelmaksoud et al. [15] introduced a convolutional-neural-network-based approach for multi-signal fault diagnosis of induction motors using single and multi-channel datasets. This study showcases the robustness of CNN models in handling different types of signal data, enhancing the accuracy of fault diagnosis in induction motors. By utilizing both single and multi-channel datasets, the research demonstrates the flexibility and adaptability of CNNs in various diagnostic scenarios. The ability to process multiple signal channels simultaneously allows for a more detailed analysis of the motor's condition, leading to more accurate fault detection and better maintenance strategies.

This literature review provides a comprehensive overview of current methods and applications of 1D CNNs in fault detection and diagnosis for electrical machines. Each study details different techniques and their effectiveness in practical applications. By examining these studies, we gain insight into the latest advancements in the field and the potential future directions for research and development. The integration of machine learning and deep learning techniques in fault detection systems represents a significant step forward in enhancing the reliability and efficiency of industrial operations.

1.1.3 CNN implementation on FPGA

Implementing Convolutional Neural Networks (CNNs) on Field-Programmable Gate Arrays (FPGAs) has emerged as a significant advancement in the field of fault diagnosis for electrical machines. This approach leverages the parallel processing capabilities of FPGAs to enhance the performance and efficiency of CNNs in real-time applications. This literature review examines the application of CNNs on FPGAs for fault detection, highlighting the latest innovations and trends in this area.

Xie et al. [16] developed a fault classification and diagnosis approach using FFT-CNN for an FPGA-based CORDIC processor. Their research demonstrates how FFT preprocessing combined with CNNs can improve fault detection accuracy and processing speed. The integration of FFT with CNNs allows for efficient feature extraction, making the FPGA implementation more effective for real-time applications.

Wu et al. [17] introduced a real-time motor fault detection system based on FPGA. This study highlights the advantages of using FPGAs to implement CNNs for motor fault detection, such as reduced latency and increased processing speed. The real-time capabilities of FPGAs make them ideal for applications where immediate fault detection and response are critical. This implementation showcases the practical benefits of deploying CNNs on FPGAs in industrial settings.

Li et al. [18] proposed a flexible CNN architecture for real-time FPGA implementation. Their research focuses on designing a CNN that can be easily reconfigured to adapt to different fault detection scenarios. The flexibility of the architecture ensures that it can be optimized for various applications, enhancing the overall efficiency and effectiveness of the fault detection system. This study demonstrates the potential of flexible CNN architectures in improving the adaptability and performance of FPGA-based fault detection systems.

Kumar and Hati [19] explored the use of CNNs with batch normalization for fault detection in squirrel cage induction motors. This study shows that incorporating batch normalization in CNNs enhances their performance by stabilizing the learning process and improving convergence rates. The FPGA implementation of this approach ensures that the fault detection system can operate in real-time, providing timely and accurate diagnostics in industrial environments.

Liu et al. [20] presented a real-time FPGA-based hardware neural network for fault detection and isolation in more electric aircraft. Their research focuses on the implementation of CNNs on FPGAs to achieve high-speed processing and low latency, which are crucial for aerospace applications. This study highlights the importance of FPGA-based CNNs in enhancing the reliability and safety of aircraft systems by providing real-time fault detection and isolation capabilities.

Syed et al. [21] developed an FPGA implementation of a fault-tolerant fused and branched CNN accelerator with reconfigurable capabilities. Their research emphasizes the importance of fault tolerance and reconfigurability in CNN implementations on FPGAs. The ability to reconfigure the CNN architecture allows for continuous optimization and adaptation to changing fault detection requirements, making the system more robust and versatile.

Osornio-Rios et al. [22] investigated an FPGA-microprocessor-based sensor for fault detection in induction motors using time-frequency and machine learning methods. This study highlights the integration of time-frequency analysis with machine learning algorithms on FPGAs to enhance fault detection accuracy and efficiency. The combination of FPGAs and microprocessors provides a powerful platform for real-time fault diagnosis in industrial applications.

Karim et al. [23] explored the use of FPGA-based on-line fault diagnostic systems for induction motors using electrical signature analysis. Their research demonstrates how FPGAs can be used to implement advanced fault detection algorithms, providing high-speed and accurate diagnostics. The study highlights the benefits of using FPGAs for real-time fault detection in industrial environments, emphasizing their potential to improve the reliability and efficiency of motor systems.

From those reviews it is clear that although there are many studies related to detection and and classification of electrical machine faults. Hardware implementation of those works are limited with most of them are implemented on microprocessors. Therefore, this work will be unique from many perspective since the CNN model that is trained for the detection of electrical machine faults will be implemented on an FPGA.

1.2 Purpose Of Thesis

The aim of this thesis is to develop and implement an Artificial Neural Network (ANN) based fault diagnosis system for electrical machines on an FPGA platform. The system will leverage the high-speed processing and parallel computing capabilities of FPGAs to achieve real-time fault detection and diagnosis, improving the reliability and efficiency of electrical machines. Also this method provides a generalized tool for the detection and classification of the electrical machine faults. Additionally, proposed method for the CNN model will be tested for different types of problems and trained with different dataset to question whether it is adaptive for those different conditions. Finally, this study is only a pioneer. In other words, at the end of the study, no product will be an output. Instead, this study will show that it is feasible to use an embedded platform which is mainly an FPGA to detect and classify electrical machine faults by demonstrating its processing capabilities.

1.3 Outline Of Thesis

In the first two subtitles, introduction and literature review are introduced. After that, study that is conducted is presented. This study explained in two main titles which are composed of work on “deep learning” part and FPGA implementation part. And in the subsequent part, experimental results will be exhibited. Experimental results further divided onto two as simulation and real time implementation results. In detail, in “Deep learning” part, the used dataset will be introduced. Then, pre-processing step for the preparation of the data will be explained. After that, intuition of using proposed 1D-CNN architecture will be mentioned. Next, testing and training results of the proposed model which is created on the python environment will takes place.

In the FPGA implementation part, used processor and its specifications are explained. Also, Created hardware structure of 1D CNN model by using “VHDL” , is going to be introduced. There are 4 main layers in the implementation part. Those are “Convolution_1D”, “Convolution_1D_Middle”, “Convolution_1D_No_MP” and “Dense” layer. All of these are explained in detail as “VHDL” structures. In addition, created custom hardware for this model on vivado is going to be explained in detail together with the hardware not to specific to this model but included in the study.

Finally, in the “Experimental Results” part, both simulation and hardware test results will be presented.

2. DEEP LEARNING PART

The reliable operation of electrical machines is critical for the continuity of industrial processes. However, the early detection of potential faults in these machines can optimize maintenance processes and extend their lifespan. This study aims to detect and classify potential faults in electrical machines using CNN (Convolutional Neural Network) networks. Fault detection in electrical machines plays a vital role in enhancing system reliability and efficiency. Traditional methods are often limited and struggle to adapt to modern technology. Therefore, machine learning and artificial intelligence-based methods, especially deep learning techniques, offer great potential in this field. CNN networks are ideal tools for fault detection in electrical machines due to their ability to learn local features in image and signal data. This study aims to demonstrate the potential of CNN networks in this field and show their embedded implementation compared to existing methods. In this part deep learning part of this study will be explained.

2.1 Dataset Creation

To develop a competent machine learning model for classifying motor faults, it is crucial to have a sufficient amount of data for model training. This research utilizes the publicly available Machinery Fault Database. This database includes 1951 different time series acquired by sensors on SpectraQuest's Machinery Fault Simulator (MFS) Alignment-Balance-Vibration (ABVT). These 1951 samples represent six different simulated conditions: normal operation, imbalance fault, horizontal and vertical misalignment faults, and inner and outer bearing faults. The representation these signals can be seen in Figure 2.1.

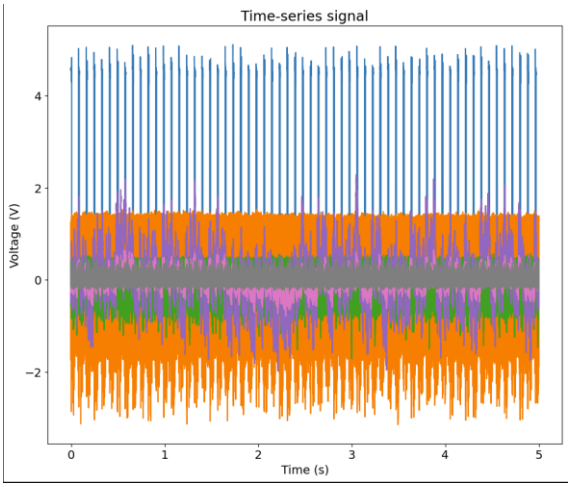

Figure 2.1 : All time series signals that creates one sample.

In this study, a dataset has been created using these samples, which is used for training and testing the machine learning model. In the initial stage, it is necessary to process the data obtained from the source and organize it into a suitable data structure. The data is organized into a series of folders labelled according to the operating condition. These folders contain data stored in comma-separated values (CSV) files, each serving as a training example.

However, in this study, only 3 of the 6 fault types will be used. These are 2.0 mm horizontal misalignment fault, 25g imbalance fault, and normal condition with no faults. There are 49 data samples for the normal condition and the imbalance condition, but 47 samples for the misalignment fault. Each data sample consists of 6 data accelerometers and one microphone recording for the sound of motor and one for speed recording. The accelerometers are attached to the motor from different angles and directions. Each time signal is sampled at 50,000 Hz for 5 seconds. Thus, each data sample consists of 250,000 data points. Additionally, all fault types and normal condition data were collected while the same motor was running at the same speed.

In the data processing stage, it is necessary to normalize and resample the collected data. Since each signal consists of 250,000 data points, processing and classifying this large dataset is challenging. Therefore, it is essential to reduce the number of data points by lowering the sampling frequency. In this study, it has been found that the accuracy obtained when the data is resampled at 500 Hz is sufficient. Therefore, the data will be resampled at 500 Hz for training the model.

2.1.1 MAFAULDA dataset overview

The Machinery Fault Database (MAFAULDA) is a comprehensive collection of multivariate time-series data used for fault diagnosis in mechanical systems. The dataset was acquired using sensors on a SpectraQuest Machinery Fault Simulator. It includes six different states: normal function, imbalance, horizontal misalignment, vertical misalignment, underhang bearing fault, and overhang bearing fault. Researchers use this dataset to test and validate various diagnostic models and techniques.

2.1.2 Fault types

The dataset includes the following fault types in the Table 2.1

Table 2.1 : List Of Common Faults.

Fault Type	**Description**
Normal	Normal operation with no faults.
Imbalance	Condition where the rotor is imbalanced.
Horizontal Misalignment	Misalignment in the horizontal direction.
Vertical Misalignment	Misalignment in the vertical direction.
Underhang Bearing Fault	Faults in the bearing located under the rotor, including outer race, ball, and cage faults.
Overhang Bearing Fault	Faults in the bearing located over the rotor, including outer race, ball, and cage faults.

2.1.3 Data collection and pre-processing

The vibration signal of the electromechanical equipment bearing is recorded as a one-dimensional time series. Each dataset is divided into groups of 400 data points. The Gramian Angular Field (GAF) method is used to encode the time series values as angular cosines and timestamps as radii, preserving time dependencies and transforming the time series into images. This pre-processing step is crucial for applying advanced diagnostic algorithms.

2.1.4 Fault conditions and sample distribution

The dataset encompasses a wide range of fault conditions. Each condition has a specific number of samples collected under different scenarios to simulate real-world

operating environments. Table 2.2 summarizes the distribution of samples across different fault conditions:

Table 2.2: Number of samples in the dataset.

Fault Type	**Number of Samples**	**Description**
Normal	49	No faults present, baseline condition.
Imbalance	333	Rotor imbalance condition.
Horizontal Misalignment	197	Misalignment in the horizontal direction.
Vertical Misalignment	301	Misalignment in the vertical direction.
Underhang Bearing Fault	558	Faults in the bearing located under the rotor.
Overhang Bearing Fault	513	Faults in the bearing located over the rotor.

2.1.5 Applications and usage

The MAFAULDA dataset is extensively used in the field of fault diagnosis and predictive maintenance. Researchers utilize this dataset to develop, test, and validate various machine learning and deep learning algorithms. The primary applications include:

- Developing diagnostic models for early detection of faults in machinery.
- Validating the performance of new algorithms against established benchmarks.
- Enhancing the reliability and efficiency of predictive maintenance systems.

2.2 Data Processing

As mentioned above, each sample consists of 6 accelerometer data and one microphone data and one speed data. Hence, it is normal for these signals to be on different scales. However, to prevent these different scales from affecting the decision of the artificial intelligence model, these signals must be normalized. Additionally, each signal consists of 250,000 data points, which presents a substantial processing load for classification and processing. Due to the reasons mentioned above, reducing

the number of data points without losing significant features can enable faster model training. By keeping the sampling duration constant and reducing the sampling frequency, we can reduce the number of data points. However, it is essential not to lose the distinguishing features of the signals during this process.

Another important step in the data processing stage is transforming the signals from the time domain to the frequency domain. This transformation is performed using the Fast Fourier Transform (FFT). The advantages of transforming the data to the frequency domain are as follows:

Reducing the Dataset Size which is using only the positive frequency spectrum reduces the number of rows per training set from 250,000 to 125,000, thereby decreasing the model training time.

More Determinative Features which is the frequency spectrum contains more determinative features according to the operating conditions, thus increasing the model accuracy.

Noise Reduction which is the frequency domain allows filtering of noise in the signals and provides a clearer view of the fundamental components of the signal.

Since we have horizontal misalignment and imbalance fault for classification purposes. We already know that their specific fault signals should be at most 5^{th} harmonic of their fundamental rotation frequency. Therefore, keeping sampling frequency as 500 Hz should not cause much information loss. Since the first harmonic corresponds to speed of the motor which is at most 50 Hz as mentioned in the data set. Therefore, from Nyquist theorem, we do not lose the 5^{th} harmonic if we down sample the signals at 500 Hz

At the end of the data processing stage, the signals have been transformed from the time domain to the frequency domain and normalized as can be seen in Figure 2.2 and 2.3. These processed data are now ready for training the deep learning model.

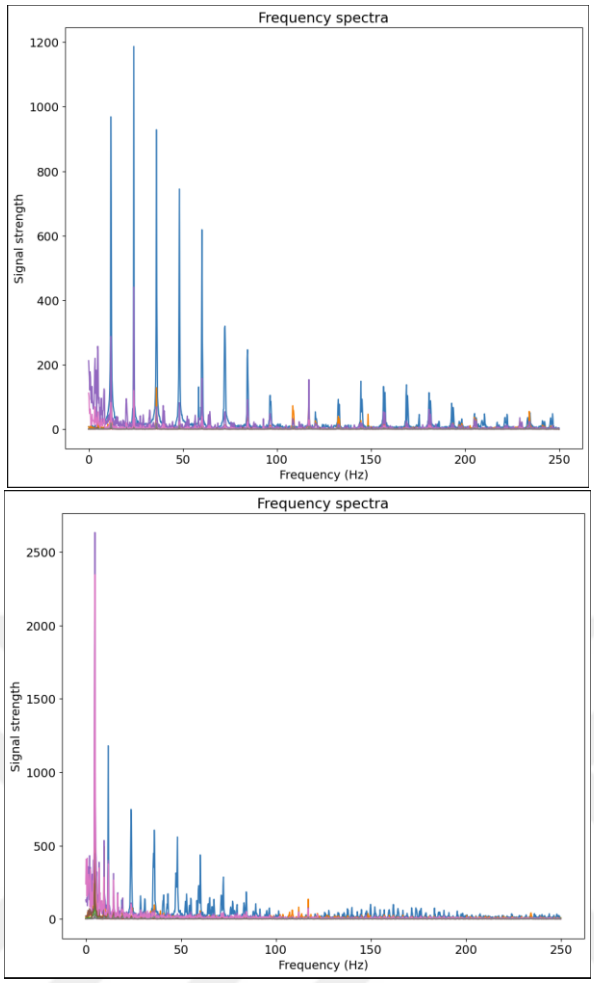

Figure 2.2: Horizontal misalignment frequency spectrum and Imbalance frequency spectrum.

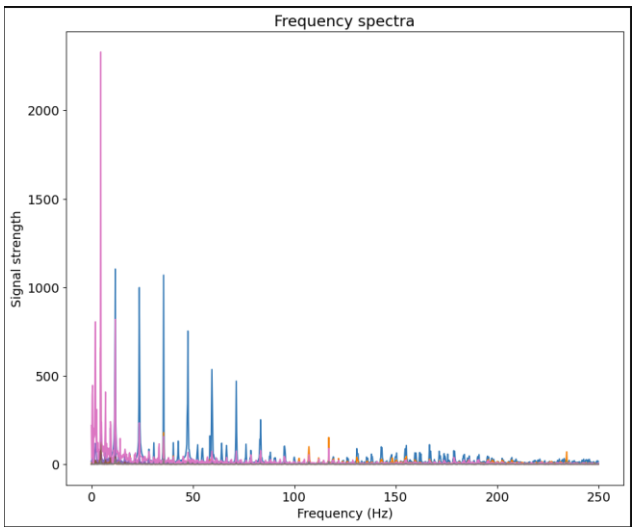

Figure 2.3: Normal Data frequency spectrum.

2.3 Developing the Deep Learning Model

In this study, a CNN (Convolutional Neural Network) neural network model will be used. CNNs are high-performing deep learning models, particularly in the fields of image and signal processing. CNNs excel in classification and detection tasks by learning local features in the data. For that purpose, A model that is proposed by (M.Ertağın ve Ark.,2023, 226) is used as deep learning model for this study since it was prepared for electrical machine faults diagnosis also the it is amongst the less complicated models proposed in literature. It can be seen in Figure 2.4.

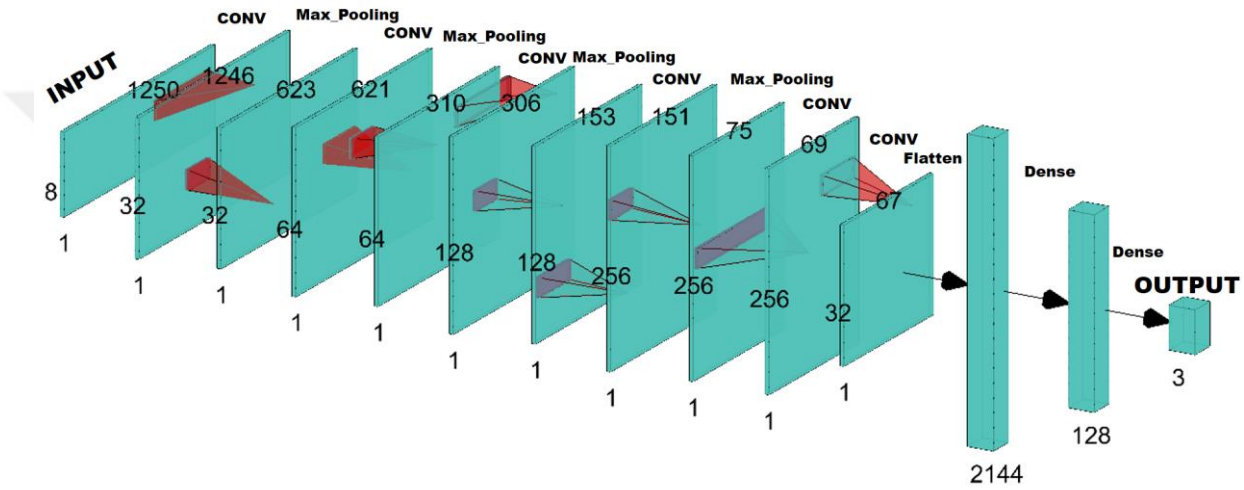

Figure 2.4: Proposed CNN Architecture.

In the development of this model, a two-layer structure was used. The first layer is the CNN network for feature extraction; the second layer is the linear classifier layer for performing the classification. The structure and the number of layers in the CNN network are determined as hyperparameters to optimize the model's accuracy. In this study, reducing the number of features and layers has been tried. However it has been found that a structure consisting of 6 convolutional layers provides the best results. Still, the parameter size is reduced by eliminating “BIAS” of the convolutional layers after finding that it has no effect on “Accuracy” of the trained model. Each convolutional layer uses filters to learn different features in the input data, and these filters are optimized to enhance the model's accuracy. Additionally, In the model, it is proposed that after the first “Dense” layer there exist an activation function which is called “Relu”. In this study, since the model will be implemented on embedded platform, the weights of the model should be decreased whenever possible. Therefore,

two dense layer is multiplied to create one dense layer after the training. And relu activation between these layers is obstacle to this reduction because it introduces a non-linearity. Thus, with the activation function in between them, resultant multiplied dense layer will not give the same result with the two separated dense layer. Therefore, no activation function is used between these two layers to effectively reduce the number of parameters to implement on FPGA. The reduction of parameter size is on the on the order of several hundred thousand. Compared to the total model parameters size. It is actually nearly %25 percent reduction in parameter size. Details can be seen in Figure 2.5.

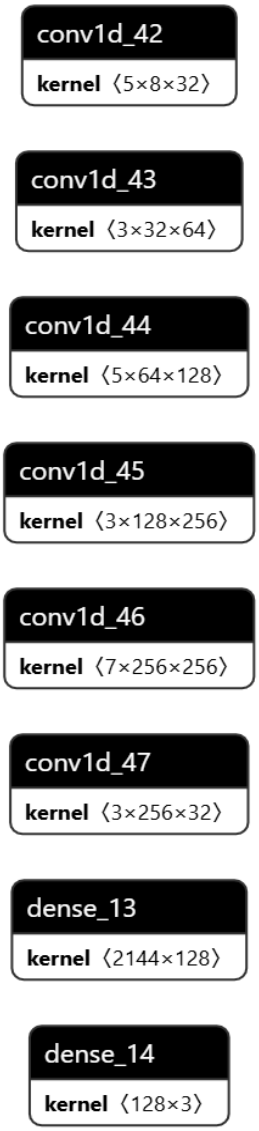

Figure 2.5: Proposed CNN Architecture layer specific details.

There are three classes that need to be classified. These are “Normal”, “Horizontal Misalignment fault” and “Imbalance fault” which are common fault types in electrical machine industry. Therefore, the outputs of the CNN network are divided into three different classes for classification. Python's TensorFlow library was used to develop the model. TensorFlow provides a robust framework for building and training deep learning models efficiently.

2.4 Model Training and Results

In this study, model training was conducted using the “Sparse Categorical Loss” function for optimization. “ADAM” optimizer is used for that purpose. The fundamental logic of model training is to minimize the error functions. As the model's accuracy increases, the error function will naturally minimize.

During model training, the model's parameters are updated using the training data. The performance of the model is monitored throughout the training process, and the generalization ability of the model is evaluated at regular intervals using a validation dataset. Throughout the training process, the error rates and accuracies on the training and validation datasets are recorded. This data helps identify whether the model is experiencing issues like overfitting or underfitting.

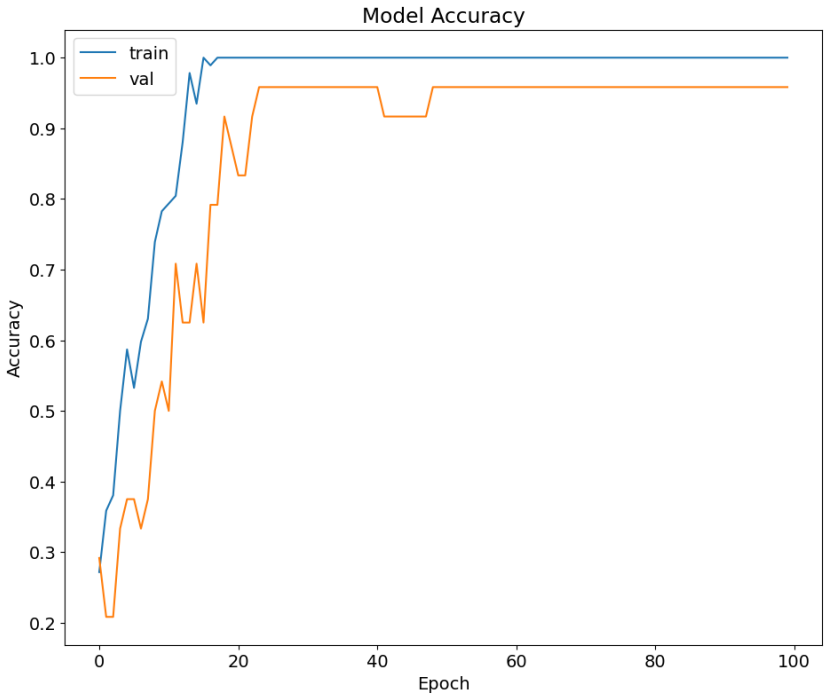

Figure 2.6: Training and Validation Accuracy.

The results of model training are evaluated using performance metrics such as accuracy, precision, recall, and F1 score. These metrics are used to measure the model's performance on different classes and its overall success. Based on the training results, no more hyperparameter adjustments are considered to improve the model's performance. As can be seen from the Figure 2.d6, model has %100 training accuracy and %95,83 validation accuracy.

2.5 Testing and Evaluation

During model training, the model might memorize the given data in some cases. To explain this situation, the model might create an output for each given data, forming a matching process, which is known as overfitting. This makes it impossible for the model to operate in real-time. Therefore, to evaluate the model's true accuracy, it is necessary to use data that was never used during training. Thus, a total of 28 test samples were used to evaluate the model.

In the testing and evaluation phase, the performance of the model is assessed on an independent test dataset. The test data consists of samples that the model has never seen during training, and it is used to evaluate the model's generalization ability. The performance of the model is measured using metrics such as accuracy, precision, recall, and F1 score on the test dataset.

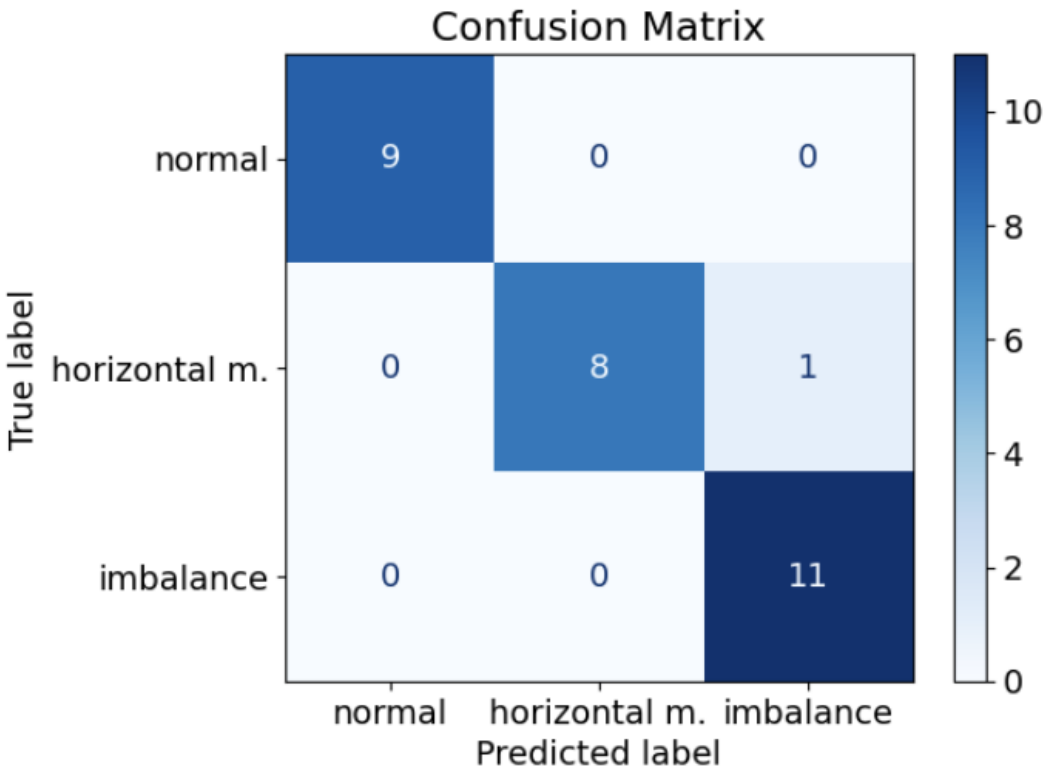

Figure 2.7: Confusion Matrix.

As can be seen from the Figure 2.7, Only one prediction is wrong among the 28 and it corresponds to %96.5 testing accuracy. Although primary concern of this study is not obtaining a high accuracy, it is still a good model for this aspect. Accuracy results can be summarized in Table 2.3. Also, other metric results can be found in Table 2.4.

Table 2.3: Metrics Of The Trained Model.

Training Accuracy	%100
Validation Accuracy	%95.83
Testing Accuracy	%96.5

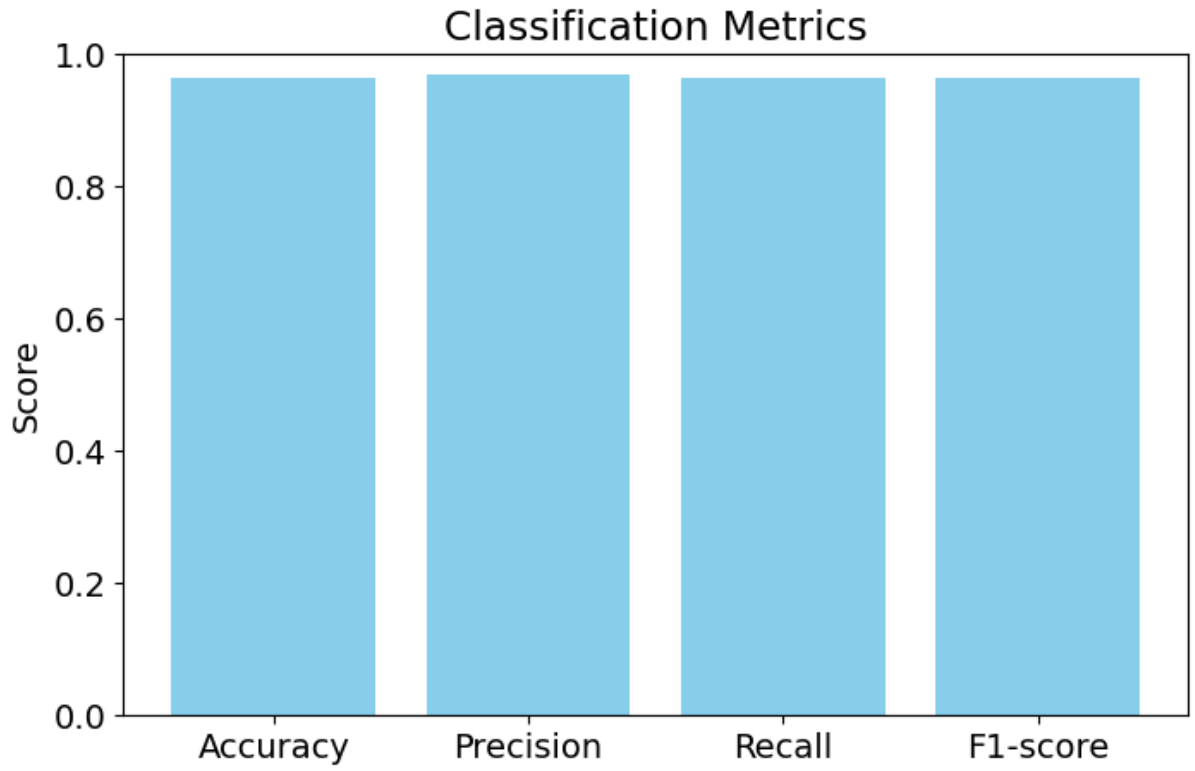

Figure 2.8: Classification Metrics.

Table 2.4: Metrics results.

Accuracy	**Precision**	**Recall**	**F1-Score**
0.9655	0.9683	0.9655	0.9652

3. VHDL IMPLEMENTATION

After obtaining the model weights that hardware design studies are started. The reliable operation of electrical machines is essential for the continuous and efficient performance of industrial processes. Early detection of potential faults in these machines can significantly optimize maintenance schedules and extend the lifespan of the equipment. In this thesis, we present the implementation of a fault detection system for electrical machines using Convolutional Neural Networks (CNNs) on an FPGA platform. The use of CNNs, known for their superior performance in pattern recognition tasks, allows for accurate and real-time fault classification based on sensor data.

The VHDL (VHSIC Hardware Description Language) code developed in this study is designed to perform real-time fault detection by processing data from multiple sensors attached to the electrical machine. The system includes several layers, each responsible for a specific function in the fault detection process. These layers include data acquisition, convolution operations, dense layer computations, and final fault classification.

There are 904842 parameters to keep in the memory for implementation purposes. Those parameters are 32-bit floating (Single precision) numbers. Thus, the total amount of memory size just to keep these parameters is 3.45MB. With an external memory chip, this amount of data can be handled.

Data Acquisition Layer which is implemented in the Convolution_1D.vhd and Convolution_1D_No_MP.vhd modules, collects real-time data from multiple sensors attached to the electrical machine. The data includes various sensor readings, which are critical for detecting faults. The “data_in” signal captures the input data to be processed.

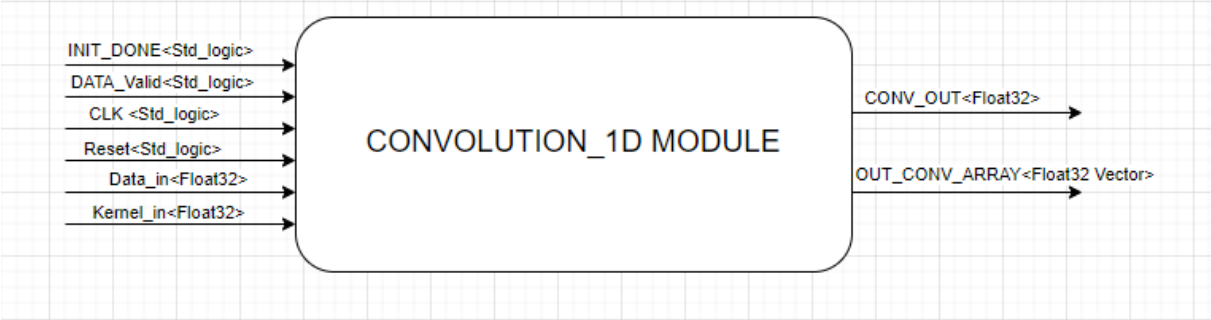

Figure 3.1: Input/output Block of 1D Convolution Module.

Convolution Layers which are The convolution operations are carried out by the Convolution_1D.vhd, Convolution_1D_No_MP.vhd, and Convolution_1D_Middle.vhd modules. Each module performs 1D convolution on the input data using predefined kernels. The differences between these modules lie in their specific roles and configurations.

Convolution_1D.vhd that module performs the initial convolution operation, capturing the fundamental features from the raw sensor data. Its block representation can be seen in Figure 3.1.

Convolution_1D_No_MP.vhd is similar to Convolution_1D.vhd, but without max pooling. This module is used to maintain higher resolution of features for more detailed analysis.

Convolution_1D_Middle.vhd is intermediate convolution layer further refines and enhances the features extracted from the previous convolution layers, ensuring that the features are progressively processed for higher accuracy in fault detection.

Dense.vhd module represents the fully connected (dense) layer, which further processes the features extracted by the convolution layers. This layer performs linear transformations on the input data using weights and biases, combining the features to make a final prediction.

The TOP_CNN.vhd module integrates all the aforementioned layers into a cohesive CNN architecture. This top-level module manages the flow of data through the different layers, ensuring that each layer's output serves as the input to the subsequent layer. The final output is a classification result indicating the presence and type of fault in the electrical machine.

By leveraging the parallel processing capabilities of FPGAs, the system achieves high-speed computation, making it suitable for real-time applications. This implementation addresses key challenges in fault detection, such as handling large volumes of data, ensuring low latency, and maintaining high accuracy. The VHDL design is structured to efficiently utilize FPGA resources, with specific attention to optimizing memory usage and computational efficiency. Through this work, we demonstrate the feasibility and effectiveness of deploying CNN-based fault detection systems on FPGA hardware, paving the way for more robust and reliable industrial maintenance solutions.

3.1 Intuitive Description Simulation Design

As can be seen from the introduction section of the VHDL implementation section there are 5 main layers to realize this application. Before explaining the intuitive idea, the model that is used in this study is revisited. Proposed layer actually composed of 13 individual layers. There are four “Conv1D”. Each of these 4 layers follows a “MaxPooling1D” layer. 2 “Conv1D” layer have no “MaxPooling1D” layer at their exit. Then, a “Flatten” layer exist to parametrize all of these feature vectors. For classification purposes, there are two subsequent “Dense” layer which will give classification scores after the operation on them.

For simulation purposes, VHDL language have some facilities. That is, by using some of the features of the VHDL, we can handle those 13 layers more easily like reducing layer number to 5 layers. Firstly, initial convolution layer should take its input serially, that is, this layer have some difference from other layers since it takes first input signal from external world. Therefore, we should keep this layer individual like before. After that, next 3 subsequent layers have all similar input and output ports. Also, their internals signals are the same. Therefore, those 3 layers can be generalized in a specific top layer. These 3 layers have max pooling layers at their exit as mentioned before. The maxpooling operation is just finding the maximum of the specific group of numbers (in our case stride is 2). Thus, instead of implementing a separate layer for the maxpooling operation. Those subsequent convolution1d and maxpooling layers can merge into one layer. After these layer, there are two convolution1d layers which have no maxpooling layer at their exit. These two layer have the same specific external ports and internal signals. So, these two can be generalized into one specific layer as

“Convolution1D_NML” NML: no maxpooling layer. After that, there is flattening layer which have no parameters to keep but just making a reshape on its input therefore, it is not efficient to make it as one individual layer. For that reason, flattening operation will be implemented within the dense layer. Finally, there are two consecutive dense layers before obtaining classification scores. Since those two layers are just composed of matrices, it is possible to make them one layer by multiplying first layer matrices with the second dense layer matrices. After the creation of individual layers, One Top layer is needed to create whole network. For that purpose, CNN_TOP layer is created. This layer holds all the generic values of individual layers and connect all individual layers to their consecutive layers.

Except Top layer, other layers consist of 3 states. Those states are “IDLE, INIT, OPERATION” states. In the IDLE state, hardware waits a signal to pass to INIT state. This state provides that in the absence of the layer inputs, avoids unnecessary operation load on FPGA. Also, after multiplication and summing operations, to not make all process more than one time, IDLE state is suitable place to settle in. when the start signal becomes high, All layers goes into INIT state. Since we are in simulation environment, parallel initialization of the layers are possible. This will be not be the case for the hardware implementation. It can be visualized as in Figure 3.2.

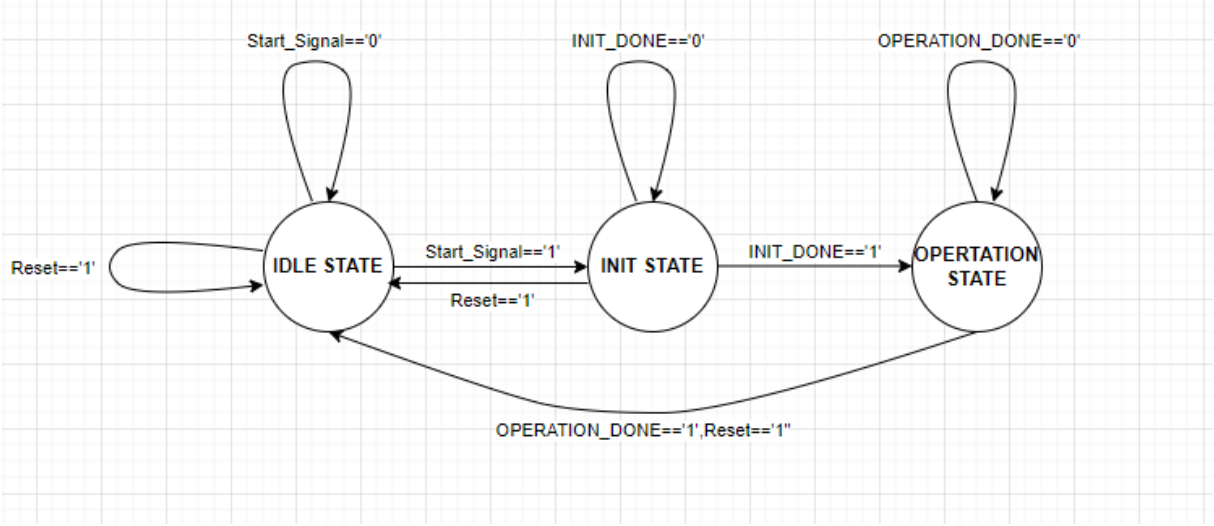

Figure 3.2: State Transition Diagram of Convolution Module.

In the INIT state, all kernels that are in those specific layers initialize with their parameters. For this operation. One exception is that since the very first layer is needed its input from the external world, when it is in the INIT state its input kernel is also initialized with the data that is tested for the hardware implementation. One specific

detail of this state is that since each variable on this implementation initialized as zero before the initialization, to avoid false assignment of these variables, “Data_valid”, signal is applied on each INIT state. This signal works such that when the first initialization occurs that is when variable get rid of its initial value, then , “data_valid” signal become high and initialization operation of each kernel matrices starts. There is one “FIFO” for each layer to keep kernel values. But after initialization is done, to ease the operation, each kernel FIFO’s translated to a matrices form. For the initialization of the kernels in vivado simulator, “std.textio” library of the VHDL is used. This library is not synthesizable. Thus, it will only be used to simulate the whole model. And in one clock cycle, all kernel FIFO’s take one input. For the hardware implementation, UART channel is used to initialize whole kernels.

There are different number of parameters for each layer. Thus, the initialization of the layers are not completed within the same duration. If each of the layers goes into “OPERATION” state independent of each other, it will lead to wrong results since most of layers takes its input from the previous layers. In order to solve this issue, After all of INIT states come to an end, each OPERATION state is started. To realize this idea, one concurrent assignment operation is used. Each layer produces a signal which is called “INIT_DONE” it is binary and when the INIT operation is done, it becomes high there are 8 initialization procedure. Therefore, 8 init_done signal comes into an “AND” operation to produce common “INIT_DONE” signal on the “TOP_LAYER”.

After the completion of INIT state, “OPERATION” state is executed. This state is main state of the whole hardware because all the multiplications and summation operations are done in this state. Also, since we merge “Relu” and “Max pooling” operations into convolution layers, these are also done on the “OPERATION” state. When all the data is ready for the operation. 1D convolution starts. For the simulation purposes, all convolution operations are done by parallel. Which is not possible in hardware implementation because it need several hundred thousands “DSP” multiplier. After each convolution operation there is a multiplexer at before the assignment of the convolution result. That multiplexer choose whether the result is positive or not. Which mimics the operation of the “Relu”. After that, there is counter which is called “Max_Pooling_Counter”. Since its stride is 2 in the whole model, it counts up to 2 (never reach 2). Reaching the limit of its implies that there will be

enough data to decide which one is biggest among them. Thus, when this counter counts to 1, at the same time 2 convolution results become available. And, higher number is assigned as convolution result to realize “Max Pooling” operation. One of last assumption made on simulation case is that since each layer mentioned above, need for its input to be ready to produce output, those inputs come from previous layers output as parallel. In other words, when on Convolution layer produce the convolution result as array, all the array is available for the next convolution layer. In hardware implementation, this will not be case because those available results normally keep on the flip-flops. Since those output sizes are very large compared to flip flop numbers in the FPGA processor. Total design for simulation purpose can be found in Figure 3.3.

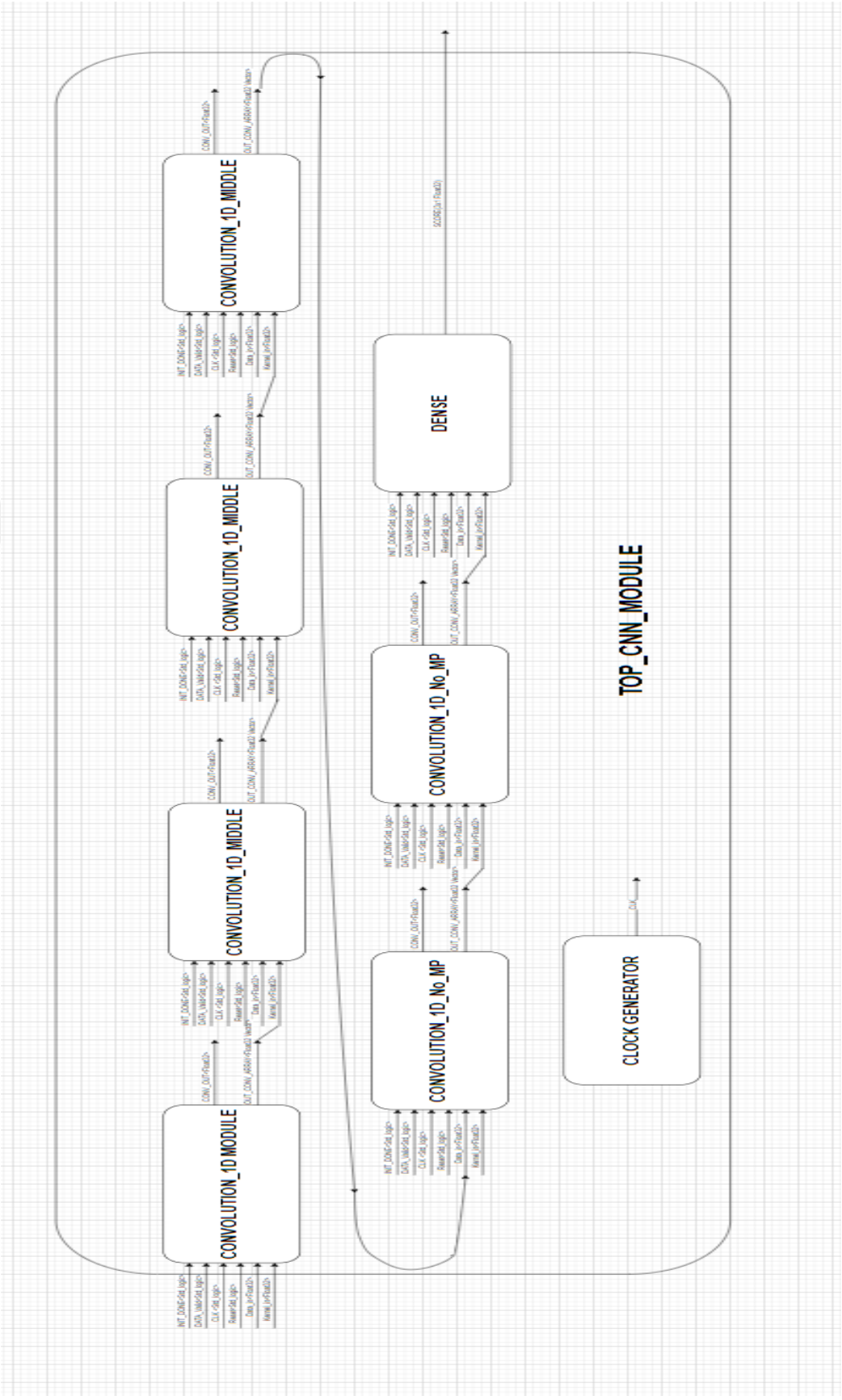

Figure 3.3: Structural Block Diagram of CNN Module.

3.2 Hardware Implementation

For the hardware implementation, the approach that is used in simulation part will be changed considerably. In the simulation environment, we are free to do huge number multiplications and additions in one clock cycle however, for the hardware implementation we are limited with number of DSP blocks in the FPGA. Therefore, it is a must to divide number of multiplications and summations between consecutive clock cycles.

Another difference between the simulation and hardware implementation is that we have operating system at background of simulation environment. Thus, we can initialize all the matrices and arrays at the same time because operating systems allows the use of threads. On the other hand, in hardware implementation, we have to initialize all signals respectively. In our case, UART module is created and has been used for the initialization of those signals. That is, In the computer, we have all the weights that are extracted from model that is trained for the fault classification. And those weights are sended from computer to FPGA by using UART module.

The most important difference in the hardware implementation is that “REAL” data type can not be used since it is not synthesizable. Instead, all the parameters and weights have to be in the shape of a logic vector. The main problem both in simulation and hardware implementation is that since FPGA don’t know what should be the precision after any multiplication. It makes straightforward multiplication operation. That is, when two numbers which are for example 32-bit are multiplied, it produces a 64-bit output. Since this work consist of many consecutive multiplication operations, after the operation is done there will be large number of bits for defining a single number. It is not feasible and necessary to make this kind of operation. Therefore, “32-bit Floating” numbers are used either in training part and implementation part.

In VHDL 2008, There are non-standard libraries that enable 32-bit Floating number operations. These libraries also can be synthesizable. Apart from facilating multiplication and summation operation there are many useful functions to realize 32-bir arithmetic. For example, any number that is in shape “STD_LOGIC_VECTOR” can be converted to 32-bit Floating number by just using the function “TO_FLOAT”.

In summary, instead of using “TEXT I/O” library, UART module is used for both the initialization of the filter weights and obtaining results on the PC by using it with

Python User Interface. And, since FGPA chip does not have enough memory to keep all the values, “DDR2 Memory Controller” is used to keep all the weights.

After the simulation and choosing the platform, custom hardware which is created for that specific model is introduced together with the non-specific hardware modules which are UART and DDR2 memory.

3.2.1 Nexys A7 development board

The Nexys A7 is a powerful development board produced by Digilent, based on Xilinx's Artix-7 FPGA. This board is ideal for educational and research purposes in areas such as digital system design, signal processing, digital communication, and microcontroller development.

3.2.1.1 Features and components

The Nexys A7 is equipped with various peripherals and expansion options. Its key features included in Table 3.1 and the card is represented in Figure 3.4.

Table 3.1: Used FPGA Board Specifications.

FPGA	Xilinx Artix-7 FPGA (options: XC7A100T or XC7A50T)
Memory	128 MB DDR3 SDRAM, 16 MB Quad-SPI Flash memory
I/O Ports	15 digital I/O, 2 analog inputs, 4 Pmod ports, 1 micro USB port
Display	VGA port
Audio	Microphone, jack input, and output
User Interface	8 LEDs, 8 switches, 5 buttons

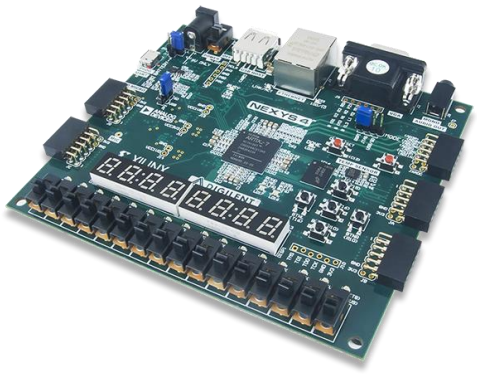

Figure 3.4: Picture of Nexys A7.

The Nexys A7 is also a highly suitable platform for digital signal processing (DSP) applications. The DSP48E1 slices on the FPGA are optimized for high-speed mathematical operations. DSP48E1 slices in the Xilinx Artix-7 FPGA can perform high-speed multiplication and addition operations, enabling efficient hardware implementation of signal processing algorithms. The parallel processing capabilities of the FPGA allow for multiple DSP operations to be performed simultaneously, which is a significant advantage in real-time data processing applications.

The Nexys A7 supports the efficient hardware implementation of Finite Impulse Response (FIR) filters, commonly used in signal cleaning and frequency selection tasks. Fast Fourier Transform (FFT) and Inverse FFT (IFFT) operations can be performed with high efficiency on the Nexys A7, which is crucial for frequency analysis and modulation applications.

3.2.1.2 Applications

The Nexys A7 can be used for a wide range of applications. Educational institutions use this board to provide students with hands-on experience in digital circuits, FPGA design, and embedded systems. Researchers and engineers also prefer the Nexys A7 for rapid prototyping and custom digital system development projects.

3.2.1.3 AI applications and CNN implementation

The Nexys A7 is particularly well-suited for artificial intelligence (AI) applications, especially for implementing Convolutional Neural Networks (CNNs). The FPGA's high parallel processing capability makes it ideal for handling the intensive computations required by CNNs. By leveraging the DSP48E1 slices for efficient matrix multiplications and convolutions, the Nexys A7 can accelerate the performance of AI models. This capability is crucial for real-time image and signal processing tasks, making the Nexys A7 an excellent choice for developing AI-driven solutions in resource-constrained environments.

3.2.1.4 Development tools

The Nexys A7 is compatible with Xilinx's Vivado Design Suite. Vivado allows users to design, simulate, and synthesize complex digital circuits using HDL (Hardware Description Language). This tool provides a comprehensive platform for developing and testing projects on the Nexys A7.

3.2.1.5 Advantages

High Performance: The Artix-7 FPGA is known for its high processing power and low power consumption, allowing efficient execution of complex algorithms and data processing tasks.

Expandability which is Pmod ports and other expansion options offer flexibility for users to add various sensors and modules.

Ideal for Education and Prototyping. That is, the easy-to-use hardware and software tools make the Nexys A7 ideal for both educational and research projects.

The Nexys A7 development board offers a powerful and versatile platform for both students and professionals. The combination of the Artix-7 FPGA's power, various peripherals, DSP features, and expansion options makes this board an indispensable tool in the world of digital design and embedded systems. Its suitability for AI applications, particularly for CNN implementations, further enhances its value, providing an excellent environment for developing advanced AI solutions.

Moreover, the processor on this board is XC7A100T-1CSG324C and its resources can be summed up in Table 3.2 below.

Table 3.2: Resources of the used processor.

Logic Cells	101,440
CLB (Configurable Logic Blocks)	15,850
LUTs (Look-Up Tables)	63,400
Flip-Flops	126,800
Block RAM	4,860 Kbits (60 Units)
DSP Slices	240
Clock Management Tiles	6 (4 MMCM, 2 PLL)
I/O Pins	210
Maximum I/O Voltage	3.3V

3.2.2 UART module

A UART is a device used in serial communication and is commonly used to facilitate data communication between microcontrollers and other devices. This VHDL code describes the design of a UART module and explains the purpose of the main ports

and signals. In this work, UART model is used for transferring the sample data to FPGA and sending the obtained score to the computer.

3.2.2.1 Ports and general structure

The ports include i_Clock, i_Serial_RX, i_Start_Transmit, i_Transmit_Byte, o_Is_Byte_Received, o_Is_TX_Active, o_Serial_TX, o_Rx_Byte, and o_TX_Done. The i_Clock port provides the clock signal to the UART module, controlling the timing of data reception and transmission processes. The i_Serial_RX port is the input line where serial data from an external device is received. The i_Start_Transmit port is a control signal used to initiate the data transmission process. The i_Transmit_Byte port contains the 8-bit data to be transmitted. Output ports include o_Is_Byte_Received, o_Is_TX_Active, o_Serial_TX, o_Rx_Byte, and o_TX_Done. The o_Is_Byte_Received indicates that a byte of data has been fully received, while o_Is_TX_Active signifies that the UART module is busy with data transmission. The o_Serial_TX port is the output line through which data is transmitted to the external device. The o_Rx_Byte port holds the 8-bit received data, and o_TX_Done indicates that the data transmission process is complete.

In the design of the UART module, separate state machines are used for data reception and transmission. These machines update the signals based on specific states to ensure proper data communication. During the data reception process (RX), signals such as r_RX_Clk_Count, r_RX_State, r_RX_Bit_Index, and r_RX_Byte are used. The r_RX_Clk_Count is a clock signal counter that counts the duration for which a bit is received. The r_RX_State holds the current state of the receiver state machine, with states defined as IDLE, WAITING_FOR_START, WAITING_FOR_BITS, and WAITING_FOR_END. The r_RX_Bit_Index keeps track of the index of the received bit, while the r_RX_Byte stores the 8-bit received data.

The data reception process consists of four main stages. In the IDLE state, the UART module is in a waiting state, looking for the serial receiver line to go low ('0'). In the WAITING_FOR_START state, it waits until the middle of the start bit. In the WAITING_FOR_BITS state, the data on the serial receiver line is read and written to the r_RX_Byte signal during each bit period. In the WAITING_FOR_END state, once the last bit is complete, the received data is written to the o_Rx_Byte signal, and the o_Is_Byte_Received signal is activated.

During the data transmission process (TX), signals such as r_TX_Clk_Count, r_TX_State, r_TX_Byte, r_TX_Bit_Index, and r_TX_Done are used. The r_TX_Clk_Count is a clock signal counter that counts the duration for which a bit is transmitted. The r_TX_State holds the current state of the transmitter state machine, with states defined as IDLE, START_BIT, WRITING_BITS, and WRITING_END_BIT. The r_TX_Byte stores the 8-bit data to be transmitted, and the r_TX_Bit_Index keeps track of the index of the bit to be transmitted. The r_TX_Done indicates that the data transmission process is complete.

The data transmission process consists of four main stages. In the IDLE state, the UART module is in a waiting state and the data transmission process has not started. In the START_BIT state, the start bit ('0') is transmitted. In the WRITING_BITS state, the data in the r_TX_Byte signal is written to the o_Serial_TX line during each bit period. In the WRITING_END_BIT state, once all bits are transmitted, the stop bit ('1') is transmitted, and the o_Is_TX_Active signal is deactivated.

This VHDL code defines the main ports and signals necessary to perform the basic functions of the UART module. Input ports such as i_Clock, i_Serial_RX, and i_Start_Transmit are used to initiate and control data reception and transmission processes, while output ports such as o_Is_Byte_Received, o_Is_TX_Active, o_Serial_TX, and o_Rx_Byte indicate the results and states of these processes. The signals and state machines used in the RX and TX processes ensure that data communication is performed correctly and synchronized. This structure includes all the components necessary for the UART module to perform its basic functions.

3.2.3 Memory controller

DDR2 (Double Data Rate 2) memory controllers are vital components in modern computing systems, facilitating efficient communication between the CPU and DDR2 memory modules. The given VHDL declaration of the DDR2 memory controller entity, mig_7series_0, outlines the various ports involved in the operation of this controller. This part provides an overview of the DDR2 memory controller, its functionality, and the key parameters that define its operation based on the provided declaration for this work. In this work, DDR2 memory is a critical component since the weights extracted from the model is in the order of Megabytes which means for most block chips block rams it is too large to handle. Therefore, recording and

processing this amount of data would be impossible if an external memory chip like DDR2 was not available.

3.2.3.1 Overview of DDR2 memory

DDR2 memory is an advanced version of the original DDR memory technology, providing higher data transfer rates and improved performance. DDR2 memory achieves this by using differential signaling and higher clock speeds, allowing data to be transferred on both the rising and falling edges of the clock signal, effectively doubling the data rate compared to its predecessor.

3.2.3.2 DDR2 memory controller

The DDR2 memory controller is responsible for managing data flow between the CPU and DDR2 memory modules. It orchestrates all read and write operations, ensuring that data is transferred efficiently and accurately. The controller handles tasks such as address mapping, command scheduling, and timing management, which are crucial for optimal performance of the memory subsystem.

3.2.3.3 Key ports and their functions

The DDR2 memory controller, mig_7series_0, includes several ports categorized into different groups based on their functions. Data ports such as ddr2_dq, ddr2_dqs_p, and ddr2_dqs_n are used for transferring data to and from the DDR2 memory modules. Address and command ports like ddr2_addr, ddr2_ba, ddr2_ras_n, ddr2_cas_n, and ddr2_we_n manage the memory addresses and control signals for read/write operations. Clock and control ports, including ddr2_ck_p, ddr2_ck_n, ddr2_cke, ddr2_cs_n, ddr2_dm, and ddr2_odt, provide the necessary clock signals and control functionalities to ensure synchronized data transfers and enable features such as data masking and on-die termination.

Application interface ports like app_addr, app_cmd, app_en, app_wdf_data, app_wdf_end, app_wdf_mask, app_wdf_wren, app_rd_data, app_rd_data_end, app_rd_data_valid, app_rdy, and app_wdf_rdy facilitate communication between the memory controller and the application layer, enabling the initiation and validation of memory operations. Additional ports such as app_sr_req, app_ref_req, app_zq_req, app_sr_active, app_ref_ack, and app_zq_ack manage self-refresh, refresh, and calibration requests and acknowledgments. System interface ports, including ui_clk,

ui_clk_sync_rst, init_calib_complete, sys_clk_i, and sys_rst, provide the system clock signals and reset functionalities necessary for the controller's operation.

3.2.3.4 Parameters of DDR2 memory controller

Various parameters define the operation and performance of a DDR2 memory controller. The clock frequency determines the speed at which the memory controller operates. DDR2 memory typically operates at clock speeds ranging from 200 MHz to 533 MHz (DDR2-400 to DDR2-1066), with effective data transfer rates of 400 MT/s to 1066 MT/s. The controller must synchronize with the memory's clock frequency to ensure proper data transfers.

Latency timings are critical parameters that affect memory performance. Key latency timings include CAS Latency (CL), the number of clock cycles between the memory controller sending a read command and the data being available on the data bus. Lower CAS latency indicates faster memory access. RAS to CAS Delay (tRCD) is the delay between the row address strobe (RAS) and the column address strobe (CAS) signals, affecting the time to access a specific column within an active row. Row Precharge Time (tRP) is the time required to deactivate the current row before activating the next row, influencing the memory's ability to switch between different rows quickly. Row Active Time (tRAS) is the minimum time a row must remain active before a precharge command can be issued, impacting the time a row is kept open for accessing data.

The data bus width determines how many bits of data can be transferred simultaneously. DDR2 memory modules typically have a 64-bit wide data bus, allowing for efficient data transfers. The memory controller must match this bus width to ensure compatibility and optimal performance. The memory capacity parameter defines the total amount of DDR2 memory that the controller can manage, with typical configurations ranging from 512 MB to 8 GB per module. However in this work 128 MB "MT47H64M16HR-25E" chip is used. The controller must handle address mapping and command scheduling for the entire memory capacity effectively.

Power management is crucial for reducing energy consumption and extending the lifespan of DDR2 memory modules. The DDR2 memory controller incorporates various power management features, such as self-refresh mode, which allows the memory to enter a low-power state while retaining data, reducing power consumption during periods of inactivity. Power-down mode temporarily disables parts of the

memory to save power when not in use. Dynamic frequency scaling adjusts the operating frequency based on the current workload, balancing performance and power efficiency.

The DDR2 memory controller, as defined by the mig_7series_0 entity, is a pivotal component in managing communication between the CPU and DDR2 memory modules. Its functionality encompasses address mapping, command scheduling, and timing management, ensuring efficient and reliable data transfers. Key parameters such as clock frequency, latency timings, data bus width, memory capacity, and power management features are essential for optimizing system performance and ensuring the seamless operation of DDR2 memory in modern computing environments. Understanding these parameters is crucial for achieving high performance and energy efficiency in memory subsystems. Inputs and outputs of DDR2 memory can be found in Figure 3.5.

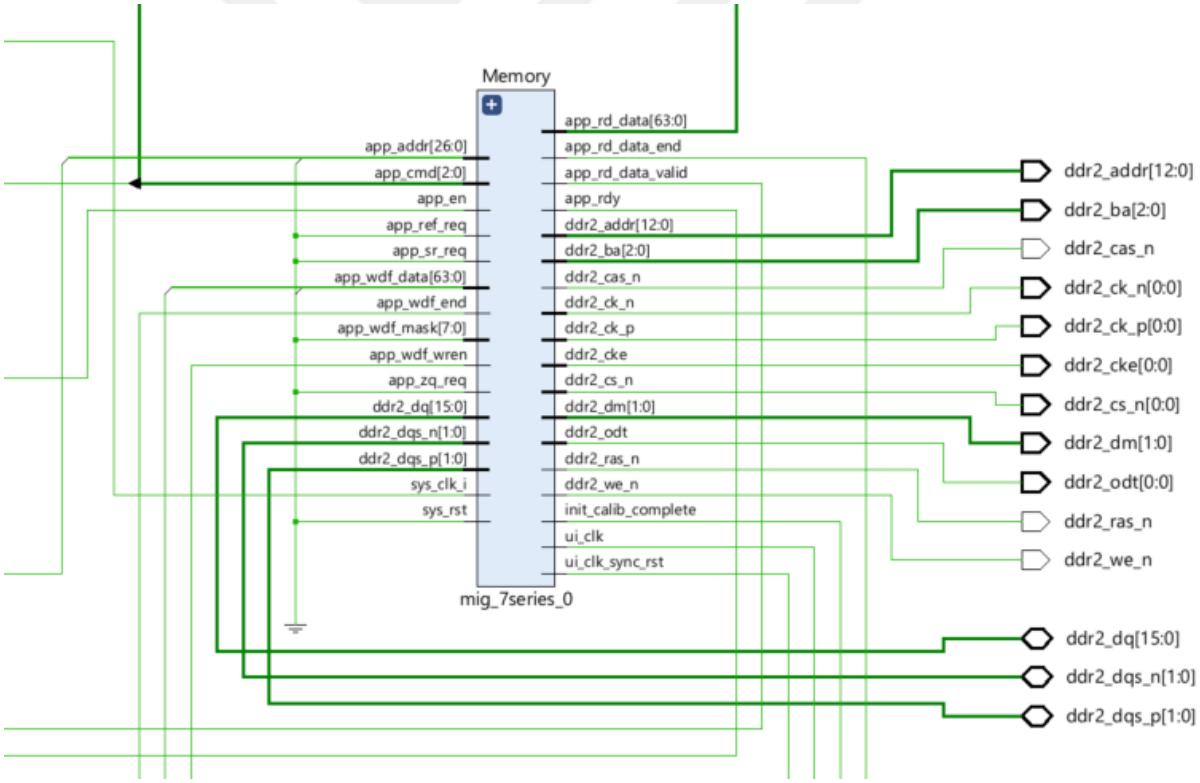

Figure 3.5: Block Diagram Of DDR2 Memory Controller.

3.3 Custom Hardware Design

In this work, actual aim is that creating a specific hardware platform for the proposed CNN model to realize the operation. For that purpose, only hardware implementation of the whole model was tried. However, there are lots of logic and arithmetic

operations which are unfeasible in terms of development time and debugging purposes. Therefore, we utilized the aid of software whenever possible. In other words, to create a hardware for that specific work, we need many custom hardware blocks which are all specific to do some task and to provide collaboration of these individual hardware blocks, a software code is created in vitis platform.

During the implementation, “Block Design IP” of the vivado toolbox is used for development platform. Block Design IP allows users to create FPGA designs using a graphical interface, simplifying the integration of IP cores and custom modules. Users can connect components like processors, memory controllers, and communication interfaces with drag-and-drop functionality. This approach streamlines complex FPGA design and configuration, making tasks faster compared to manual VHDL/Verilog coding.

For this application, both the default and custom IPs are used. The most important blocks that are used will be explained one by one. Additionally, Since Nexys A7 board is used, usage of these blocks are mainly determined by hardware used.

At first, Algorithmic state machine(ASM) is created. In Figure 3.6, very small but good representative part of the ASM is shown. First, initial state is “Kernel_INIT_1”. After it initializes the kernel_buffer, it checks for whether the kernel_counter is reached total number of kernels for that specific layer. If not, it passes state Data_INIT_1. If yes, it passes to state Kernel_INIT_2 then the state machine proceeds layer by layer periodically until it reaches the RESULT.

In Data_INIT_1 state, it checks whether the Data_Counter reaches the total number of data or not. If yes, then, it passes to Kernel_INIT_1 for new kernel_buffer. If not, then it passes to the operation state to produce single output which is also one element of next layers input layer.

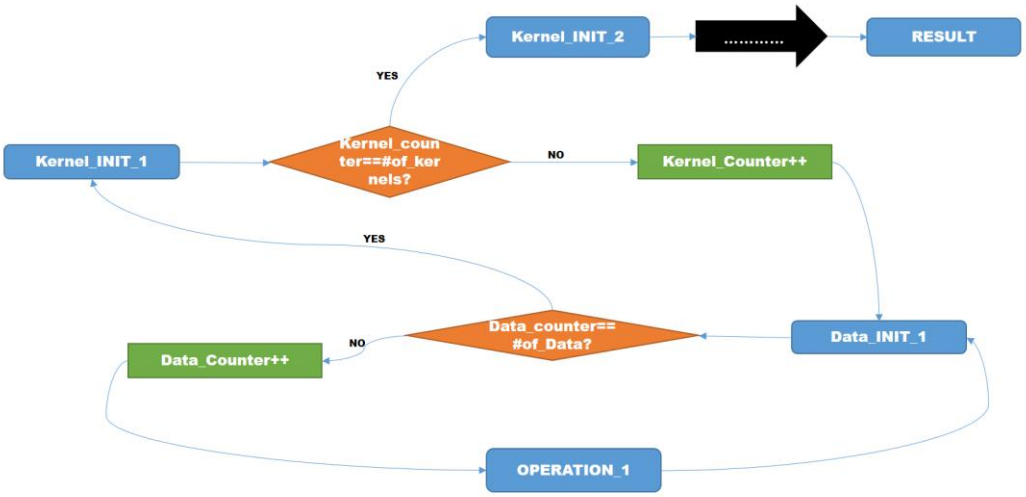

Figure 3.6: Algorithmic State Machine.

3.3.1 Microblaze

MicroBlaze is a 32-bit RISC-based soft processor core developed by Xilinx, designed to run on FPGAs. Its flexible and customizable architecture allows users to configure features such as cache, FPU, and MMU based on their needs. MicroBlaze is commonly used in real-time embedded systems, control units, and processor-based accelerators. It integrates seamlessly with Vivado and Vitis platforms, enabling C/C++-based development and facilitating hardware-software co-design.

For this work, this module is used for initiliazing the important parameters. Creating the states. Realizing the state transitions and especially directing the data path such that where the data will go or where data will come. In summary, whole hardware created on vivado platform is controlled by this virtual microprocessor. In other word, it sets to worl all hardware peripherals which are connected to it.

MicroBlaze communicates with peripherals primarily using the AXI (Advanced eXtensible Interface) protocols, such as AXI4-Lite for low-bandwidth control/status communication and AXI4 for high-bandwidth data transfers. These protocols allow for seamless data exchange with various IP cores, including UARTs, GPIOs, and custom hardware accelerators. The communication is managed through an interconnect fabric that links the processor to connected peripherals.

When handling AXI Stream (AXI4-Stream), MicroBlaze can interface with high-speed, continuous data streams, ideal for applications like data processing or custom IP accelerators that require unidirectional, burst-mode data transfer. AXI4-Stream

supports lightweight, packet-based data transmission without address phases, making it perfect for scenarios where consistent and efficient data flow is necessary, such as audio/video streaming or real-time data acquisition. This combination of AXI interfaces allows MicroBlaze to interact flexibly and efficiently with various types of peripherals and custom logic.

For this work, AXI4-Lite, AXI-4 and AXI-4 Stream protocols are used. Former two of them used for addressed based communication therefore they are slow compared to AXI-4 Stream which has a few parameters to communicate. Its block diagram can be seen in Figure 3.7.

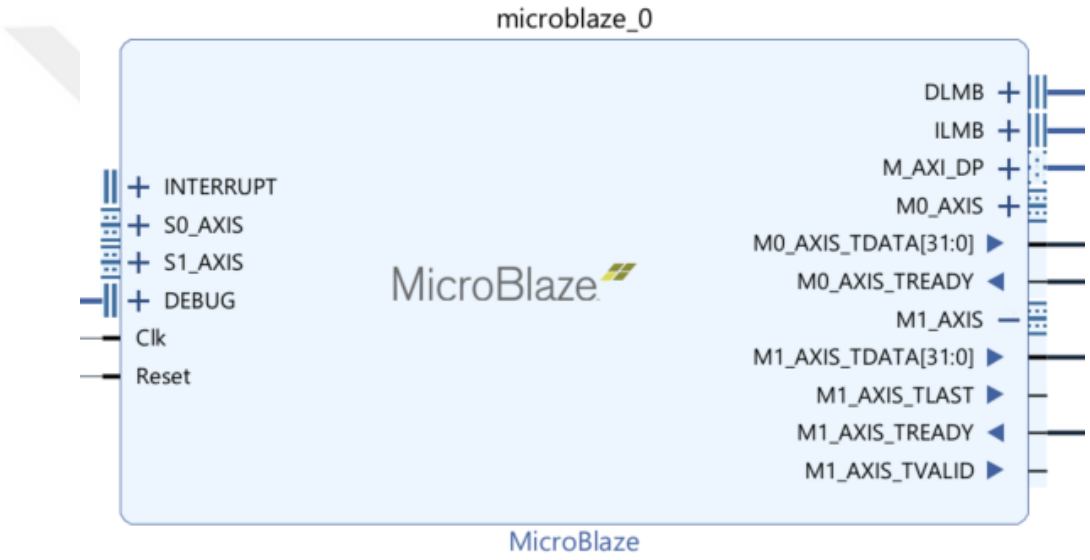

Figure 3.7: Block representation of MicroBlaze.

As can be seen from the figure 15, we have created two stream links (M0, M1, S0, S1) for controlling the operation of "max_pooling" and "Relu", a debug port, An "AXI Data Path(M_AXI_DP) for data communication with addressed peripherals, DLMB and ILMB for the storage of data and instructions to "block ram" (Interrupt pin is created but not used).

Additionally, when configuring the microblaze, optimization of the performance, enabling the barrel shift register and choosing the highest amount of Block ram size is done for minimizing the time of operation.

3.3.1.1 Debug Module:

The Debug Module allows developers to perform real-time debugging on the MicroBlaze processor, including setting breakpoints, single-stepping through code, and inspecting registers and memory.It connects through interfaces like JTAG, enabling interaction with development tools such as Xilinx's Vitis IDE or Vivado.This module is crucial for efficient debugging and testing, allowing developers to identify and fix issues in embedded software and hardware interaction.

3.3.1.2 Processor Reset System:

The Processor Reset System is responsible for handling reset operations for the MicroBlaze core and associated peripherals.

It ensures that the processor and connected components are initialized into a known state upon power-up or when a reset condition is triggered.This system supports various reset types, such as soft reset (keeping peripheral states intact) or hard reset (re-initializing all logic), ensuring system reliability and stability.These modules play vital roles in maintaining smooth development, testing, and robust operation in MicroBlaze-based FPGA designs.

3.3.2 Block RAM

Block RAM (BRAM) in FPGA designs refers to dedicated memory blocks within the FPGA fabric that can be used for data storage. When used with MicroBlaze, BRAM provides a fast, efficient memory option for storing instructions and data, contributing to the overall performance of the embedded system.

3.3.2.1 Key points about MicroBlaze and Block RAM

BRAM can be connected directly to the MicroBlaze processor as local memory, allowing low-latency access and fast data retrieval.

BRAM is often used to store the executable code (instruction memory) and data required by the MicroBlaze processor, enabling on-chip storage that avoids the delay of external memory access.

The size of BRAM allocated to MicroBlaze can be adjusted according to the requirements of the application, making it a flexible solution for memory management.

FPGA BRAM typically supports dual-port access, allowing simultaneous read and write operations or enabling multiple memory access paths, which can be advantageous for parallel data processing.

Using BRAM minimizes the access time compared to external memory options, enhancing the performance of time-critical applications by providing higher bandwidth and lower latency.

Overall, integrating Block RAM with MicroBlaze is ideal for embedded systems where fast, on-chip memory is essential for efficient data processing and execution. This block is very essential component in our work because since there are lots of data that are processed, we need a fast read and write operations to speed up the process and the fastest memory type available is BRAM. However, it comes with an trade-off. There is not enough size for BRAM to keep all of the weights that come from the trained CNN model. Therefore, the limited source of this fast speed memory should be used smartly.

In this work, BRAM is used to create single operation buffers. For example, if the size of the filters is 1x5 and depth of the input is 8, we need one buffer with the size of 40 single precision numbers for the data and correspondingly we need one buffer with the size of 40 single precision numbers for the weight. Those weights and data numbers are kept in DDR2 memory. However, to arrange the operation and providing the alignment of which are of those data and weights are multiplied by each other, those are separated by these buffers and kept temporarily. In Figure 3.8, the usage of these buffers are illustrated.

```
switch(state){
    case kernel_init_1:
        if(kernel_counter==32)
            {state=kernel_init_2;
            kernel_counter=0;}
        else
                {
                    XAxiCdma_Reset(&AxiCdma);
                    index_offset=FILTER_1_ADDRESS+160*kernel_counter;
                    status=XAxiCdma_SimpleTransfer(&AxiCdma,index_offset,(UINTPTR)Kernel_Buffer , 160,NULL,NULL);
                    while(XAxiCdma_IsBusy(&AxiCdma))
                    {}
                    kernel_counter=kernel_counter+1;
                    state=data_init_1;
                }
        break;

    case data_init_1:
        if(data_counter==1246)
            {kernel_init_done=0;
            state=kernel_init_1;
            data_counter=0;}
        else

              { XAxiCdma_Reset(&AxiCdma);
                index_offset=DDR_BASE_ADDRESS+32*data_counter;
                status=XAxiCdma_SimpleTransfer(&AxiCdma, index_offset,(UINTPTR)Data_Buffer , 160,NULL,NULL);
                data_counter=data_counter+1;
                state=operation_1;

              }
        break;
    case operation_1:
```

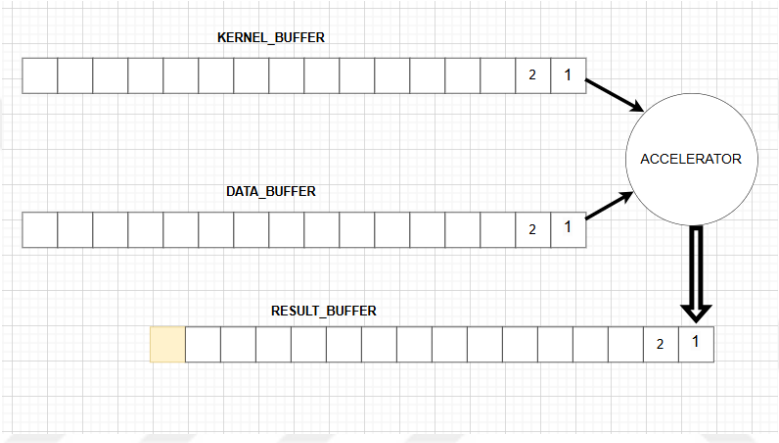

Figure 3.8: Buffer operation representation.

Until now, the essential parts that are almost always used with the microblaze processor is explained. Althougt they are common in most applications, configuration of those modules are unique to this application. Especially, for realizing the “DMA” operation, BRAM IPs are configured in an extraordinary way.

3.3.3 Central direct memory access (CDMA)

The Central Direct Memory Access (CDMA) controller in Vivado is a powerful IP core that facilitates efficient data transfer between memory-mapped peripherals or memory blocks within an FPGA design. CDMA is crucial for offloading data transfer tasks from the processor, thereby improving system performance and freeing up processing resources for other tasks.

3.3.3.1 Key features of CDMA in Vivado

High-Throughput Data Transfer: The CDMA controller supports high-speed data movement between different memory locations within the FPGA, reducing latency compared to software-based transfers.

Memory-to-Memory Transfer: CDMA enables the direct copying of data from one memory location to another without processor intervention. This can include BRAM to DDR, DDR to DDR, or any combination of connected memory blocks.

AXI Interface Compatibility: The CDMA controller is compatible with the AXI4 protocol, allowing it to interact seamlessly with other AXI-based components in the design. It can act as both an AXI master and slave.

Configurable Burst Transfers: The CDMA IP supports burst transfer operations, optimizing data movement for bulk transfers and improving overall bandwidth utilization.

Scatter-Gather Support: CDMA can be configured to handle scatter-gather operations through a linked list of memory descriptors, enabling complex data transfers without manual processor involvement.

Interrupt Generation: The IP can generate interrupts upon the completion of a data transfer, allowing the system to be notified and respond promptly.

3.3.3.2 Applications of CDMA

Data Streaming: Ideal for applications that require continuous data streaming from one memory space to another, such as video processing, image buffering, and real-time signal processing. Processor Offloading: Used in embedded systems to offload data transfer tasks from the main processor, enhancing multitasking capabilities and improving overall performance. Custom IP Integration: CDMA facilitates data movement between custom IP cores and external or internal memory blocks, making it suitable for complex, high-throughput data processing designs.

3.3.3.3 Best configurations for this work

Optimize Burst Sizes: Configure burst sizes according to the memory controller's capabilities to achieve maximum transfer efficiency. This feature is configured according to microblaze burst capacity and DDR2 memory burst capacity since CDMA works mainly between these two memories.

Use Scatter-Gather for Complex Transfers: For applications requiring non-contiguous data transfers, utilize scatter-gather mode to automate complex data movements. But for this application, this is not used since it contributes to complexity of the high level software that is created for this work and providing very small amount of performance.

DDR2 ram hold whole of the weights and input signal numbers, BRAM holds the single operand weights and operand data. And those operands are continually updated after each single operation. Therefore, main processing load of this work is these memory transportations from DDR2 Ram to BRAM. CDMA is fastest solution of these type operation. It is actually the memory to memory transportation version of normal DMA. It representation can be seen in Figure 3.9.

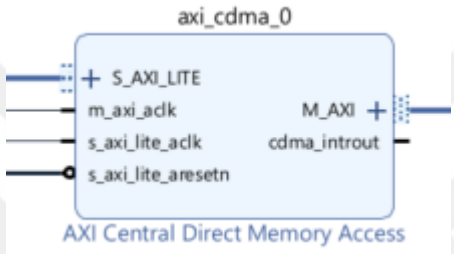

Figure 3.9: Block representation of CDMA.

This IP requires 3 main parameters one is the source address and the other is destination address and third one is how many bytes will be transferred. After the initiation of this transportation peripheral can do task independent from the MCU. Thus, processing system can continue its work after just the initiation of this process.

3.3.4 Direct memory access (DMA, memory to stream(M2S), stream to memory(S2M))

Direct Memory Access (DMA) is a method used in computing to transfer data directly between memory and peripherals or between different memory areas without involving the central processing unit (CPU) for each transfer operation. This capability enables high-speed data transfers and reduces CPU overhead, allowing the processor to perform other tasks concurrently. In FPGA designs, DMA controllers are integral for efficient data management, particularly in data-intensive applications like image processing, real-time signal analysis, and hardware accelerators.

In this work, DMA is utilized to transfer operand data from the Block RAM (BRAM) to the accelerator within the FPGA. The BRAM serves as a fast, on-chip memory that stores the input data needed by the accelerator. The DMA facilitates seamless and

efficient data transfer by reading operands from the BRAM and feeding them directly to the accelerator, minimizing latency and ensuring that the accelerator receives the data promptly for processing.

Once the accelerator has processed the input data, the DMA controller plays a crucial role in transferring the results back from the accelerator's output to the main memory. This output transfer is essential for storing the processed data where it can be accessed by other parts of the system or sent to external interfaces for further use or analysis. The combination of DMA and BRAM helps in optimizing the data flow within the FPGA, supporting high-throughput applications with minimal CPU intervention.

Therefore, in this work 3 DMA blocks are used one of them transfers “weights”, one of them transfers “data” and the third one transfers back the result from accelerator. Which means M2S and S2M operation directions are happened. Additionally, this block is addressed IP. In other words its registers can be configured from microblaze by AXI4 protocol. However, when sending or receiving the data it utilizes the AXI4 Stream protocol to speed up the data flow.

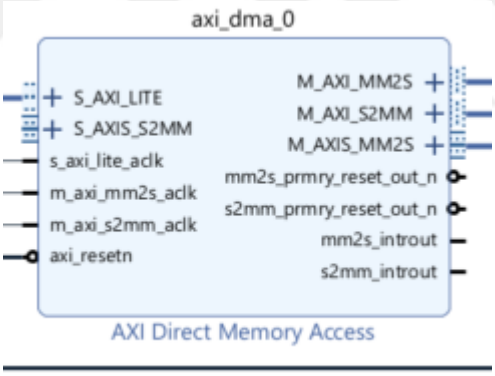

Figure 3.10: Block representation of DMA.

As can be seen from the Figure 3.10, left side of the pins are related to the AXI-4 protocol and the other side is related to data flow direction.

3.3.5 AXI4-Stream data FIFO

The AXI4-Stream Data FIFO is an IP core designed for buffering and managing data flow within FPGA designs that use the AXI4-Stream protocol. This FIFO (First-In-First-Out) component acts as a temporary storage buffer for data being transferred between different modules or IP blocks, helping to decouple producer and consumer rates. It ensures that data can be read and written at different clock rates or with varying

latencies, which is essential for maintaining data integrity and flow control in streaming applications. The AXI4-Stream Data FIFO supports features such as packet-based transfers, data width conversion, and configurable FIFO depths, making it highly adaptable for use in applications like data processing pipelines, video and audio streaming, and communication interfaces. By providing reliable storage and management of streaming data, the AXI4-Stream Data FIFO helps prevent data loss or underflow/overflow conditions, ensuring robust and efficient data handling in complex FPGA systems.

In this work, there are 3 FIFOs used for each one of the DMAs. In other word, for each one of the stream data paths which are weight, data and result data paths. The reason is that when for example 90 weights are multiplied with 90 data, each one of these numbers had to be aligned with respect to their index numbers such that first indexed weight multiplied with the first indexed data. However, this alignment is not provided by the DMA transferring since the initiation the DMA process is started from the software and two DMA can not be initiated exactly at the same time. Thus, DMAs transfer data from memory to FIFOs before these numbers go to accelerator. And the speed of DMA may not catch the speed of accelerator. For that reason, output of the accelerator is kept in FIFO before transferred by DMA to avoid data crash due to speed differences between different IPs.

Figure 3.11: Block representation of FIFO.

Of course, these FIFOs are created from limited memory of the FPGA. The depth and the width of the FIFOs can be configured. In our case, since the maximum number of data flow is 2144 and those are single precision numbers, the width is set 32 bit and depth is set to 2144. FIFO can be visualized as in Figure 3.11

Before explaining the most important module for this work, summarizing the dataflow is suitable by a schematic in Figure 3.12.

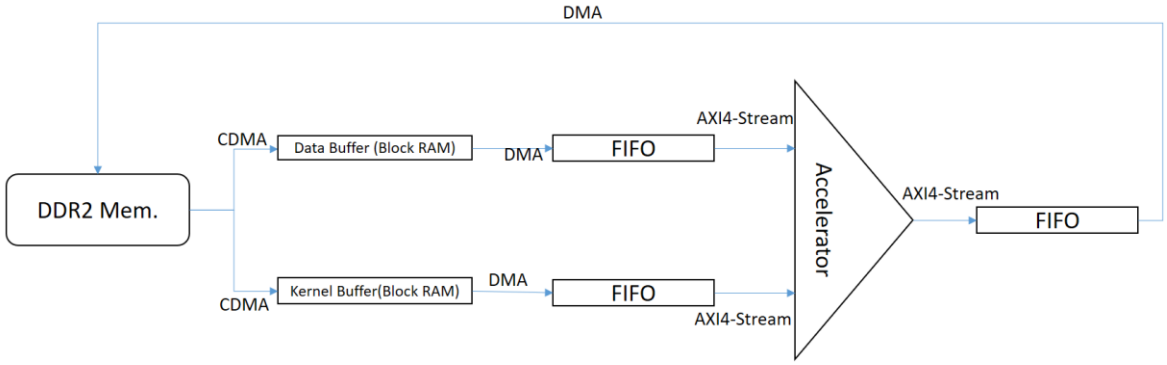

Figure 3.12: Block representation of Accelerator operation.

3.3.6 Accelerator (Arithmetic operation part)

Most important part of the custom hardware is the "Accelerator" part because all of the mathematical operations are done in this module. The mathematical operation that is needed is multiply and add operation to realize convolution operation. Additionally, all of the number are single precision floating numbers. Thus, these numbers cannot be operated by simple multiplication and operation blocks that are used by signed integer number.

For that reason, utilizing the DSPs of the FPGA is crucial. In Block design IP, there is an ip which is called "Floating_point" IP. All of the mathematical operations which are done on floating numbers can be realized by configuring this IP. Therefore, two IPs which derived from "Floating_point" IP is created. One is for multiplication purposes and the other one is for the accumulation the multiplied numbers. These two ips are connected in series. Also these IPs are configured in a way that provides maximum throughput. Therefore, trading from latency.

To speed up the accelerator (in Figure 3.13), many DSPs are used instead of 1 for the multiplication and summation operation because there are needed parameters which should be prepared at the same time. And these blocks are configured for high speed operation.

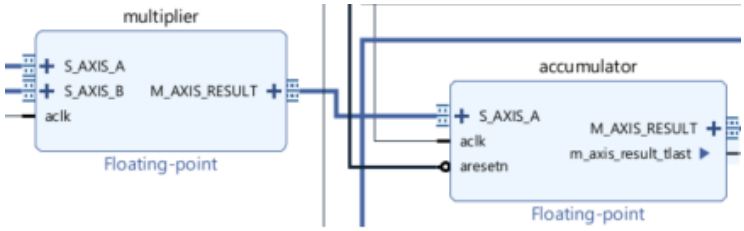

Figure 3.13: Block representation of Accelerator.

Actually, the common parts of the accelerator is explained. However, accelerator also includes “Relu” and “Max_Pooling” operations. Since “ReLu” operation is activated first 6 of the 7 layers and “Max_Pooling” operations are activated in first 4 of the 7 layers, those blocks are explained separately.

3.3.7 Accelerator (ReLu part)

There is no available IP block which can realize “ReLu” operation in vivado IP catalogue. Therefore, a custom IP that can realize this operation is created by using VHDL language. After coding in VHDL, packaging this piece of hardware makes a custom IP for handling this operation. The code written in VHDL is available in appendix?. “Accumulator” block is directly connected to “ReLu” custom IP. However, since this block should not be activated for the last 3 layers of the 7, A signal which is called “ByPass” signal added to deactivate the “ReLu” operation.

When “ByPass” signal is at value “0x11111111” , “ReLu” block simply bypassing the input data to output. And when “ByPass” signal is “0x00000000”, the IP do the supposed operation. In vitis platform, Activation and deactivation of this IP is controlled by using this signal very easily and can be seen in Figure 3.14.

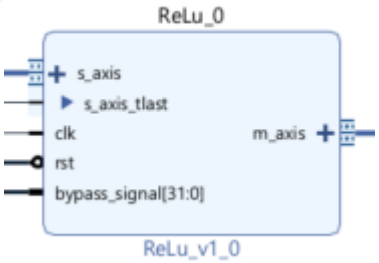

Figure 3.14: Block representation of ReLu operator.

One additional point to mention is that since arithmetic part of the accelator send and receive the data by using AXI4-Stream protocol, “ReLu” IP is created in that protocol.

3.3.8 Accelerator (Max_Pooling part)

In the first 4 of 7 layers, “Max_Pooling” layer exists after the convolutional layer. Therefore, there should be some hardware which can realize the operation of the “Max_Pooling” layer and after some layer, it should be deactivated.

For that purpose, By following AXI-4 Stream protocol, A Hardware is created in VHDL and packaged for integration with other blocks. Like “Relu” IP, a “ByPass” signal is added to IP to control it enabling and disenabling in vitis platform.

The Stride of the “Max_Pooling” layers in this model is 2. Therefore the code waits for 2 consecutive data to give output. Since the arithmetic part of the accelerator does give correct result until the end of the stream, “Max_Pooling”(Figure 3.15) layer waits to take input until the end of the stream.

Figure 3.15: Block representation of Max_pooling operator.

3.3.9 AXI Interconnect

The AXI Interconnect is a vital IP component in FPGA designs that facilitates communication between multiple AXI-based master and slave devices. Acting as a multiport switch or hub, the AXI Interconnect manages data traffic, ensuring that data can be efficiently routed between connected components within the system. It supports various AXI protocols, including AXI4, AXI4-Lite, and AXI4-Stream, allowing for both high-bandwidth data transfers and simpler control/status signal exchanges.

The AXI Interconnect handles multiple masters and slaves by implementing arbitration and routing logic, ensuring that requests are managed fairly and data paths are directed correctly. It can perform functions like address decoding, data width conversion, and protocol adaptation, making it versatile for complex system-on-chip (SoC) designs where different IP cores need to communicate seamlessly. With support for configurable data paths, clock domain crossings, and burst transfers, the AXI Interconnect helps maintain system efficiency and flexibility, allowing designers to scale and optimize their data flow architecture for specific performance requirements.

Actually main purpose of this IP is the fact that since Microblaze has only single data path port, connection of the all peripherals to this port is not feasible. Therefore, AXI Interconnect provides the connection and communication of microblaze and other peripherals. In this study, 2 “AXI Interconnect” IPs are used. First is used to provide connection of all the DMAs, CDMA, another AXI Interconnect, and UART. The second is used to connection between memory parts with each other and more critically connection these memory parts with the microblaze.

Actually, one of the trickiest part of this work was providing connection of those memory parts with each other at the same time connection of those with microblaze. And one AXI Interconnect connects DDR2 memory with the BRAM and the other AXI Interconnect connects those two memory parts with the microblaze.

The actual reason why 2 of these blocks are used is the fact that since the CDMA needed to data transferring between two memory parts it should know where these two memories are located however it cannot be connected to same AXI interconnect with these memories since after then no connection pins are left to communicate with microblaze. It can be visualized in Figure 3.16. Also this was the last piece of the puzzle to create the hardware. Therefore, whole hardware (Figure 3.17) can be created now and modules can be addressed(Figure 3.18).

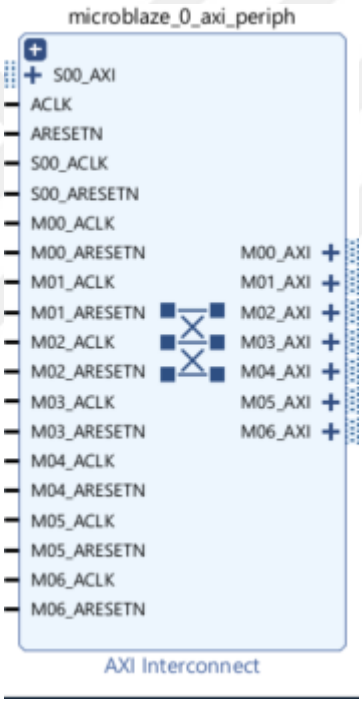

Figure 3.16: Block representation of AXI Interconnect.

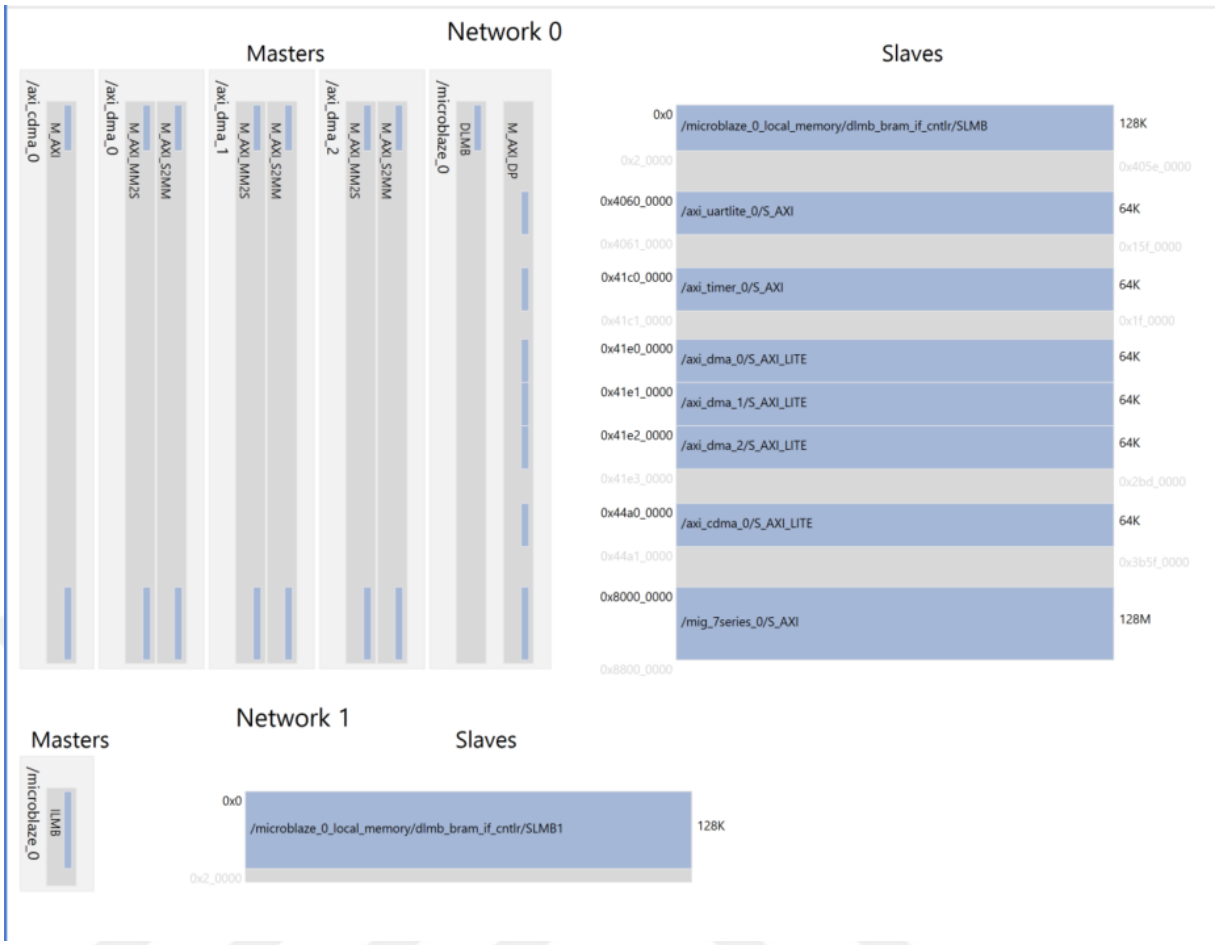

Figure 3.17: Addressing results of peripherals.

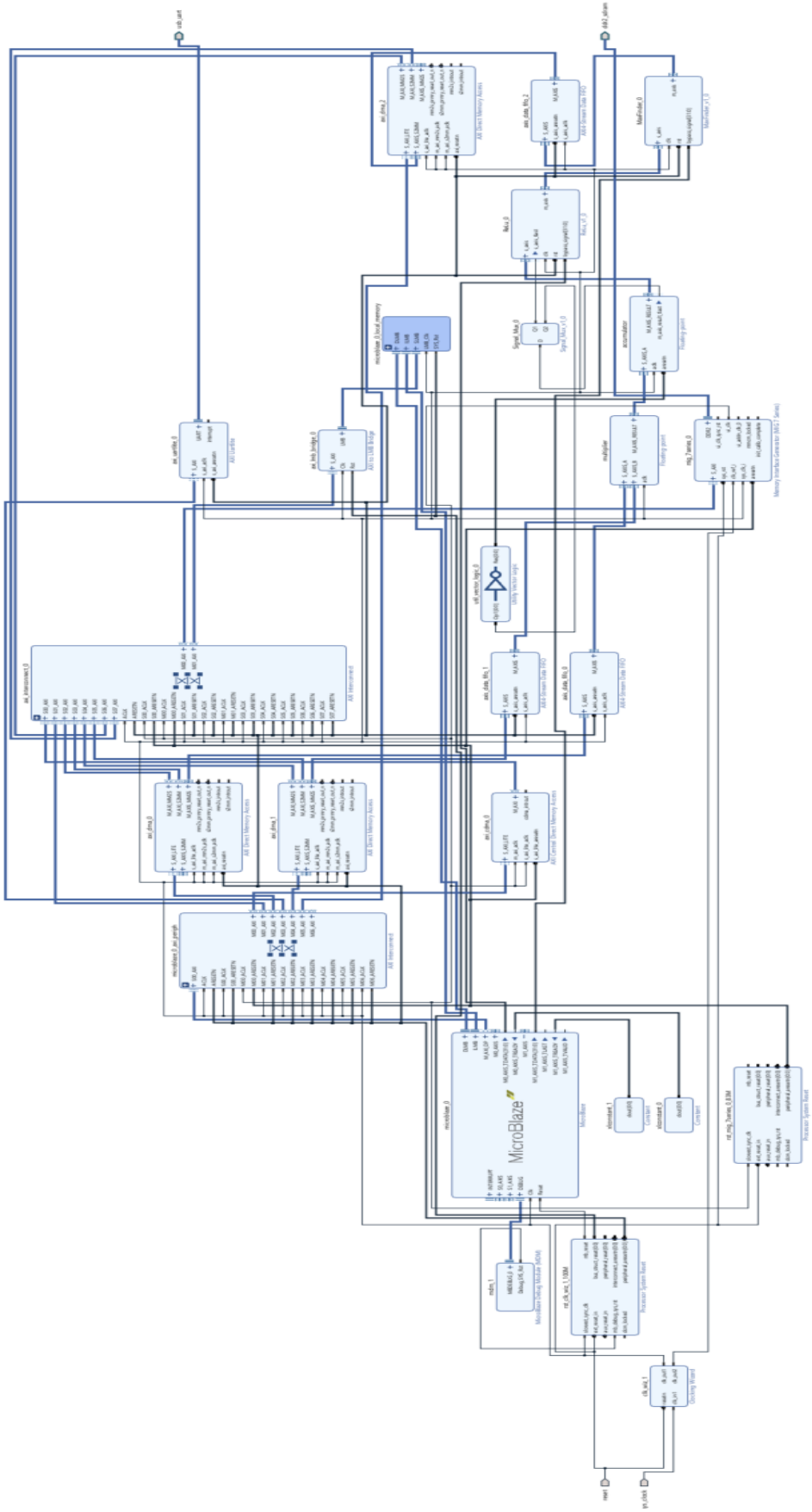

Figure 3.18: whole hardware block diagram.

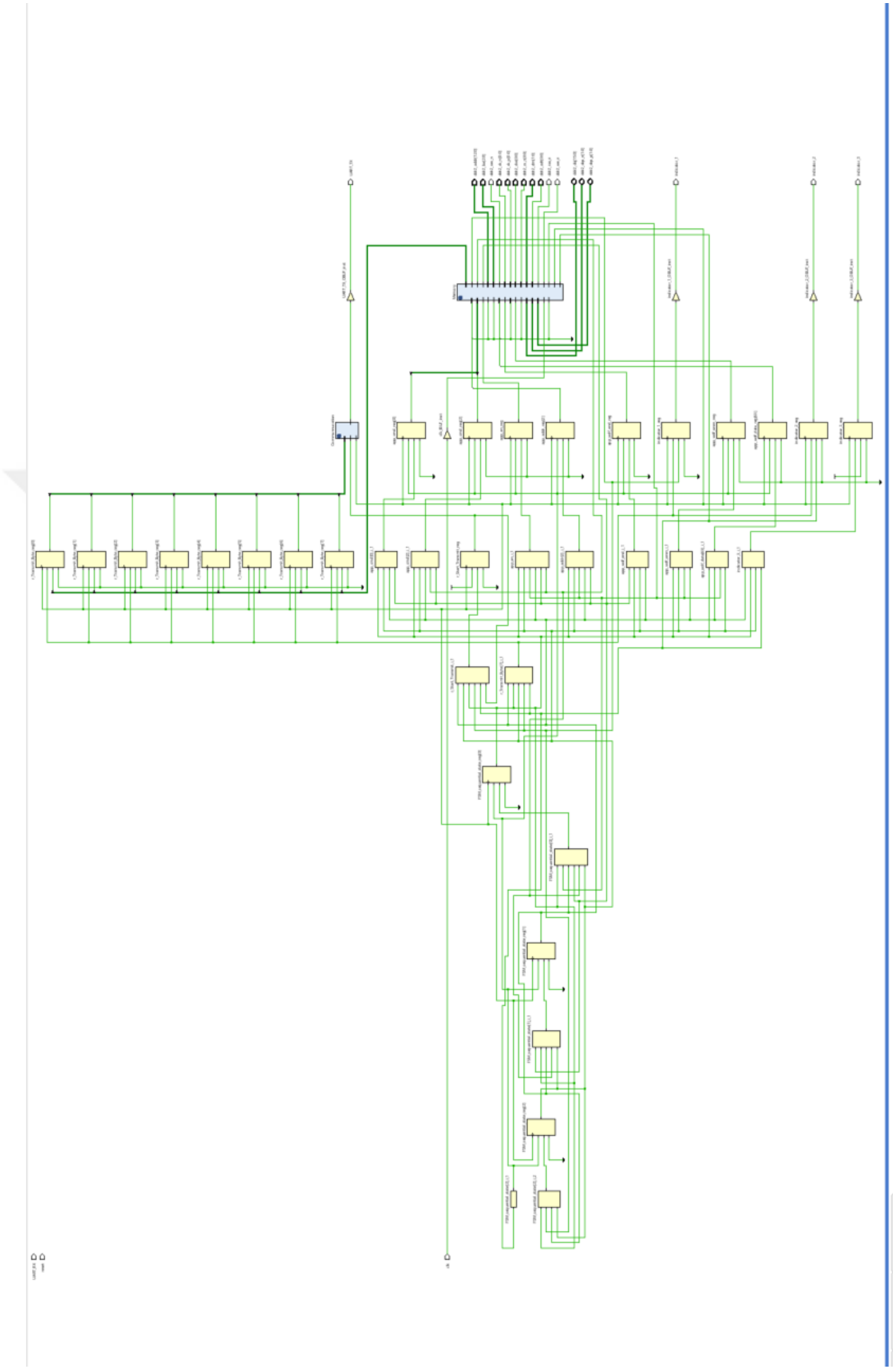

Figure 3.19: UART and DDR2 module implementation results.

4. EXPERIMENTAL RESULTS

4.1 Simulation Results

The aim of simulation is proofing the concept that we have thought about. The assumptions that is made during the simulation is not always applicable in hardware implementations. Therefore, it was enough for this part to see that convolution operation is working and data streaming between the layers are correct.

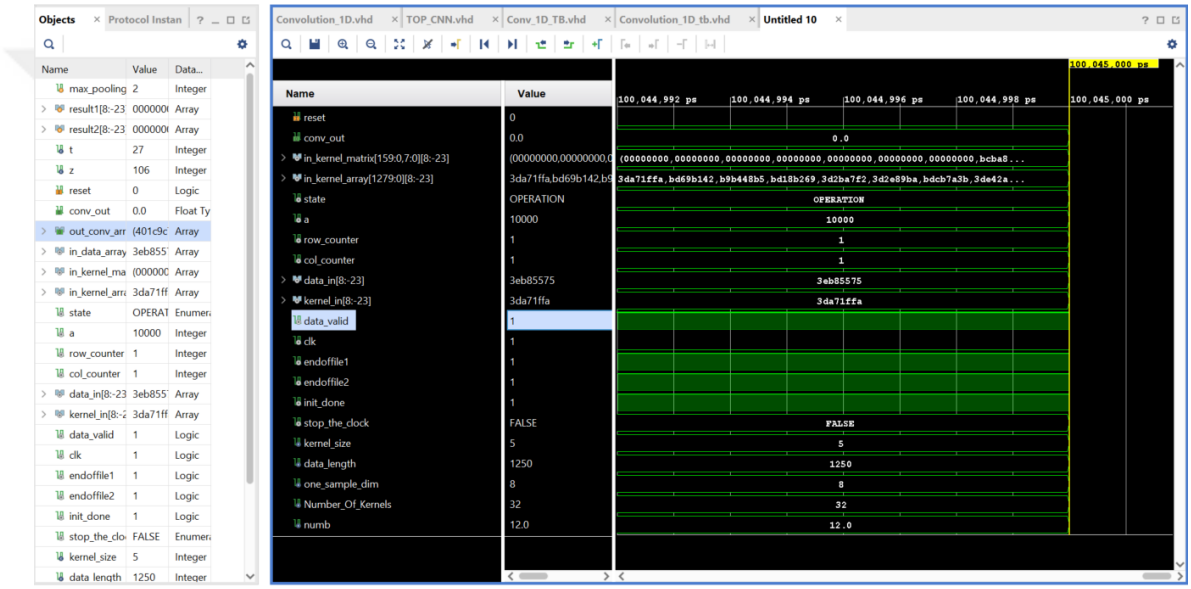

Figure 4.1: Simulation view.

From the simulations that we have made during the study period. The algorithm that we created is working and gives the same result as if it is in the python environment. After obtaining simulation results, we can proceed to hardware implementation. The results can be seen in Figure 4.1.

Two most important assumptions that we make during the simulation will be bottlenecks in the hardware implementation. Those are parallel data streaming between layers. Second will be the assumption of limitless multiplication operation. However, since DSP number in FPGA is limited this will not be case for the hardware implementation. Thus, we will have to limit the number of summation and multiplication in the next step.

4.2 Hardware Results

The custom hardware which appears at the end of the work is tested and results are corrected by the results obtained from python environment where the original model is trained and tested. However, this hardware is not the only testing material. In other words, this hardware is created after lots of process.

At the first stage only microblaze and some specific IPs existed. There was no any DMAs or Accelerators to speed up the operation. After then, DMAs starts to be added to hardware. First the CDMA and later other DMAs with the Accelerator part. Thus, Testing part will divided into four stages to exhibit improvement of addition of the special hardwares.

One important point is that the sample data and weights are downloaded to the FPGA by using UART. When the serial data comes into FPGA, hardware starts to record each data to DDR2 memory. After finishing data transferring, High level code created on Vitis platform automatically initiates the prediction operation of the sample data.

Additionally, the prediction time is calculated between the start of the “state machine” and final score points. Thus, we exclude the time of sample data download and some small configuration code executions. Unfortunately, we have bounded by UART baudrate limitations to speed up the downloading process although the baudrate is set to be the maximum. It takes about 2 minutes and 20 seconds to complete downloading. However, once the weights are downloaded, very small time required to refresh testing data like 2.2 second.

Finally, all stages are tested with the first training input(X_train[0]) and the scores obtained are corrected accordingly. And when calculating the time between the start and end of the execution. A peripheral which is called “AXI Timer” is used. This timer (Figure 4.2) is started with the state machine execution at the same time and after obtaining the score it immediately stops and records its current value. The count that timer stores is corresponds to clock thick number. In other words, how many cycle of clock wave passed. Therefore, every thick corresponds to period of the clock cycle which is 0.01nano second in last implementation.

Figure 4.2: Block representation of AXI Timer.

The numbers that are at the first column of the Table 4.1 and table 4.2 are correspond to “Only microblaze usage”, “Microblaze and added DMA dataflow”, “Microblaze added DMA and added accelerator” and “ Microblaze added DMA and added accelerator and ReLu on hardware”

4.2.1 Only microblaze usage

In this trial, no DMAs are used. And there was no accelerator. Data and kernel values are extracted from DDR2 memory with microprocessor and arithmetical calculations are also made on microprocessor. The aim of the trial is showing the improvement between an implementation that is based on fully software and implementation based on both hardware and software. This model takes 6 minutes and 32 seconds to complete its execution.

4.2.2 Microblaze and added DMA dataflow

In this trial, CDMAs are added for initializing the operand buffers. However, no accelerator or streaming DMAs are used. By this way, data flow is speeded up. On the other hand, this result is still lack of performance for being used for industrial purposes. Without accelerator and without pipelining, the prediction time is about 1 minutes and 50.80 second.

4.2.3 Microblaze added DMA and added accelerator

In this trial, Accelerator comes into play. Its contribution to timing performance is huge. In this configuration Relu and Max Pooling operations are still in software implementation.

scores	float [3]	[10.84972, ...
[0]	float	10.84972
[1]	float	-4.767785
[2]	float	-6.607886
current_count	uint32_t	0x23a7b64a

Figure 4.3: Debugging screen of Vitis.

The number in Figure 4.3, 0x23a7b64a is corresponds to time elapsed during the execution. It corresponds to 598193738 in decimal base. Which means it takes 5.98 second to complete the process by designed hardware.

4.2.4 Microblaze added DMA and added accelerator with ReLu on hardware

Final hardware can make the prediction of the inputs at 5.0587 seconds. From the Figure 4.4, it can be seen that time it takes to complete the operation is the time correspondence of hexadecimal value(1e271462) and this number is 505877602 on decimal base. And since the clock which is connected to AXI_Timer has clock frequency of 100MHz, its period is 0.01 nano second. When the thick number and period is multiplied 5.0587 second time is obtained as a result. It means it takes 5.0587 seconds to complete execution.

scores	float [3]	[10.84972, -4.7677...
[0]	float	10.84972
[1]	float	-4.767785
[2]	float	-6.607886
ddr_ptr[16777224]	volatile uint8_t	'\000'
current_count	uint32_t	0x1e271462

Figure 4.4: Debugging screen of Vitis.

These results are exactly matched with the results obtained from python (Figure 4.6) however since no softmax activation is added to the hardware, the results are corrected with the online softmax calculator (Figure 4.5).

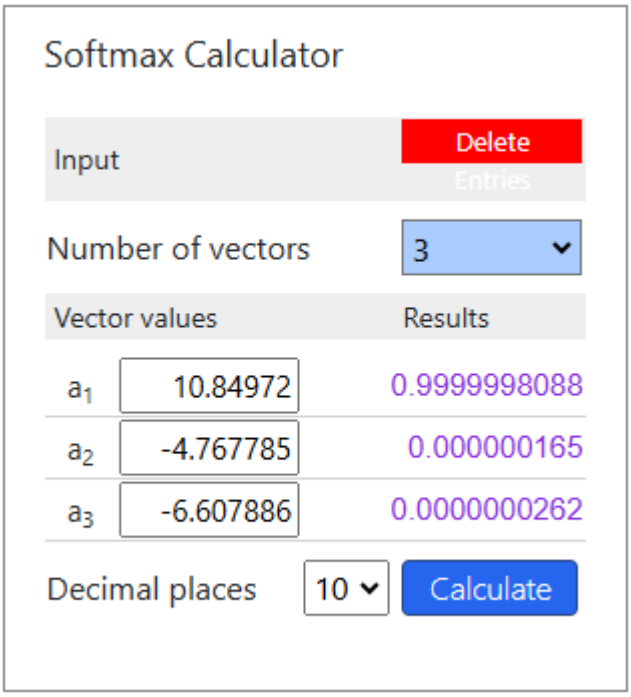

Figure 4.5: Softmax calculator results.

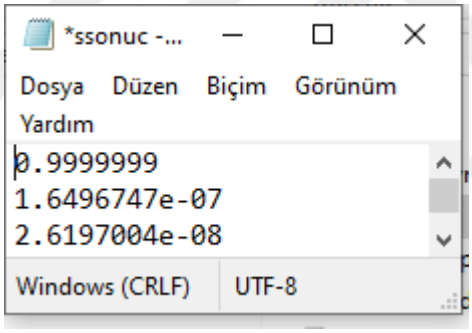

Figure 4.6: Results obtained from python.

1. Slice Logic

Site Type	Used	Fixed	Prohibited	Available	Util%
Slice LUTs	23712	0	0	63400	37.40
LUT as Logic	21716	0	0	63400	34.25
LUT as Memory	1996	0	0	19000	10.51
LUT as Distributed RAM	1060	0			
LUT as Shift Register	936	0			
Slice Registers	25976	2	0	126800	20.49
Register as Flip Flop	25962	2	0	126800	20.47
Register as Latch	0	0	0	126800	0.00
Register as AND/OR	14	0	0	126800	0.01
F7 Muxes	423	0	0	31700	1.33
F8 Muxes	2	0	0	15850	0.01

3. Memory

Site Type	Used	Fixed	Prohibited	Available	Util%
Block RAM Tile	47.5	0	0	135	35.19
RAMB36/FIFO*	47	0	0	135	34.81
RAMB36E1 only	47				
RAMB18	1	0	0	270	0.37
RAMB18E1 only	1				

4. DSP

Site Type	Used	Fixed	Prohibited	Available	Util%
DSPs	16	0	0	240	6.67
DSP48E1 only	16				

Figure 4.7: Implementation results of the hardware resources for last trial.

As can be seen from the Figure 4.7, the resources of the used FPGA is sufficient to implement this hardware. %37.40 of LUT cells and %20.49 of registers are used. Also most of the Block RAM is not used. Additionally, very small percentage of DSPs are implemented in this design.

Table 4.1: Summary of the trial results.

	LUT	FF	DSP	Fmax (MHz)	#ofclock CYCLE	LATENCY (s)	POWER (W)	ENERGY (J)
1	5914	5872	0	160	62800000000	392.5000	1.242	487.48
2	8949	10537	0	154	44758560000	110.8000	1.246	235.14
3	23648	25892	16	100	598193738	5.9819	1.328	7.94
4	23712	25962	16	100	505877602	5.0587	1.339	6.76

According to Table 4.1, best time performance belongs to 4'th trial which makes operation mostly in hardware at the expense of very little increase in wattage compared to the other trials. However, when comparing the resources, 4'th and 3'rd trials use much more FPGA resources compared to 1'st and 2'nd trials. Although, maximum synchronous frequency that can be used for the digital circuits is high in 1'st and 2'nd trial, it does not make 1'st and 2'nd implementation faster than 3'rd and 4'th trial. Additionally, 1'st and 2'nd implementation may consume little less power compared to 3'rd and 4'th trial. However, when calculating total energy used, 3'rd and 4'th implementation consumes much less energy compared 1'st and 2'nd implementation. Finally, implementing "ReLu" operation in hardware rather than software, provides nearly 1 second time gain compared to 3'rd trial.

Table 4.2: Latency, Area, and Energy Trade-offs Analysis.

	Area	Latency(s)	Energy(J)	Decrease in latency(%)	Increase in area(%)	Decrease in energy(%)
1	0.1395	392.5000	487.48	0	0	0
2	0.2242	110.8000	235.14	254.22	60.71	107.36
3	0.6438	5.9819	7.94	6461.67	361.50	6040.70
4	0.6454	5.0587	6.76	7660.74	362.65	7113.17

Table 4.2 compares the performance of four different custom hardware trials in terms of latency, area, and energy consumption. The first hardware serves as the reference, and percentage changes in subsequent trials are evaluated. In this analysis, the decrease in latency is recalculated relative to the current latency, resulting in significantly higher percentages due to the dramatic reductions observed in latency.

In the first trial, the latency is 392.5 seconds, energy consumption is 487.48 joules, and area usage is 0.1395 units. This serves as the baseline for comparison.

The second trial reduces latency to 110.8 seconds, marking a 254.22% decrease relative to current latency, while energy consumption drops by 107.36% and area usage increases by 60.71%. Here, the moderate increase in area is negligible when compared to the substantial improvements in latency and energy efficiency.

In the third trial, latency decreases drastically to 5.9819 seconds (a 6461.67% reduction) while energy consumption drops to 7.94 joules. However, the area usage increases by 361.50%. Notably, the increase in area remains very small when compared to the immense reduction in latency. This demonstrates that the gains in speed and energy efficiency far outweigh the additional area cost, highlighting the trial's remarkable performance.

The fourth trial further reduces latency to 5.0587 seconds, achieving a 7660.74% reduction in latency and 98.61% energy savings, while area usage increases by 362.65%. Similar to the third trial, the increase in area is minimal relative to the tremendous improvements in latency. The nearly identical area usage between the third and fourth trials underscores that the additional performance gain comes at no significant extra area cost, making this trial the most efficient in terms of both speed and energy.

Overall, the results reveal a clear trend: while area usage increases significantly in the third and fourth trials, the improvements in latency and energy consumption are so dramatic that the area trade-off becomes negligible. The third and fourth trials demonstrate that even substantial area increases are minor when compared to the massive reductions in latency and energy. This makes the latter two trials ideal for applications prioritizing performance and efficiency over area constraints.

5. CONCLUSION

In conclusion, this thesis presents a comprehensive study on the development and implementation of an Artificial Neural Network (ANN)-based fault diagnosis system for electrical machines using Field-Programmable Gate Arrays (FPGAs). The research underscores the critical role of reliable fault detection systems in enhancing the operational efficiency and longevity of electrical machines, which are vital components in various industrial applications.

The integration of deep learning techniques, particularly Convolutional Neural Networks (CNNs), with the high-speed and parallel processing capabilities of FPGAs, has demonstrated significant improvements in real-time fault detection and diagnosis. The proposed system leverages the powerful data processing capabilities of CNNs to analyze vibration signals and accurately classify different types of faults. By implementing the CNN model on an FPGA platform, the system achieves low latency, high throughput, and robust performance, making it suitable for real-time industrial applications.

Throughout this research, several key contributions have been made. First, the study highlights the effectiveness of using 1D CNNs for fault detection in electrical machines. The CNN model developed in this work has shown high accuracy in classifying faults from the MAFAULDA dataset, which consists of various fault conditions simulated under controlled environments. The use of mobile accelerometers to collect data further emphasizes the practicality and cost-effectiveness of the proposed approach.

Second, the implementation of the CNN model on an FPGA has been meticulously detailed. The VHDL code developed for the Convolution_1D, Convolution_1D_No_MP, Convolution_1D_Middle, and Dense layers has been optimized to ensure efficient utilization of FPGA resources. The real-time processing capabilities of the FPGA have been successfully demonstrated using the Nexys A7 development board, highlighting the feasibility of deploying such systems in real-world industrial settings.

Moreover, the study underscores the importance of data pre-processing techniques, such as normalization and Fast Fourier Transform (FFT), in enhancing the performance of the CNN model. These techniques help in reducing the complexity of the dataset and extracting meaningful features, which are crucial for accurate fault detection.

The results of this research indicate that the proposed ANN-based fault diagnosis system can significantly improve the reliability and efficiency of electrical machines. The real-time fault detection capabilities of the system enable proactive maintenance strategies, reducing unexpected downtimes and maintenance costs. Additionally, the adaptability of the CNN model to different fault conditions and datasets suggests its potential for broader applications in various industrial domains.

Future work will focus on extending the proposed method to include a wider range of fault types and exploring its application in other areas of industrial automation. Further optimization of the VHDL implementation and the integration of additional sensor data will be pursued to enhance the system's robustness and accuracy. This research contributes to the advancement of intelligent maintenance systems and sets the stage for more innovative applications of machine learning and FPGA technologies in industrial fault diagnosis

REFERENCES

[1] Peesapati, Venkat, Bhavana Sammeta, and Maheshwar Rao Podile. "Methods for Advanced Wind Turbine Condition Monitoring and Early Diagnosis: A Literature Review." Energies 14, no. 22 (2021): 7459. https://www.mdpi.com/1996-073/14/22/7459.

[2] Carvalho, Fernando R., Caio G. Pantoja, and Peter H. N. Salas. "Autonomous Detection of Electrical Machine Faults Using Deep Learning: A Review." Energies 16, no. 17 (2023): 6345. https://www.mdpi.com/1996-1073/16/17/6345.

[3] Singh, Arun Kumar, and R. N. Yadav. "Predictive Maintenance Using Machine Learning: Trends and Challenges." Energies 16, no. 15 (2023): 5602. https://www.mdpi.com/1996-1073/16/15/5602.

[4] Kim, Junyeong, et al. "Deep Learning-Based Fault Diagnosis for Electrical Machines: A Comprehensive Review." arXiv preprint arXiv:2006.04278 (2020). https://arxiv.org/abs/2006.04278.

[5] Kudelina, Karolina, Bilal Asad, Toomas Vaimann, Anton Rassõlkin, Ants Kallaste, and Huynh Van Khang. "Methods of Condition Monitoring and Fault Detection for Electrical Machines." Energies 14, no. 22 (2021): 7459. https://doi.org/10.3390/en14227459.

[6] Akbar, Siddique, Toomas Vaimann, Bilal Asad, Ants Kallaste, Muhammad Usman Sardar, and Karolina Kudelina. "State-of-the-Art Techniques for Fault Diagnosis in Electrical Machines: Advancements and Future Directions." Energies 16, no. 17 (2023): 6345. https://doi.org/10.3390/en16176345.

[7] Gultekin, Muhammed Ali, and Ali Bazzi. "Review of Fault Detection and Diagnosis Techniques for AC Motor Drives." Energies 16, no. 15 (2023): 5602. https://doi.org/10.3390/en16155602.

[8] Magar, Rishikesh, Lalit Ghule, Junhan Li, Yang Zhao, and Amir Barati Farimani. "FaultNet: A Deep Convolutional Neural Network for Bearing Fault Classification." arXiv preprint arXiv:2010.02146 (2020). https://doi.org/10.48550/arXiv.2010.02146.

[9] Tang, Wensi, Guodong Long, Lu Liu, Tianyi Zhou, Michael Blumenstein, and Jing Jiang. "Omni-Scale CNNs: A Simple and Effective Kernel Size Configuration for Time Series Classification." arXiv preprint arXiv:2002.10061 (2020). https://arxiv.org/abs/2002.10061.

[10] Singh, Anurag, and Deepak Gupta. "Advances in Data-Driven Fault Diagnosis for Electrical Machines: A Comprehensive Review." SN Computer Science 5, no. 2 (2023): 2159. https://doi.org/10.1007/s42979-023-02159-4.

[11] Taherkhani, A., Cosma, G. & McGinnity, T.M. A Deep Convolutional Neural Network for Time Series Classification with Intermediate Targets. SN COMPUT. SCI. 4, 832 (2023). https://doi.org/10.1007/s42979-023-02159-4

[12] Gu, Yufeng, Yongji Zhang, Mingrui Yang, and Chengshan Li. "Motor On-Line Fault Diagnosis Method Research Based on 1D-CNN and Multi-Sensor Information." Applied Sciences 13, no. 7 (2023): 4192. https://doi.org/10.3390/app13074192.

[13] Ertarğın, Merve, Turan Gürgenç, Özal Yıldırım, and Ahmet Orhan. "A Deep Learning Approach for Motor Fault Detection using Mobile Accelerometer Data." European Journal of Technique 13, no. 2 (2023). https://dergipark.org.tr/en/pub/ejt.

[14] Morenas, Javier de las, Francisco Moya-Fernández, and Julio Alberto López-Gómez. "The Edge Application of Machine Learning Techniques for Fault Diagnosis in Electrical Machines." Sensors 23, no. 5 (2023): 2649. https://doi.org/10.3390/s23052649.

[15] Manar Abdelmaksoud, Marwan Torki, Mohamed El-Habrouk, Medhat Elgeneidy, Convolutional-neural-network-based multi-signals fault diagnosis of induction motor using single and multi-channels datasets,Alexandria Engineering Journal,Volume 73, 2023,Pages 231-248, ISSN 1110-0168, https://doi.org/10.1016/j.aej.2023.04.053.

[16] Xie, Yu, He Chen, Yin Zhuang, and Yizhuang Xie. "Fault Classification and Diagnosis Approach Using FFT-CNN for FPGA-Based CORDIC Processor." Electronics 13, no. 1 (2024): 72. https://doi.org/10.3390/electronics13010072.

[17] Wu, Jian, et al. "A Real-Time Motor Fault Detection System Based on FPGA." IEEE Transactions on Industrial Electronics 67, no. 5 (2020): 3692-3702. https://ieeexplore.ieee.org/document/8696298.

[18] Li, Jian, et al. "A Flexible CNN Architecture for Real-Time FPGA Implementation." Journal of Vibration and Control 26, no. 17-18 (2020): 1461-1471. https://www.sciencedirect.com/science/article/pii/S0141933120305402.

[19] Kumar, Prashant, and Ananda Shankar Hati. "Convolutional Neural Network with Batch Normalisation for Fault Detection in Squirrel Cage Induction Motor." IET Power Electronics 15, no. 1 (2021): 39-50. https://doi.org/10.1049/elp2.12005.

[20] Liu, Qin, Tian Liang, Zhen Huang, and Venkata Dinavahi. "Real-Time FPGA-Based Hardware Neural Network for Fault Detection and Isolation in More Electric Aircraft." IEEE Access (2019). https://doi.org/10.1109/ACCESS.2019.2950918.

[21] Syed, Rizwan Tariq, Yanhua Zhao, Junchao Chen, Marko Andjelkovic, Markus Ulbricht, and Milos Krstic. "FPGA Implementation of a Fault-Tolerant Fused and Branched CNN Accelerator With Reconfigurable Capabilities." IEEE Access (2024). https://doi.org/10.1109/ACCESS.2024.3392240.

[22] Osornio-Rios, Roque Alfredo, Isaias Cueva-Perez, Alvaro Ivan Alvarado-Hernandez, Larisa Dunai, Israel Zamudio-Ramirez, and Jose Alfonso Antonino-Daviu. "FPGA-Microprocessor Based Sensor for Faults Detection in Induction Motors

Using Time-Frequency and Machine Learning Methods." Sensors 24, no. 8 (2024): 2653. https://doi.org/10.3390/s24082653.

[23] Karim, E., Tayab Memon, and Imtiaz Hussain. "FPGA based on-line fault diagnostic of induction motors using electrical signature analysis." International Journal of Information Technology 11, no. 2 (2018). https://doi.org/10.1007/s41870-018-0238-5.

[24] İ. Palit, *FPGA Tabanlı Hata Tespit ve Sınıflandırma Sistemi Tasarımı*, Yüksek Lisans Tezi, 2024.

APPENDICES

APPENDIX A: The code piece that is used in simulation purposes

APPENDIX B: The vitis code which is created for driving the hardware

APPENDIX A:

Some of the TOP_CNN Layer VHDL code for this study.

```
library IEEE_proposed;
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.NUMERIC_STD.ALL;
--use IEEE_proposed.float_pkg.all;

package bus_multiplexer_pkg is
type conv_result is array(natural range <>,natural range <>) of real;
end package;

library IEEE;
library IEEE_proposed;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.NUMERIC_STD.ALL;
--use IEEE_proposed.float_pkg.all;
use work.bus_multiplexer_pkg.all;



entity TOP_CNN is
generic(
Number_of_layers : integer := 7;

Layer_1_kernel_size : integer := 5;
Layer_1_data_length : integer := 1250;
Layer_1_one_sample_dim: integer := 8;
Layer_1_Number_Of_Kernels: integer := 32;

Layer_2_kernel_size : integer := 3;
Layer_2_data_length : integer := 623;
Layer_2_one_sample_dim: integer := 32;
Layer_2_Number_Of_Kernels: integer := 64;

Layer_3_kernel_size : integer := 5;
Layer_3_data_length : integer := 310;
Layer_3_one_sample_dim: integer := 64;
Layer_3_Number_Of_Kernels: integer := 128;

Layer_4_kernel_size : integer := 3;
Layer_4_data_length : integer := 153;
Layer_4_one_sample_dim: integer := 128;
Layer_4_Number_Of_Kernels: integer := 256;
```

```
Layer_5_kernel_size : integer := 7;
Layer_5_data_length : integer := 75;
Layer_5_one_sample_dim: integer := 256;
Layer_5_Number_Of_Kernels: integer := 256;

Layer_6_kernel_size : integer := 3;
Layer_6_data_length : integer := 69;
Layer_6_one_sample_dim: integer := 256;
Layer_6_Number_Of_Kernels: integer := 32;

Kernel_Hor_Dim : integer  :=2144;
Kernel_Ver_Dim : integer  := 3;
Number_of_Classes: integer  :=3;
data_length : integer:=67 ;
one_sample_dim: integer:=32);

Port (

clk : in std_logic := '0';
reset : in std_logic :='0';
init_done  : in std_logic := '0';
data_valid : in std_logic:='0';
data_in : in real :=0.0;
kernel_in : in real :=0.0;
kernel2_in : in real :=0.0;
kernel3_in : in real :=0.0;
kernel4_in : in real :=0.0;
kernel5_in : in real :=0.0;
kernel6_in : in real :=0.0;
kernel7_in : in real :=0.0;
conv_out : out real :=0.0
-- out_conv_array_2 : out conv_result(Layer_2_data_length-Layer_2_kernel_size
downto 0, Layer_2_Number_Of_Kernels-1 downto 0):=(others=>(others=>0.0))
-- out_conv_array_6 : out conv_result(Layer_6_data_length-Layer_6_kernel_size
downto 0, Layer_6_Number_Of_Kernels-1 downto 0):=(others=>(others=>0.0))
--result     : out softmax(Number_of_Classes-1 downto 0) :=(others=>0.0)
);
end TOP_CNN;

architecture Behavioral of TOP_CNN is
component Convolution_1D
generic(
kernel_size : integer;
data_length : integer;
one_sample_dim: integer;
Number_Of_Kernels: integer


);
Port (
```

```
init_done  : in std_logic := '0';
data_valid : in std_logic := '0';
clk : in std_logic;
reset : in std_logic :='0';
data_in : in real :=0.0;
kernel_in : in real :=0.0;
conv_out : out real :=0.0;
out_conv_array : out conv_result((data_length-kernel_size)/2-(data_length mod 2)
downto 0, Number_Of_Kernels-1 downto 0):=(others=>(others=>0.0))
);
end component;
component Convolution_1D_Middle is
generic(

kernel_size : integer ;
data_length : integer ;
one_sample_dim: integer ;
Number_Of_Kernels: integer

);
Port (
init_done  : in std_logic := '0';
data_in_buffer : in conv_result(data_length-1 downto 0, one_sample_dim-1 downto
0):=(others=>(others=>0.0));
out_conv_array : out conv_result((data_length-kernel_size)/2-(data_length mod 2)
downto 0, Number_Of_Kernels-1 downto 0):=(others=>(others=>0.0));
clk : in std_logic;
reset : in std_logic :='0';
kernel_in : in real :=0.0;
conv_out : out real :=0.0
);

end component;

component Convolution_1D_No_MP is
generic(
kernel_size : integer ;
data_length : integer ;
one_sample_dim: integer;
Number_Of_Kernels: integer

);
Port (
init_done : in std_logic := '0';
data_in_buffer : in conv_result(data_length-1 downto 0, one_sample_dim-1 downto
0):=(others=>(others=>to_float(0.0)));
out_conv_array : out conv_result(data_length-kernel_size downto 0,
Number_Of_Kernels-1 downto 0):=(others=>(others=>to_float(0.0)));
clk : in std_logic;
reset : in std_logic :='0';
```

```
kernel_in : in real :=0.0;
conv_out : out real :=0.0
);
end component;

component Dense is
generic(
Kernel_Hor_Dim : integer;
Kernel_Ver_Dim : integer;
Number_of_Classes: integer;
data_length : integer ;
one_sample_dim: integer

);
Port (
clk : in std_logic;
reset : in std_logic:= '0';
init_done  : in std_logic := '0';
data_in_buffer : in conv_result(data_length-1 downto 0, one_sample_dim-1 downto
0):=(others=>(others=>to_float(0.0)));
Kernel_in : in float32;
data_valid : in std_logic := '0'

);
end component;


signal    out_conv_array :  conv_result(Layer_1_data_length-Layer_1_kernel_size
downto 0, Layer_1_Number_Of_Kernels-1 downto 0):=(others=>(others=>0.0));
signal    out_conv_array_1 :  conv_result((Layer_1_data_length-
Layer_1_kernel_size)/2  downto 0, Layer_1_Number_Of_Kernels-1 downto
0):=(others=>(others=>0.0));
signal    data_valid : std_logic :='0';
 signal  out_conv_array_2 :  conv_result((Layer_2_data_length-
Layer_2_kernel_size)/2-(Layer_2_data_length mod 2) downto 0,
Layer_2_Number_Of_Kernels-1 downto 0):=(others=>(others=>0.0));
signal out_conv_array_3 :  conv_result((Layer_3_data_length-
Layer_3_kernel_size)/2-(Layer_3_data_length mod 2) downto 0,
Layer_3_Number_Of_Kernels-1 downto 0):=(others=>(others=>0.0));
signal out_conv_array_4 :  conv_result((Layer_4_data_length-
Layer_4_kernel_size)/2-(Layer_4_data_length mod 2) downto 0,
Layer_4_Number_Of_Kernels-1 downto 0):=(others=>(others=>0.0));
signal out_conv_array_5 :  conv_result(Layer_5_data_length-Layer_5_kernel_size
downto 0, Layer_5_Number_Of_Kernels-1 downto 0):=(others=>(others=>0.0));
signal out_conv_array_6 : conv_result(Layer_6_data_length-Layer_6_kernel_size
downto 0, Layer_6_Number_Of_Kernels-1 downto 0):=(others=>(others=>0.0));
signal result : conv_result(Number_of_Classes-1 downto 0);
begin
Layer_1: Convolution_1D
   generic map(
```

```
    kernel_size     =>          Layer_1_kernel_size,
    data_length     =>          Layer_1_data_length,
    one_sample_dim  =>          Layer_1_one_sample_dim,
    Number_Of_Kernels   =>      Layer_1_Number_Of_Kernels)
    port map(
    clk     =>      clk,
    reset   =>      reset ,
    init_done   =>      init_done,
    data_valid  =>      data_valid,
    data_in     =>      data_in ,
    kernel_in   =>      kernel_in,
    conv_out    =>      conv_out,
    out_conv_array => out_conv_array_1
    );

Layer_2: Convolution_1D_Middle
    generic map(
    kernel_size     =>      Layer_2_kernel_size,
    data_length     =>      Layer_2_data_length,
    one_sample_dim  =>      Layer_2_one_sample_dim,
    Number_Of_Kernels   =>  Layer_2_Number_Of_Kernels
    )
    port map(                       -- Bu mimari anlamlı mı?
    clk     =>      clk ,
    reset   =>      reset ,
    init_done   =>      init_done,
    kernel_in   =>      kernel2_in,
    conv_out    =>      conv_out,
    data_in_buffer =>   out_conv_array_1,
    out_conv_array => out_conv_array_2 );
Layer_3: Convolution_1D_Middle
    generic map(
    kernel_size     =>      Layer_3_kernel_size,
    data_length     =>      Layer_3_data_length,
    one_sample_dim  =>      Layer_3_one_sample_dim,
    Number_Of_Kernels   =>  Layer_3_Number_Of_Kernels
    )
    port map(                       -- Bu mimari anlamlı mı?
    clk     =>      clk ,
    reset   =>      reset ,
    init_done   =>      init_done,
    kernel_in   =>      kernel3_in,
    conv_out    =>      conv_out,
    data_in_buffer =>   out_conv_array_2,
    out_conv_array => out_conv_array_3 );

Layer_4: Convolution_1D_Middle
    generic map(
    kernel_size     =>      Layer_4_kernel_size,
    data_length     =>      Layer_4_data_length,
```

```
    one_sample_dim   =>          Layer_4_one_sample_dim,
    Number_Of_Kernels       =>     Layer_4_Number_Of_Kernels
    )
    port map(                           -- Bu mimari anlamlı mı?
    clk         =>         clk ,
    reset       =>         reset ,
    init_done    =>         init_done,
    kernel_in    =>         kernel4_in,
    conv_out     =>         conv_out,
    data_in_buffer =>    out_conv_array_3,
    out_conv_array => out_conv_array_4 );
Layer_5: Convolution_1D_No_MP
    generic map(
    kernel_size      =>          Layer_5_kernel_size,
    data_length      =>          Layer_5_data_length,
    one_sample_dim   =>          Layer_5_one_sample_dim,
    Number_Of_Kernels       =>     Layer_5_Number_Of_Kernels
    )
    port map(                           -- Bu mimari anlamlı mı?
    clk         =>         clk ,
    reset       =>         reset ,
    init_done    =>         init_done,
    kernel_in    =>         kernel5_in,
    conv_out     =>         conv_out,
    data_in_buffer =>    out_conv_array_4,
    out_conv_array => out_conv_array_5 );
Layer_6: Convolution_1D_No_MP
    generic map(
    kernel_size      =>          Layer_6_kernel_size,
    data_length      =>          Layer_6_data_length,
    one_sample_dim   =>          Layer_6_one_sample_dim,
    Number_Of_Kernels       =>     Layer_6_Number_Of_Kernels
    )
    port map(                           -- Bu mimari anlamlı mı?
    clk         =>         clk ,
    reset       =>         reset ,
    init_done    =>         init_done,
    kernel_in    =>         kernel6_in,
    conv_out     =>         conv_out,
    data_in_buffer =>    out_conv_array_5,
    out_conv_array => out_conv_array_6 );
Layer_7: Dense
    generic map(

      Kernel_Hor_Dim => Kernel_Hor_Dim,
      Kernel_Ver_Dim => Kernel_Ver_Dim,
      Number_of_Classes => Number_of_Classes,
      data_length => data_length,
      one_sample_dim => one_sample_dim
          )
```

```
    port map(

    clk => clk,
    reset=> reset,
    init_done => init_done,
    data_in_buffer =>out_conv_array_6,
    Kernel_in => kernel7_in,
    data_valid => data_valid
   -- result => result




--    );
end Behavioral;
```

APPENDIX B:

The Code written on vitis platform is shown.

```
#include <stdio.h>
#include "platform.h"
#include "xil_printf.h"
#include "xuartlite.h"      // UART Lite sürücüsü
#include "xaxicdma.h"
#include "mb_interface.h"
#include "xaxidma.h"

#define DDR_BASE_ADDRESS  0x80000000  // Replace with the actual base
address of your DDR memory
#define BUFFER_SIZE       32        // Number of 32-bit words to test
#define UARTLITE_DEVICE_ID XPAR_UARTLITE_0_DEVICE_ID
#define DATA_SIZE         2585792      // Alınacak veri boyutu (900,000
byte)
//#define DATA_SIZE         45120
#define DMA_DEV_ID     XPAR_AXICDMA_0_DEVICE_ID

// KERNEL_ADRESSES
#define SAMPLE_ADDRESS          DDR_BASE_ADDRESS
#define FILTER_1_ADDRESS  0x80009C40
#define FILTER_2_ADDRESS  0x8000B040
#define FILTER_3_ADDRESS  0x80011040
#define FILTER_4_ADDRESS  0x80039040
#define FILTER_5_ADDRESS  0x80099040
#define FILTER_6_ADDRESS  0x80259040
#define LINEAR_C_ADDRESS  0x80271040
#define RESULT_ADDRESS      0x81000000
#define RESULT_ADDRESS_2    0x82000000
#define RESULT_ADDRESS_3    0x83000000
#define RESULT_ADDRESS_4    0x84000000
#define RESULT_ADDRESS_5    0x85000000
#define RESULT_ADDRESS_6    0x86000000

#define ISR_OFFSET  0x0    // Interrupt Status Register
#define IER_OFFSET  0x4    // Interrupt Enable Register
#define TDFR_OFFSET 0x8    // Transmit Data FIFO Reset
#define TDFV_OFFSET 0xC    // Transmit Data FIFO Vacancy
#define TDFD_OFFSET 0x10   // Transmit Data FIFO Write Port
#define TLR_OFFSET  0x14   // Transmit Length Register
#define RDFR_OFFSET 0x18   // Receive Data FIFO Reset
#define RDFO_OFFSET 0x1C   // Receive Data FIFO Occupancy
#define RDFD_OFFSET 0x20   // Receive Data FIFO Read Port
#define RLR_OFFSET  0x24   // Receive Length Register




XUartLite UartLite;  // UART Lite örneği
union {
    uint32_t i;
    float f;
} u;
```

```
enum state{data_init_1, kernel_init_1, operation_1,idle,data_init_2,
kernel_init_2, operation_2,data_init_3, kernel_init_3, operation_3
      ,data_init_4, kernel_init_4, operation_4,data_init_5, kernel_init_5,
operation_5, data_init_6, kernel_init_6, operation_6,data_init_7,
kernel_init_7, operation_7};
u8 kernel_init_done=0;
u8 data_init_done=0;
XAxiCdma AxiCdma;
XAxiDma AxiDma_0, AxiDma_1,AxiDma_2;
//
XAxiCdma_Config *CfgPtr;

uint32_t value=0;
u32 index_offset=0;
u32 index_offset_2=0;
u32 counter=0;
uint16_t kernel_counter=0;
uint16_t data_counter=0;
uint32_t result_counter=0;

float Kernel_Buffer[40]={0};
float Data_Buffer[40]={0};

float Kernel_Buffer_2[96]={0};
float Data_Buffer_2[96]={0};

float Kernel_Buffer_3[320]={0};
float Data_Buffer_3[320]={0};

float Kernel_Buffer_4[384]={0};
float Data_Buffer_4[384]={0};

float Kernel_Buffer_5[1792]={0};
float Data_Buffer_5[1792]={0};

float Kernel_Buffer_6[768]={0};
float Data_Buffer_6[768]={0};

float Kernel_Buffer_7[2144]={0};
float Data_Buffer_7[2144]={0};


float common_result[2144]={0};
float scores[3]={0};


float result_1=0;
float result_2=0;

uint32_t swap_endianness(uint32_t value) {
    return ((value >> 24) & 0x000000FF) |  // İlk byte'ı en sona
           ((value >> 8) & 0x0000FF00)  |  // İkinci byte'ı ortalara
           ((value << 8) & 0x00FF0000)  |  // Üçüncü byte'ı ortalara
           ((value << 24) & 0xFF000000);   // Son byte'ı başa
}

float Find_Max(float value1,float value2)
{
      //float x=0;
```

```
	if(value1>value2)
		return value1;
	else
		return value2;
	//return x;
}

enum state state=kernel_init_1;

int main()
{
	float result=0;
	init_platform();
    u32 data_to_send = 0x11111111;  // Gönderilecek veri
    u32 received_data;
    u8 status;
    float input_0[8]={-23,23,23,23,23,23,23,23};
    float input_1[8]={1,2,3,4,5,6,7,8};
    float input_2[8]={0};
    float input_3[8]={9,10,11,12,13,14,15,16};
    XAxiDma_Config *CfgPtr_0 =
XAxiDma_LookupConfig(XPAR_AXIDMA_0_DEVICE_ID);
    XAxiDma_Config *CfgPtr_1 =
XAxiDma_LookupConfig(XPAR_AXIDMA_1_DEVICE_ID);
    XAxiDma_Config *CfgPtr_2 =
XAxiDma_LookupConfig(XPAR_AXIDMA_2_DEVICE_ID);
    if (!CfgPtr_0 || !CfgPtr_1) {
        xil_printf("No DMA configuration found.\r\n");
        return;
    }

    // DMA 0 ve DMA 1'in başlatılması
    status = XAxiDma_CfgInitialize(&AxiDma_0, CfgPtr_0);
    status = XAxiDma_CfgInitialize(&AxiDma_1, CfgPtr_1);
    status = XAxiDma_CfgInitialize(&AxiDma_2, CfgPtr_2);

    // DMA sıfırlama
    XAxiDma_Reset(&AxiDma_0);
    XAxiDma_Reset(&AxiDma_1);
    XAxiDma_Reset(&AxiDma_2);

//    status = XAxiDma_SimpleTransfer(&AxiDma_0, (UINTPTR)input_0, 32,
XAXIDMA_DMA_TO_DEVICE);
//    status = XAxiDma_SimpleTransfer(&AxiDma_1, (UINTPTR)input_1, 32,
XAXIDMA_DMA_TO_DEVICE);
//    status = XAxiDma_SimpleTransfer(&AxiDma_2, (UINTPTR)input_2, 32,
XAXIDMA_DEVICE_TO_DMA);




	volatile uint8_t *ddr_ptr = (volatile uint8_t *)DDR_BASE_ADDRESS;
	int received_bytes;
	int total_received = 0;
	u8 uart_buffer[BUFFER_SIZE];

	XUartLite_Initialize(&UartLite, UARTLITE_DEVICE_ID);
```

```
          CfgPtr = XAxiCdma_LookupConfig(DMA_DEV_ID);
          status=XAxiCdma_CfgInitialize(&AxiCdma, CfgPtr,CfgPtr-
>BaseAddress);



          while (XAxiCdma_IsBusy(&AxiCdma)) {
                  // DMA meşgul durumda...
              }

       while (total_received < DATA_SIZE) {
              // UARTLite'tan veri al, buffer'a yaz
              received_bytes = XUartLite_Recv(&UartLite, uart_buffer,
sizeof(uart_buffer));

              // Alınan veriyi DDR belleğe yaz
              for (int i = 0; i < received_bytes; i++) {
                  ddr_ptr[total_received] = uart_buffer[i];
                  total_received++;
              }
          }

       while(1)
       {


      switch(state){
            case kernel_init_1:
                  if(kernel_counter==32)
                        {state=kernel_init_2;
                        kernel_counter=0;}
                  else
                              {
                                    XAxiCdma_Reset(&AxiCdma);

index_offset=FILTER_1_ADDRESS+160*kernel_counter;

      status=XAxiCdma_SimpleTransfer(&AxiCdma,index_offset,(UINTPTR)Kernel
_Buffer , 160,NULL,NULL);
                              while(XAxiCdma_IsBusy(&AxiCdma))
                              {}
                              kernel_counter=kernel_counter+1;
                              state=data_init_1;
                              }
                  break;

            case data_init_1:
                  if(data_counter==1246)
                        {kernel_init_done=0;
                        state=kernel_init_1;
                        data_counter=0;}
                  else

                          { XAxiCdma_Reset(&AxiCdma);

      index_offset=DDR_BASE_ADDRESS+32*data_counter;
```

```
                                    status=XAxiCdma_SimpleTransfer(&AxiCdma,
index_offset,(UINTPTR)Data_Buffer , 160,NULL,NULL);
                                data_counter=data_counter+1;
                                    state=operation_1;

                                }
                        break;
                case operation_1:

//                      for(int i=0; i<40; ++i)
//                      {
//                              result=result+Data_Buffer[i]*Kernel_Buffer[i];
//Buraya bir custom IP bizi bekler...
//
//
//                      }
                        XAxiDma_Reset(&AxiDma_0);
                        XAxiDma_Reset(&AxiDma_1);
                        XAxiDma_Reset(&AxiDma_2);

                        status = XAxiDma_SimpleTransfer(&AxiDma_0,
(UINTPTR)Data_Buffer, 160, XAXIDMA_DMA_TO_DEVICE);
                        status = XAxiDma_SimpleTransfer(&AxiDma_1,
(UINTPTR)Kernel_Buffer, 160, XAXIDMA_DMA_TO_DEVICE);
//                      while(XAxiDma_Busy(&AxiDma_1, XAXIDMA_DMA_TO_DEVICE))
//                      {}
                        status = XAxiDma_SimpleTransfer(&AxiDma_2,
(UINTPTR)common_result, 160, XAXIDMA_DEVICE_TO_DMA);
                        while(XAxiDma_Busy(&AxiDma_2, XAXIDMA_DEVICE_TO_DMA))
                        {}

                result=common_result[39];
                                ++result_counter;

//                              if(result<=0)
//                                      result=0;
                                if(result_counter==1)
                                {
                                        result_1=result;
                                        state=data_init_1;
                                }
                                else if(result_counter==2)
                                {
                                        result_2=result;
                                        result_counter=0;
                                        result=Find_Max(result_1,result_2);
                                        XAxiCdma_Reset(&AxiCdma);
//                                      u.f=result;
//                                      u.i=swap_endianness(u.i);
//                                      result=u.f;
//                                      while(!(XAxiCdma_ResetIsDone(&AxiCdma)))
//                                      {}
                                        XAxiCdma_SimpleTransfer(&AxiCdma, &result,
RESULT_ADDRESS+((kernel_counter-1)+16*(data_counter-2))*4 , 4,NULL,NULL);
                                while(XAxiCdma_IsBusy(&AxiCdma))
                                {}

//
```

```
//                                Xil_Out32((RESULT_ADDRESS+(kernel_counter-
1)+32*(data_counter-2)), u.i);
                                result=0;
                                state=data_init_1;
                                ++counter;
                          }


                  break;

            case kernel_init_2:
                  if(kernel_counter==64)
                        {state=kernel_init_3;
                        kernel_counter=0;}
                  else
                                {           XAxiCdma_Reset(&AxiCdma);

      index_offset=FILTER_2_ADDRESS+kernel_counter*384;

      status=XAxiCdma_SimpleTransfer(&AxiCdma, index_offset,(float
*)Kernel_Buffer_2 , 384,NULL,NULL);
                                      while(XAxiCdma_IsBusy(&AxiCdma))
                                      {}

      kernel_counter=kernel_counter+1;
                                            state=data_init_2;
                                }
                  break;
            case data_init_2:
                  if(data_counter==620)
                        {kernel_init_done=0;
                        state=kernel_init_2;
                        data_counter=0;}
                  else

                          {
                                XAxiCdma_Reset(&AxiCdma);

index_offset=RESULT_ADDRESS+128*data_counter;

status=XAxiCdma_SimpleTransfer(&AxiCdma,index_offset
,(UINTPTR)Data_Buffer_2 , 384,NULL,NULL);
                                while(XAxiCdma_IsBusy(&AxiCdma))
                                {}
                                  data_counter=data_counter+1;
                                  state=operation_2;

                           }
                  break;
            case operation_2:
//                                for(int i=0; i<96; ++i)
//                                {
//
      result=Data_Buffer_2[i]*Kernel_Buffer_2[i];
//                                }
                           XAxiDma_Reset(&AxiDma_0);
                           XAxiDma_Reset(&AxiDma_1);
                           XAxiDma_Reset(&AxiDma_2);
```

```
                              status = XAxiDma_SimpleTransfer(&AxiDma_0,
(UINTPTR)Data_Buffer_2, 384, XAXIDMA_DMA_TO_DEVICE);
                              status = XAxiDma_SimpleTransfer(&AxiDma_1,
(UINTPTR)Kernel_Buffer_2, 384, XAXIDMA_DMA_TO_DEVICE);
//                            while(XAxiDma_Busy(&AxiDma_1,
XAXIDMA_DMA_TO_DEVICE))
//                            {}
                              status = XAxiDma_SimpleTransfer(&AxiDma_2,
(UINTPTR)common_result, 384, XAXIDMA_DEVICE_TO_DMA);
                              while(XAxiDma_Busy(&AxiDma_2,
XAXIDMA_DEVICE_TO_DMA))
                              {}
                              result=common_result[95];
                                 ++result_counter;
//                               if(result<=0)
//                                     result=0;
                                 if(result_counter==1)
                                 {
                                       result_1=result;
                                       state=data_init_2;
                                 }
                                 else if(result_counter==2)
                                 {   result_2=result;
                                       result_counter=0;
                                       result=Find_Max(result_1,result_2);
                                       XAxiCdma_Reset(&AxiCdma);
                                       XAxiCdma_SimpleTransfer(&AxiCdma,
&result ,RESULT_ADDRESS_2+4*((kernel_counter-1)+32*(data_counter-2)),
4,NULL,NULL);
                                 while(XAxiCdma_IsBusy(&AxiCdma))
                                 {}
                                       result=0;
                                       state=data_init_2;
                                       ++counter;
                                 }

                  break;
            case kernel_init_3:
                  if(kernel_counter==128)
                        {state=kernel_init_4;
                        kernel_counter=0;}
                  else
                              {
                                       XAxiCdma_Reset(&AxiCdma);

      index_offset=FILTER_3_ADDRESS+kernel_counter*1280;

      status=XAxiCdma_SimpleTransfer(&AxiCdma, index_offset,(float
*)Kernel_Buffer_3 , 1280,NULL,NULL);
                              while(XAxiCdma_IsBusy(&AxiCdma))
                              {}
                                       kernel_counter=kernel_counter+1;
                                       state=data_init_3;
                              }
                  break;
            case data_init_3:
                  if(data_counter==306)
                        {kernel_init_done=0;
                        state=kernel_init_3;
```

```
                        data_counter=0;}
                  else

                        {
                              XAxiCdma_Reset(&AxiCdma);
                              status=XAxiCdma_SimpleTransfer(&AxiCdma,
RESULT_ADDRESS_2+256*data_counter,(float *)Data_Buffer_3 ,1280,NULL,NULL);
                              data_counter=data_counter+1;
                              state=operation_3;

                         }
                  break;
            case operation_3:
//                for(int i=0; i<320; ++i)
//                {
//                      result=Data_Buffer_3[i]*Kernel_Buffer_3[i];
//                }
//                for(int i=0; i<4; i++)
//                {
                  XAxiDma_Reset(&AxiDma_0);
                  XAxiDma_Reset(&AxiDma_1);
                  XAxiDma_Reset(&AxiDma_2);

                  status = XAxiDma_SimpleTransfer(&AxiDma_0,
(UINTPTR)Data_Buffer_3, 1280, XAXIDMA_DMA_TO_DEVICE);
                  status = XAxiDma_SimpleTransfer(&AxiDma_1,
(UINTPTR)Kernel_Buffer_3, 1280, XAXIDMA_DMA_TO_DEVICE);
//                while(XAxiDma_Busy(&AxiDma_1, XAXIDMA_DMA_TO_DEVICE))
//                {}
                  status = XAxiDma_SimpleTransfer(&AxiDma_2,
(UINTPTR)common_result, 1280, XAXIDMA_DEVICE_TO_DMA);
                  while(XAxiDma_Busy(&AxiDma_2, XAXIDMA_DEVICE_TO_DMA))
                  {}
//                }


                  result=common_result[319];
                        ++result_counter;
//                      if(result<=0)
//                            result=0;
                        if(result_counter==1)
                        {
                              result_1=result;
                              state=data_init_3;
                        }
                        else if(result_counter==2)
                        {
                              result_2=result;
                              result_counter=0;
                              result=Find_Max(result_1,result_2);
                              XAxiCdma_Reset(&AxiCdma);
                              XAxiCdma_SimpleTransfer(&AxiCdma, &result
,RESULT_ADDRESS_3+4*((kernel_counter-1)+64*(data_counter-2)),
4,NULL,NULL);
                        while(XAxiCdma_IsBusy(&AxiCdma))
                        {}
                              result=0;
                              state=data_init_3;
                              ++counter;
```

```
                        }

                        break;
                case kernel_init_4:
                        if(kernel_counter==256)
                                {state=kernel_init_5;
                                kernel_counter=0;}
                        else
                                        {
                                                XAxiCdma_Reset(&AxiCdma);

        index_offset=FILTER_4_ADDRESS+kernel_counter*1536;

        status=XAxiCdma_SimpleTransfer(&AxiCdma, index_offset,(float
*)Kernel_Buffer_4 , 1536,NULL,NULL);
                                        while(XAxiCdma_IsBusy(&AxiCdma))
                                        {}
                                                kernel_counter=kernel_counter+1;
                                                state=data_init_4;
                                        }
                        break;
                case data_init_4:
                        if(data_counter==150)
                                {kernel_init_done=0;
                                state=kernel_init_4;
                                data_counter=0;}
                        else

                                  {
                                        XAxiCdma_Reset(&AxiCdma);
                                        status=XAxiCdma_SimpleTransfer(&AxiCdma,
RESULT_ADDRESS_3+512*data_counter,(float *)Data_Buffer_4 ,1536,NULL,NULL);
                                        data_counter=data_counter+1;
                                        state=operation_4;

                                   }
                        break;
                case operation_4:
//                      for(int i=0; i<384; ++i)
//                      {
//                              result=Data_Buffer_4[i]*Kernel_Buffer_4[i];
//                      }
//                      for(int i=0; i<4; i++)
//                      {
                    XAxiDma_Reset(&AxiDma_0);
                    XAxiDma_Reset(&AxiDma_1);
                    XAxiDma_Reset(&AxiDma_2);

                    status = XAxiDma_SimpleTransfer(&AxiDma_0,
(UINTPTR)Data_Buffer_4, 1536, XAXIDMA_DMA_TO_DEVICE);
                    status = XAxiDma_SimpleTransfer(&AxiDma_1,
(UINTPTR)Kernel_Buffer_4, 1536, XAXIDMA_DMA_TO_DEVICE);
//                  while(XAxiDma_Busy(&AxiDma_1, XAXIDMA_DMA_TO_DEVICE))
//                  {}
                    status = XAxiDma_SimpleTransfer(&AxiDma_2,
(UINTPTR)common_result, 1536, XAXIDMA_DEVICE_TO_DMA);
                    while(XAxiDma_Busy(&AxiDma_2, XAXIDMA_DEVICE_TO_DMA))
                    {}
//                    }
```

```
                result=common_result[383];
                                ++result_counter;
//                              if(result<=0)
//                                    result=0;
                                if(result_counter==1)
                                      {
                                      result_1=result;
                                      state=data_init_4;
                                      }
                                else if(result_counter==2)
                                {
                                      result_2=result;
                                      result_counter=0;
                                      result=Find_Max(result_1,result_2);
                                      XAxiCdma_Reset(&AxiCdma);

      XAxiCdma_SimpleTransfer(&AxiCdma,&result ,
RESULT_ADDRESS_4+4*((kernel_counter-1)+128*(data_counter-2)),
4,NULL,NULL);
                                while(XAxiCdma_IsBusy(&AxiCdma))
                                {}
                                      result=0;
                                      state=data_init_4;
                                      ++counter;
                                }

                  break;
            case kernel_init_5:
                  if(kernel_counter==256)
                        {state=kernel_init_6;
                        kernel_counter=0;}
                  else
                                {
                                      XAxiCdma_Reset(&AxiCdma);

      index_offset=FILTER_5_ADDRESS+kernel_counter*7168;

      status=XAxiCdma_SimpleTransfer(&AxiCdma, index_offset,(float
*)Kernel_Buffer_5 , 7168,NULL,NULL);
                                while(XAxiCdma_IsBusy(&AxiCdma))
                                {}
                                      kernel_counter=kernel_counter+1;
                                      state=data_init_5;
                                }
                  break;
            case data_init_5:
                  if(data_counter==69)
                        {kernel_init_done=0;
                        state=kernel_init_5;
                        data_counter=0;}
                  else

                        {
                              XAxiCdma_Reset(&AxiCdma);
                              status=XAxiCdma_SimpleTransfer(&AxiCdma,
RESULT_ADDRESS_4+1024*data_counter,(float *)Data_Buffer_5
,7168,NULL,NULL);
                              data_counter=data_counter+1;
                              state=operation_5;
```

```
                        }
                        break;
                case operation_5:
//                      for(int i=0; i<1792; ++i)
//                      {
//                              result=Data_Buffer_5[i]*Kernel_Buffer_5[i];
//                      }
//                      for(int i=0; i<16; ++i)
//                      {
                        XAxiDma_Reset(&AxiDma_0);
                        XAxiDma_Reset(&AxiDma_1);
                        XAxiDma_Reset(&AxiDma_2);

                        status = XAxiDma_SimpleTransfer(&AxiDma_0,
(UINTPTR)Data_Buffer_5, 7168, XAXIDMA_DMA_TO_DEVICE);
                        status = XAxiDma_SimpleTransfer(&AxiDma_1,
(UINTPTR)Kernel_Buffer_5, 7168, XAXIDMA_DMA_TO_DEVICE);
//                      while(XAxiDma_Busy(&AxiDma_1, XAXIDMA_DMA_TO_DEVICE))
//                      {}
                        status = XAxiDma_SimpleTransfer(&AxiDma_2,
(UINTPTR)common_result, 7168, XAXIDMA_DEVICE_TO_DMA);
                        while(XAxiDma_Busy(&AxiDma_2, XAXIDMA_DEVICE_TO_DMA))
                        {}
//                      }

                        result=common_result[1791];


//                                      if(result<=0)
//                                      result=0;

                                        XAxiCdma_Reset(&AxiCdma);
                                        XAxiCdma_SimpleTransfer(&AxiCdma,&result ,
RESULT_ADDRESS_5+4*((kernel_counter-1)+256*(data_counter-1)),
4,NULL,NULL);
                                while(XAxiCdma_IsBusy(&AxiCdma))
                                {}
                                        result=0;
                                        state=data_init_5;
                                        ++counter;
                                break;

                case kernel_init_6:
                        if(kernel_counter==32)
                                {state=kernel_init_7;
                                kernel_counter=0;}
                        else
                                        {
                                                XAxiCdma_Reset(&AxiCdma);

        index_offset=FILTER_6_ADDRESS+kernel_counter*3072;

        status=XAxiCdma_SimpleTransfer(&AxiCdma, index_offset,(float
*)Kernel_Buffer_6 , 3072,NULL,NULL);
                                        while(XAxiCdma_IsBusy(&AxiCdma))
                                        {}
                                                kernel_counter=kernel_counter+1;
                                                state=data_init_6;
```

```
                                    }
                        break;
                case data_init_6:
                        if(data_counter==67)
                                {kernel_init_done=0;
                                state=kernel_init_6;
                                data_counter=0;}
                        else

                                  {
                                        XAxiCdma_Reset(&AxiCdma);
                                        status=XAxiCdma_SimpleTransfer(&AxiCdma,
RESULT_ADDRESS_5+1024*data_counter,(float *)Data_Buffer_6
,3072,NULL,NULL);
                                        data_counter=data_counter+1;
                                        state=operation_6;

                                   }
                        break;
                case operation_6:
//                      for(int i=0; i<768; ++i)
//                      {
//                              result=Data_Buffer_6[i]*Kernel_Buffer_6[i];
//                      }
//                      for(int i=0; i<8; ++i)
//                      {
                    XAxiDma_Reset(&AxiDma_0);
                    XAxiDma_Reset(&AxiDma_1);
                    XAxiDma_Reset(&AxiDma_2);

                    status = XAxiDma_SimpleTransfer(&AxiDma_0,
(UINTPTR)Data_Buffer_6, 3072, XAXIDMA_DMA_TO_DEVICE);
                    status = XAxiDma_SimpleTransfer(&AxiDma_1,
(UINTPTR)Kernel_Buffer_6, 3072, XAXIDMA_DMA_TO_DEVICE);
//                  while(XAxiDma_Busy(&AxiDma_1, XAXIDMA_DMA_TO_DEVICE))
//                  {}
                    status = XAxiDma_SimpleTransfer(&AxiDma_2,
(UINTPTR)common_result, 3072, XAXIDMA_DEVICE_TO_DMA);
                    while(XAxiDma_Busy(&AxiDma_2, XAXIDMA_DEVICE_TO_DMA))
                    {}
//                      }
                    result=common_result[767];
//                    if(result<=0)
//                    result=0;

                      XAxiCdma_SimpleTransfer(&AxiCdma, &result
,RESULT_ADDRESS_6+4*((kernel_counter-1)+32*(data_counter-1)),
4,NULL,NULL);
                while(XAxiCdma_IsBusy(&AxiCdma))
                {}
                        state=data_init_6;
                        ++counter;

                        break;
                case kernel_init_7:
                        if(kernel_counter==3)
                                {
                                state=idle;
                                kernel_counter=0;
```

```
                        }
                        else
                        {
                                XAxiCdma_Reset(&AxiCdma);

      index_offset=LINEAR_C_ADDRESS+kernel_counter*8576;
                                status=XAxiCdma_SimpleTransfer(&AxiCdma,
index_offset,(float *)Kernel_Buffer_7 , 8576,NULL,NULL);
                        while(XAxiCdma_IsBusy(&AxiCdma))
                        {}
                                kernel_counter=kernel_counter+1;
                                if(kernel_counter>1)
                                        state=operation_7;
                                else
                                        state=data_init_7;

                        }
                    break;

            case data_init_7:

                                  XAxiCdma_Reset(&AxiCdma);
                                  status=XAxiCdma_SimpleTransfer(&AxiCdma,
RESULT_ADDRESS_6,(float *)Data_Buffer_7 ,8576,NULL,NULL);
                                  state=operation_7;
                    break;
            case operation_7:
//                  for(int i=0;i<32;i++)
//                  {
                    putfsl(data_to_send,0);
                    XAxiDma_Reset(&AxiDma_0);
                XAxiDma_Reset(&AxiDma_1);
                XAxiDma_Reset(&AxiDma_2);

                status = XAxiDma_SimpleTransfer(&AxiDma_0,
(UINTPTR)Data_Buffer_7, 8576, XAXIDMA_DMA_TO_DEVICE);
                status = XAxiDma_SimpleTransfer(&AxiDma_1,
(UINTPTR)Kernel_Buffer_7, 8576, XAXIDMA_DMA_TO_DEVICE);
//              while(XAxiDma_Busy(&AxiDma_1, XAXIDMA_DMA_TO_DEVICE))
//              {}
                status = XAxiDma_SimpleTransfer(&AxiDma_2,
(UINTPTR)common_result, 8576, XAXIDMA_DEVICE_TO_DMA);
                while(XAxiDma_Busy(&AxiDma_2, XAXIDMA_DEVICE_TO_DMA))
                {}
//                  }
                    scores[result_counter]=common_result[2143];
                ++result_counter;


//                  XAxiCdma_SimpleTransfer(&AxiCdma, &result ,scores,
4,NULL,NULL);
//          while(XAxiCdma_IsBusy(&AxiCdma))
//          {}
                    state=kernel_init_7;
                    ++counter;
                    break;
            case idle:
                    print("operation_done");
                    break;
```

```
        }}
    cleanup_platform();
    return 0;
}
// Initialize the DMA engine
```

CURRICULUM VITAE

Name Surname: Mert Yaşar AYDIN

Education

MSc, Electronics Engineering
09.2021-01.2025
ISTANBUL TECHNICAL UNIVERSITY
GPA : (3.28/4.00)

BSc, Electrical and Electronic Engineering
09.2014-06.2020
MIDDLE EAST TECHNICAL UNIVERSITY
GPA : (3.00/4.00)

Professional Experience

TÜBİTAK RUTE
Kocaeli, TURKEY
Researcher
01.2023-continue

TÜBİTAK MAM Institute of Materials
Kocaeli, TURKEY
Researcher
03.2021-01.2023