CHAPTER 1

INTRODUCTION

1.1 OVERVIEW

In the current age of digitization, handwriting recognition plays an important role in information processing. There is a lot of information available on paper and less processing of digital files than the processing of traditional paper files. The purpose of the handwriting recognition system is to convert handwritten letters into machine-readable formats. Major applications include vehicle license-plate identification, postal paper-sorting services, historical document preservation in the check truncation system (CTS) scanning and archaeology departments, old document automation in libraries and banks, and more. All of these areas deal with large databases and therefore requirehigh identification accuracy, low computational complexity, and consistent performance of the identification system. Over time, the number of fields that can implement deep learning is increasing. In deep learning, convolutional neural networks (CNN) are being used for visual image analysis. CNN can be used inobject detection, facial recognition, robotics, video analysis, segmentation, pattern recognition, natural language processing, spam detection, topical gradation, regression analysis, speech recognition, image classification.

Convolutional Neural Network (CNN): A convolutional neural network (CNN) is a specialized type of artificial neural network commonly used for image and video processing tasks. CNNs are designed to automatically and efficiently extract relevant features from input data through a series of convolutional layers. The key component of a CNN is the convolutional layer,Through successive layers of convolutional and pooling operations, CNNs can capture increasingly complex and abstract features from the input data. Finally, the extracted features are flattened and fed into fully connected layers, followed by an output layer for classification or regression tasks. Overall, CNNs excel in tasks that require understanding and analysis of visual data due to their ability to automatically learn hierarchical representations of features, enabling them to achieve impressive performance in image classification, object detection, and image segmentation, among other computer vision tasks.

Detection of handwritten numbers, including accuracy in these areas, has reached human perfectionusing deep convolutional neural networks (CNNs). Recently CNN has become one of the most attractive approaches and has been the ultimate factor in recent success and in several challengingmachine learning applications. Considering all the factors stated above we have chosen CNN for our challenging tasks of image classification. We can use it to identify handwritten numbers, which is one of the higher education and business transactions. There are many applications of handwritten digit recognition for our real-life purposes. Hence we are using the Convolutional NeuralNetwork (CNN) and MNIST dataset.

1.2 PROBLEM STATEMENT

Handwriting recognition has been the main subject of research for almost the last forty years. This research work analyzes the behaviour of classification techniques (CNN) in a large handwriting dataset (MNIST) to predict a digit. Machine-learning techniques, particularly when applied to NeuralNetworks like CNN or ANN, have played an increasingly important role in the design of these recognition systems. Several methods have been developed in handwritten digit recognition and these methods have been classified into categories: knowledge-based methods, feature-based methods, template-based methods and appearance-based methods. Errors in Digit recognition cause severe problems like digits written on a bank cheque if recognized erroneously could result in unfortunate consequences.

1.3 MOTIVATION

Handwritten digit recognition is an important area of research and application in the field of machine learning and computer vision. There are several motivations for developing accurate algorithms for handwritten digit recognition:

a. Digit recognition is a fundamental problem in pattern recognition and machine learning, and accurate recognition of handwritten digits is an important step towards solving more complex recognition problems.
b. Digit recognition has many practical applications, such as recognizing postal codes on letters, recognizing bank account numbers on checks, and recognizing license plate numbers on vehicles.

c. Digit recognition is also important in the field of document analysis and recognition, where it is used to recognize handwritten text in scanned documents.

d. Handwritten digit recognition is a challenging problem because of the variability in handwriting styles and the different ways people write the same digit. Solving this problem requires the development of sophisticated algorithms that can handle this variability.

e. The development of accurate algorithms for handwritten digit recognition has implications for other areas of research, such as character recognition, handwriting analysis, and natural language processing.

1.3 OBJECTIVE

I. Handwriting recognition has been the main subject of research for almost the last forty years. This research work analyzes the behaviour of classification techniques (CNN) in a large handwriting dataset (MNIST) to predict a digit. Machine-learning techniques, particularly when applied to NeuralNetworks like CNN or ANN, have played an increasingly important role in the design of these recognition systems.

II. Several methods have been developed in handwritten digit recognition and these methods have been classified into categories: knowledge-based methods, feature-based methods, template-based methods and appearance-based methods. Errors in Digit recognition cause severe problems like digits written on a bank cheque if recognized erroneously could result in unfortunate consequences.

III. The goal of our work is to create a model that will be able to recognize and classify the handwrittendigits from images by using concepts of Convolution Neural Network. Though the goal of our research is to create a model for digit recognition and -classification, it can also be extended to lettersand an individual’s handwriting.

IV. The major goal of the proposed system is understanding Convolutional Neural Network, and applying it to the handwritten digit recognition system by workingon the MNIST dataset.

CHAPTER 2

LITERATURE SURVEY

Handwritten digit recognition has gained the interest of many researchers in recent years for its wideuse in the field of text recognition systems in all languages and scripts. Reference [1] proposes a hardware accelerator for the inference of a CNN, which includes a convolutional layer, a pooling layer, and a fully connected layer. The design is implemented on an FPGA platform, and the results show significant speedup compared to software-based implementations. In reference [2], FPGA acceleration is explored for a MLP neural network designed for digit recognition. The proposed system is implemented on a Xilinx Virtex-5 FPGA, and the results show faster inference times and lower power consumption compared to a software-based implementation. Reference [3] presents a digit-recognition CNN implemented on an FPGA, where the system is optimized for performance and area utilization. The proposed system uses a hardware-software co-design approach and achieves high recognition accuracy and low power consumption. In reference [4], an FPGA-based handwritten digit recognition system is proposed, which includes a pre-processing module, a feature extraction module, and a classification module. The system is designed to operate in real-time and achieves high recognition accuracy. Reference [5] proposes ZyNet, a tool for automating the implementation of deep neural networks on low-cost, reconfigurable edge computing platforms. The proposed system uses a high-level synthesis flow and can generate optimized hardware designs for various edge devices. Reference [6] explores the role of activation functions in CNNs and their impact on accuracy and computation complexity. The study analyzes various activation functions and their effectiveness in different CNN architectures. Reference [7] evaluates fast algorithms for CNNs on FPGAs and proposes a design using the fast Fourier transform algorithm. The proposed system achieves high performance and low resource utilization. Reference [8] presents an FPGA-based handwritten number recognition system using a CNN. The proposed system achieves high accuracy and low power consumption and is implemented on a Xilinx Zynq platform. Reference [9] provides a survey of FPGA-based accelerators for CNNs, where different design approaches, architectures, and optimization techniques are analyzed. The survey provides insights into the current state-of-the-art and challenges in FPGA-based acceleration of CNNs. Reference [10] proposes a CNN based on TensorFlow and implements the design on an FPGA platform. The study explores the performance and accuracy trade-offs of various CNN architectures and optimization techniques. Reference [11] provides access to the source code of ZyNet, which is a tool for automating the implementation of deep neural networks on low-cost, reconfigurable edge computing platforms.

The references mentioned in the statement provide insights into the current research trends and challenges in implementing Convolutional Neural Networks (CNNs) and Multi-Layer Perceptron (MLP) neural networks on Field-Programmable Gate Arrays (FPGAs) for various applications. The studies aim to optimize hardware designs for performance, area utilization, and power consumption, which are crucial factors in designing efficient FPGA-based systems.

The research trend in FPGA-based CNN and MLP design has been growing in recent years due to the increasing demand for low-power and high-performance computing systems. FPGAs provide a flexible platform for hardware acceleration, which is essential for running complex neural network models. Moreover, FPGAs can be customized to suit the specific requirements of different applications. The studies focus on various design approaches and optimization techniques to achieve the best possible performance, area utilization, and power consumption. These approaches include data quantization, weight pruning, pipelining, and parallelization. Data quantization reduces the bit width of weights and activations to decrease the number of arithmetic operations, which leads to a reduction in power consumption and area utilization. Weight pruning reduces the number of weights and connections in the network, which reduces the complexity and memory requirements of the network. Pipelining divides the network into stages, which enables parallel processing and reduces the latency of the network. Parallelization involves partitioning the network into multiple smaller networks that can be run simultaneously, which increases the throughput of the network.

In summary, the references provide valuable insights into the current research trends and challenges in FPGA-based CNN and MLP design. The studies highlight the importance of optimizing hardware designs for performance, area utilization, and power consumption, and explore various design approaches and optimization techniques to achieve these goals.

CHAPTER 3

METHODOLOGY

Handwritten digit recognition using an FPGA (Field-Programmable Gate Array) can be implemented using the following methodology:

1. Datasets:

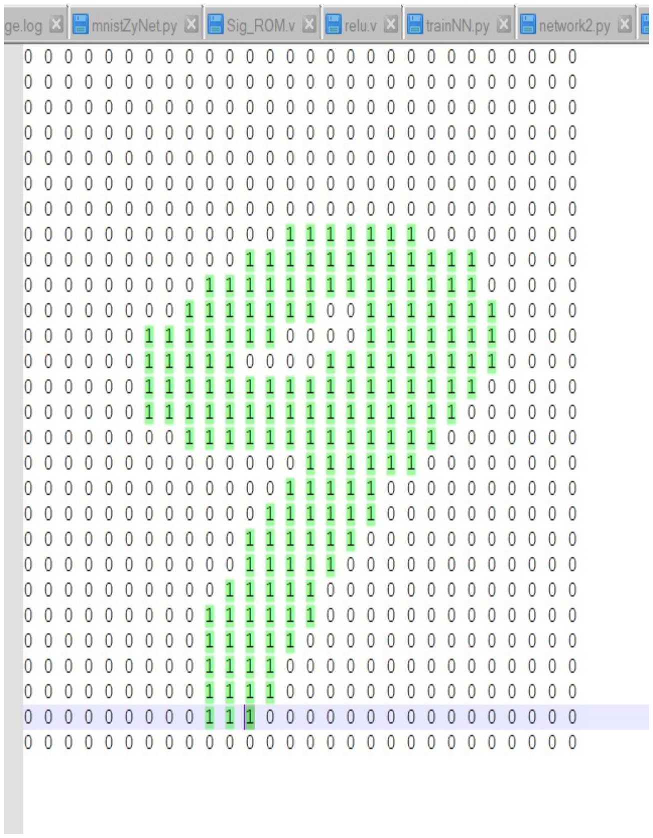
The MNIST database is a well-known dataset in the field of computer vision, which contains handwritten digits that have been scanned and stored as grayscale images with a 28x28 pixel resolution. To use this data in a convolutional neural network (CNN) implemented on an FPGA, the data needs to be preprocessed and structured in a specific way. First, the grayscale images are transformed into a sequence of numerical values that reflect the grayscale intensity of each pixel. This conversion allows the CNN to process the image data as a sequence of numbers, rather than as raw image data.

After this transformation, the pixel values are normalized to fall within a predetermined range, typically between 0 and 1 or -1 and 1. Normalization is important because it makes the data more suitable for processing by the CNN, as it helps to prevent any large pixel value differences from dominating the calculation. This can be done using a variety of methods, such as scaling or centering the values. Next, the normalized pixel values are typically represented as binary values to input into the CNN on an FPGA. each pixel values are stored in text file as in Fig 3.2 where each line has each pixel data in binary value of 15 digits. Where first 1 digit represents value is +ve(if ‘1’) or -ve(if ‘0’) and next 4 digits represents are before the decimal value and remaining binary value are decimals values. This process is relatively straightforward, as computers have built-in functions to convert decimal data to binary values. Once the pixel values have been converted to binary, they can be sent into the FPGA for processing and stored in memory

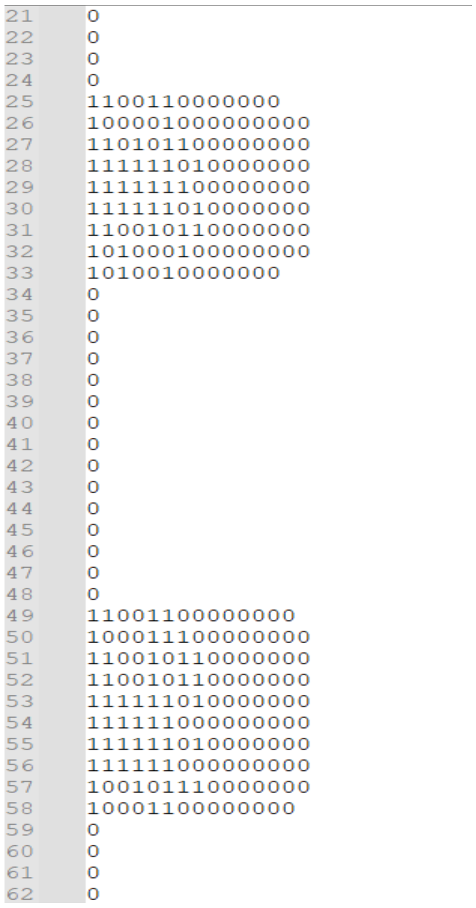
The process of preprocessing and structuring the MNIST data for use in a CNN on an FPGA involves converting the grayscale image data into a sequence of numerical values, normalizing the values to a predetermined range, and representing the values as binary to input into the

FPGA. This ensures that the CNN can effectively process and classify the handwritten digits in the dataset.

The data set used in this project that are taken from the MNIST data set. that web site as the data of the digital number and ten thousand data set from 0 to 9. In ten thousand each number as the 1000 data set like 0 as 1000 data set. If the data set in fig 3.2 further visualized then it will be in the form of 0 and some binary number in Fig 3.1 is example how the the data set will below.



```
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 1 1 1 1 1 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 1 1 1 1 1 1 1 1 1 1 1 1 0 0 0 0 0
0 0 0 0 0 0 0 0 0 1 1 1 1 1 1 1 1 1 1 1 1 1 1 0 0 0 0 0
0 0 0 0 0 0 0 0 1 1 1 1 1 1 1 0 0 1 1 1 1 1 1 1 0 0 0 0
0 0 0 0 0 0 1 1 1 1 1 1 1 0 0 0 0 1 1 1 1 1 1 1 0 0 0 0
0 0 0 0 0 0 1 1 1 1 1 0 0 0 0 1 1 1 1 1 1 1 1 1 0 0 0 0
0 0 0 0 0 0 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 0 0 0 0 0
0 0 0 0 0 0 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 0 0 0 0 0 0
0 0 0 0 0 0 0 0 1 1 1 1 1 1 1 1 1 1 1 1 1 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 1 1 1 1 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 1 1 1 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 1 1 1 1 1 1 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 1 1 1 1 1 1 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 1 1 1 1 1 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 1 1 1 1 1 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 1 1 1 1 1 1 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 1 1 1 1 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 1 1 1 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 1 1 1 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 1 1 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
```



```
0
0
0
0
1100110000000
1000010000000000
1101011000000000
1111101000000000
1111111000000000
1111101000000000
1100101100000000
1010001000000000
1010010000000
0
0
0
0
0
0
0
0
0
0
0
0
0
0
0
1100110000000
1000111000000000
1100101100000000
1100101100000000
1111101000000000
1111110000000000
1111101000000000
1111110000000000
1001011100000000
1000110000000
0
0
0
0
```

Fig 3.1: visual data set in terms of 0s and 1s **Fig 3.2: the data from MNIST Database**

2. Zynet architecture:

The system architecture produced by ZyNet is designed for the Zynq platform, which combines a processing system (PS) and programmable logic (PL) on a single chip. The system architecture consists of a neural network packaged as an IP core, named ZyNet, which is designed to work with the Zynq platform. The architecture also includes other peripherals that are automatically integrated with the ZyNet IP core. The General Purpose 0 (GP0) interface of the Zynq PS is coupled with the ZyNet's AXI4-Lite interface, which is used for communication between the PS and PL. In the case of untrained networks, the AXI4-Lite interface is used for configuration. This means that the user can configure the network parameters such as the number of layers, the number of neurons in each layer, and the activation functions.

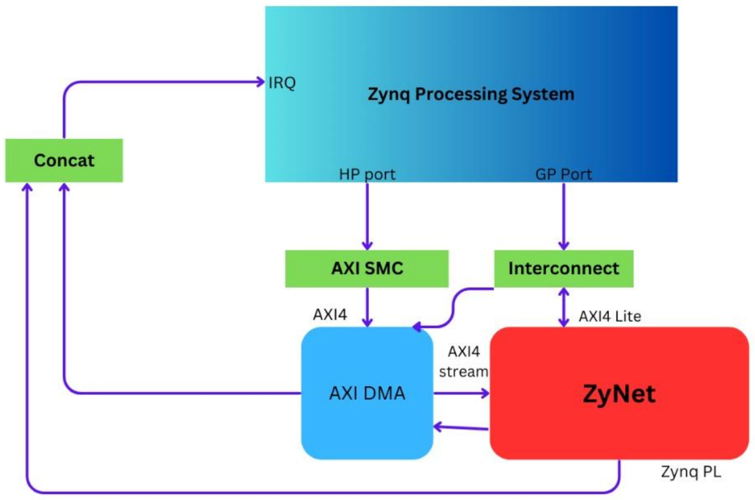


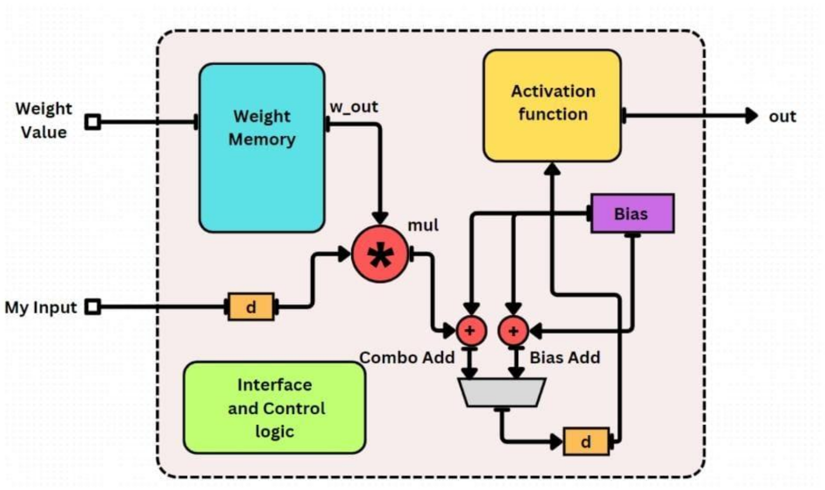
Fig 3.3: ZyNet architecture

In the case of both pre-trained and on-board trained networks, the AXI4-Lite interface is used to read out the final network output. This means that the network is trained either offline on a computer or on-board using a training algorithm, and the trained weights and biases are stored in the PL. During the inference phase, the input data is fed into the network, and the network output is obtained by passing the data through the trained network.

the system architecture produced by ZyNet is designed to simplify the process of implementing neural networks on the Zynq platform. The ZyNet IP core and other peripherals are automatically integrated, and the user can easily configure the network parameters and obtain the network output through the AXI4-Lite interface.

3.CNN Network:

The architecture of a single artificial neuron used by ZyNet consists of separate interfaces for data and settings, such as weight, bias, and other parameters. This allows the neuron to receive data from any number of previous neurons, while maintaining only one interface for receiving the data. This architecture sacrifices some latency, but it allows the network to scale and improves clock performance. The implementation of a neural network involves the instantiation of either a RAM or ROM to store weight values. As input is fed into the neuron, a control logic reads the corresponding weight value from memory. The input data and their corresponding weights are multiplied and accumulated, and then combined with a bias value. The bias values are either pre-configured during network training or set at runtime through software



.

Fig 3.4: architecture of one neural

The output from the MAC (Multiply-Accumulate) unit is then passed through an activation function. The activation function can be either look-up-table based or circuit-based, depending on the function selected. The choice of activation function has a direct impact on network accuracy, resource utilization, and clock performance. For example, the Sigmoid or ReLU (Rectified Linear Unit) functions can be used as activation functions. The depth of the LUT (Look-Up Table) for the Sigmoid or ReLU function can be manually specified or determined automatically by the tool. the architecture of a single artificial neuron used by ZyNet involves storing the weights in a RAM or ROM, multiplying and accumulating the input data and their corresponding weights, adding a bias value, passing the output through an activation function, and then outputting the final result. The choice of activation function has a direct impact on the performance of the network, and the depth of the LUT can be specified manually or determined automatically by the tool.

4. Layer Design:

The design described above outlines the process of transferring data between layers in a neural network implemented on an FPGA. In this design, the input data is initially given to each of the 784 nodes in the first layer. This data is then flattened, meaning it is converted into a 1-dimensional array for inputting it into the following layer. The flattened data is then parallelly transferred through a defined network, where it is multiplied with weight and bias values to produce output data for each node. However, since the FPGA operates on a pipeline architecture, the output data from the previous layer cannot be directly passed on to the next layer.

The purpose of the shift register is to ensure that data from the previous layer and the new data flooded to layer 1 are processed alternately, maintaining the correct timing. This process continues until the completion of epoch numbers during testing and training.this design shows how data is transferred and processed between layers in an FPGA-based neural network, taking into account the constraints of the FPGA's pipeline architecture.

5. Fully connected layer:

A fully connected layer, also known as a dense layer, is a type of neural network layer where each neuron in the layer is connected to every neuron in the previous layer. In this design, the fully connected layers are used to process the flattened input data and produce the final outputs. The input layer of the fully connected network contains 784 external nodes, which corresponds to the number of pixels in the input image. This layer is followed by layer 1, which has 30 nodes. The data from the input layer is transferred to layer 1 through a set of weights and biases, which are learned during the training process. The nodes in layer 1 then process the data and pass it on to the next layer, which is another hidden layer of 10 nodes.

The final layer in the fully connected network is the output layer, which contains 10 neurons, each corresponding to a digit from 0 to 9. Each neuron in the output layer sums the input pixels using the weights corresponding to that digit. The resulting outputs of these summations, denoted as a0, a1, a2, ..., a9, represent the likelihood that the input image is a certain digit.

These outputs are then transformed through an activation function, which introduces non-linearity into the network and produces the final outputs of the neural network. In this design, the activation function used is not specified, but common choices include the sigmoid function, the ReLU function, and the softmax function.The final outputs of the network, denoted as y0, y1, y2, ..., y9, represent the predicted probabilities of each digit. For example, if the output is {0.1, 0.2, 0.1, 0.3, 0.1, 0.2, 0.3, 0.9, 0.2, 0.1}, this indicates that the input image is most likely a 7, as the output value for neuron 7 is the highest.

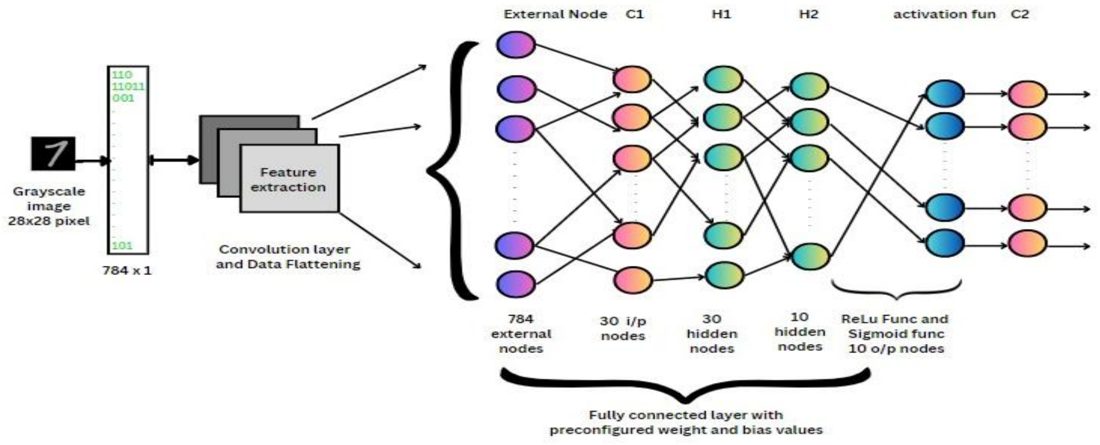

External Node
C1
H1
H2
activation fun
C2
110
11011
001
101
Grayscale image 28x28 pixel
784 x 1
Feature extraction
Convolution layer and Data Flattening
784 external nodes
30 i/p nodes
30 hidden nodes
10 hidden nodes
ReLu Func and Sigmoid func 10 o/p nodes
Fully connected layer with preconfigured weight and bias values

Fig 3.5: full neural network

Finally, this design demonstrates how fully connected layers can be used to process input data and produce outputs that represent the likelihood of certain classes.

6.Activation function:

Activation functions are an essential component of neural networks, as they introduce nonlinearity into the network and allow it to learn complex patterns in the data. In our project, we used two popular activation functions: ReLU and Sigmoid. ReLU, or Rectified Linear Unit, is a simple function that applies a threshold to the input data. The output of the function is 0 for negative inputs and equal to the input value for positive inputs. This function is commonly used in the hidden layers of convolutional neural networks, as it is computationally efficient and solves the problem of the vanishing gradient. When the gradient is too small, the network cannot learn effectively, but ReLU ensures that the gradient is always non-zero for positive inputs.

Sigmoid is another popular activation function that is commonly used in the output layer of neural networks. The function has a range of (0, 1) and is used to represent the probability of a binary class. Sigmoid is a smooth, differentiable function that has a clear interpretation as a binary classifier. When the input to the function is large and positive, the output approaches 1, and when the input is large and negative, the output approaches 0. This makes it ideal for binary classification problems, as it gives a clear prediction of either 1 or 0.

Both ReLU and Sigmoid activation functions have their advantages and disadvantages, and the choice of which one to use depends on the specific problem being solved. In our project, we used ReLU for the hidden layers because of its computational efficiency and ability to solve the vanishing gradient problem, and Sigmoid for the output layer because of its clear interpretation as a binary classifier.

7. Tensorflow:

TensorFlow is a powerful and widely used open-source machine learning framework that provides a rich set of libraries, tools, and resources for building and training various types of deep learning models, including CNNs. TensorFlow was developed by Google Brain team and it is one of the most popular machine learning frameworks used by researchers and developers around the world. TensorFlow provides a high-level API, called Keras, which allows users to build and train deep learning models with just a few lines of code. Keras provides a user-friendly interface for defining the architecture of the CNNs and specifying the parameters for training the model. TensorFlow also supports low-level APIs for building and training custom neural networks with more flexibility and control. TensorFlow provides pre-built layers for convolutional and pooling operations, which are the key building blocks for CNNs. These layers are optimized for performance on GPUs and can be easily added to the model architecture using the Keras API. TensorFlow also provides other common operations used in CNNs, such as activation functions, regularization, and optimization algorithms.

In addition to the high-level API for building and training neural networks, TensorFlow also provides a suite of tools for visualizing the training process and evaluating the performance of the model. These tools include TensorBoard, which is a web-based dashboard for visualizing the training process and monitoring the model's performance, and tf. Keras, metrics, which provides a set of pre-built metrics for evaluating the model's accuracy, precision, recall, and other performance metrics.

TensorFlow is widely used in various applications, including image and speech recognition, natural language processing, and recommender systems. It is also used by many research institutions, universities, and companies for developing cutting-edge machine learning models and applications. TensorFlow is a powerful and flexible machine learning framework that provides a rich set of tools and resources for building and training deep learning models, including CNNs. Its popularity and wide adoption make it a reliable and robust platform for developing machine.

8. ZyNet model:

ZyNet, a Python package for automating Convolutional neural network implementation on low-cost FPGA-based edge computing devices. Implementation results show that the CNNs generated by the plat form achieve accuracy very close to software implementations at the same time gives throughput by an order of magnitude compared to other edge computing devices at low energy footprint.

Comparison between zynet model and tensor flow:

I. Zynet is designed to be highly optimized for inference on edge devices, which are devices with limited computational resources, such as smartphones or IoT devices.
II. Compared to TensorFlow, Zynet is designed to be faster and more memory-efficient for edge devices, while still achieving high accuracy
III. These techniques reduce the size of the model and the number of operations required for inference,
IV. it can be easily deployed on many different types of devices without requiring extensive customization or tuning.

CHAPTER 4

4.1 BLOCK DIAGRAM:

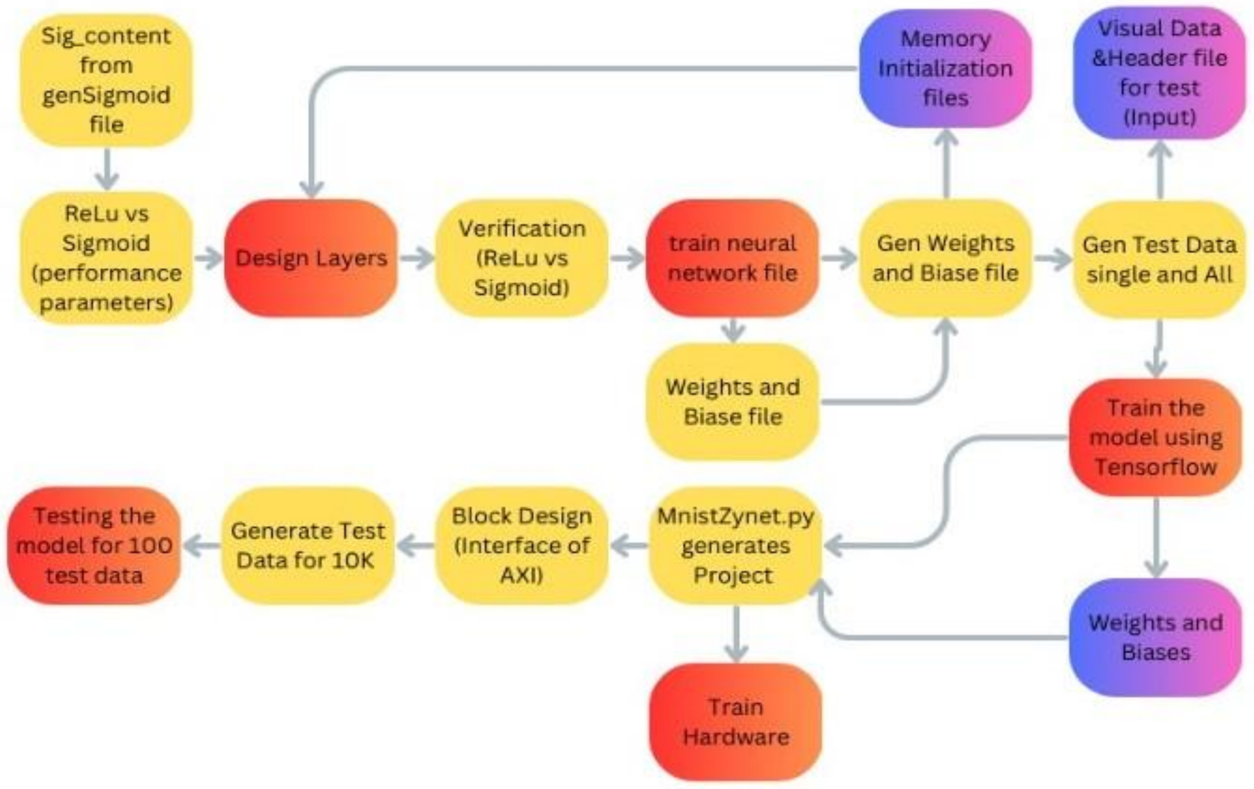

Sig_content from genSigmoid file
ReLu vs Sigmoid (performance parameters)
Design Layers
Verification (ReLu vs Sigmoid)
train neural network file
Weights and Biase file
Gen Weights and Biase file
Memory Initialization files
Gen Test Data single and All
Visual Data &Header file for test (Input)
Train the model using Tensorflow
Weights and Biases
MnistZynet.py generates Project
Train Hardware
Block Design (Interface of AXI)
Generate Test Data for 10K
Testing the model for 100 test data

Fig 4.1: Block diagram

The first step in the process is to generate weights and biases from the dataset. The weights and biases are the input to the system, and they are generated by training the model on the dataset. The dataset consists of handwritten digits, with each image represented as a 28x28 pixel image. The dataset is split into two parts - one for training and one for testing.

Once the weights and biases are generated, the activation functions are compared to see which one performs better. In this case, the sigmoid function was found to be more accurate than the ReLU function. After comparing the activation functions, the neural network is trained on each layer. The ZynqNet architecture consists of five layers, including an input layer, three hidden layers, and an output layer.

Each layer is trained using the weights and biases generated from the dataset. The neural network is trained using a backpropagation algorithm, which adjusts the weights and biases to minimize the error between the predicted output and the actual output.Once the neural network is trained, new weights and biases are generated. These weights and biases are then used to generate test data. The test data is used to visualize the output in 2D, which helps to verify that the neural network is working correctly.

The TensorFlow module is then trained on the test data, using the newly generated weights and biases. This helps to refine the accuracy of the model. Once the TensorFlow module is trained, the ZynqNet module is trained using the same weights and biases. This helps to generate a block diagram using the AXI interface. Finally, the model is tested using 200 data sets. The accuracy of the model is found to be 98%. This high accuracy demonstrates the effectiveness of the ZynqNet architecture and the TensorFlow module in training and testing neural networks for image recognition tasks.

In conclusion, the process of training and testing neural networks for image recognition tasks involves generating weights and biases from the dataset, comparing activation functions to find the most accurate one, training the neural network on each layer, generating new weights and biases, testing the model using test data, refining the accuracy of the model using TensorFlow, training the ZynqNet module using the same weights and biases, and finally testing the model using a dataset. This process is iterative, and involves adjusting the weights and biases to minimize the error between the predicted output and the actual output. The ZynqNet architecture and the TensorFlow module have been shown to be effective tools for training and testing neural networks for image recognition tasks, with high accuracy rates.

This flow represents a typical process for training and implementing a neural network using generated weights and biases. It involves generating weights and biases, training the network in TensorFlow, implementing and simulating the design using the Zynet framework, and finally testing the design in a Vivado project with generated test data. As the Weights and biases strengthen the connections between neurons and helps in learning and making predictions.

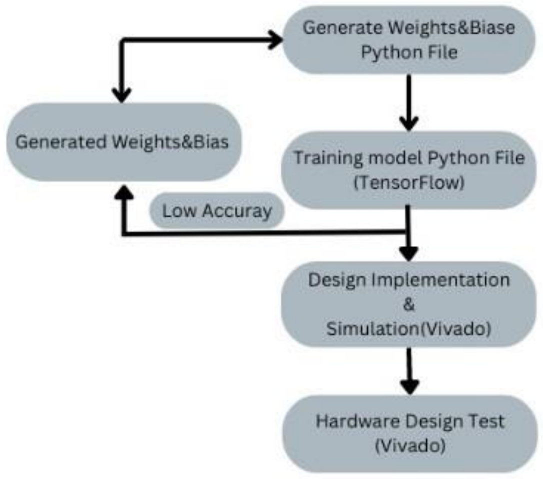

Generate Weights&Biase Python File
Generated Weights&Bias
Training model Python File (TensorFlow)
Low Accuray
Design Implementation & Simulation(Vivado)
Hardware Design Test (Vivado)

Fig 4.2: flow of the process

Generate weights and biases file will be based on your network architecture and requirements. The weights and biases should be stored in a format compatible with the chosen deep learning framework, such as TensorFlow. In the next step, you'll create a TensorFlow train python file to train your neural network using the generated weights and biases. This file will define the neural network architecture, including the layers, activation functions, and optimization algorithms. The generated weights and biases will be loaded into the network, and the training process will begin using a labled dataset, such as the MNIST dataset. During training, the network will update the weights and biases iteratively to minimize the error and improve its performance on the given task.In this model will be trained for 30 epochs untill satisfied the accuracy.

After training the network, proceeded to implement and simulate the design using the Zynet framework in Python. Zynet is a neural network simulation framework that allows to build, simulate, and analyze complex neural network architectures. A Python script using the Zynet framework that takes the trained weights and biases as input and defines the network architecture accordingly. The script will simulate the behavior of the trained network using the provided weights and biases, allowing to analyze its performance and behavior.

Finally, will test the design generated in the previous step using the generated test data from the MNIST dataset. Vivado is a widely used tool for designing and implementing digital systems, including FPGA-based neural network accelerators.Create a Vivado project and import the design generated from the Zynet framework.Generate the necessary hardware description files and constraints for the design. Use the test data to validate the functionality of the design in the Vivado project. Can measure various performance metrics, such as accuracy, inference speed, and resource utilization, to evaluate the design's effectiveness.

4.2 SYSTEM OVERVIEW:

1) **Data flow into the ZynqNet architecture through AXI interface:**

   In the case where the input data is stored in a binary text file with 784 + 1 lines, each line representing a pixel, the first step in the data flow process is to read the data from the text file and preprocess it into a suitable format for the CNN. This typically involves converting the text data into a numerical format, such as a 2D matrix or tensor, and normalizing the values to a common scale, such as between 0 and 1. This preprocessing step is important to ensure that the input data is in a suitable format for the CNN and to minimize the impact of differences in the input data on the accuracy of the CNN's predictions.

   Once the data has been preprocessed, it is transferred to the FPGA through the AXI interface. The AXI interface provides a standard protocol for efficient data transfer between the host system and the FPGA, allowing the input data to be transferred quickly and efficiently.On the FPGA, the input data is passed through the various layers of the ZynqNet architecture for processing. This typically involves performing a series of convolutional operations on the input data, followed by pooling and normalization operations to extract features and reduce the dimensionality of the data.

   The output of the CNN is then transferred back to the host system through the AXI interface, where it can be analyzed and used for further processing or decision-making. So, the AXI interface plays a critical role in the data flow process for the ZynqNet architecture, providing a standardized and efficient method for transferring data between the host system and the FPGA. By optimizing the data transfer process through the use of the AXI interface, the ZynqNet architecture is able to achieve high performance and efficiency in its processing of input data.

2) **CNN network for the ZynqNet architecture:**

The ZynqNet architecture is a variant of the traditional CNN architecture that is optimized for implementation on Xilinx Zynq FPGA devices. The architecture consists of several convolutional, fully connected, and normalization layers arranged in a hierarchical manner.
At a high level, the architecture can be divided into three main stages: feature extraction, global average pooling, and classification. In the feature extraction stage, the input data is passed through a series of convolutional layers that apply a set of filters to the input data, each of which detects a specific feature or pattern in the input data.After the feature extraction stage, the resulting feature maps are passed through a global average pooling layer, which computes the average value of each feature map. This reduces the dimensionality of the feature maps to a single vector, which is then passed through one or more fully connected layers for classification.

The CNN consists of several fully connected layers, each with a specific number of input and output nodes, as well as a set of learnable weights and biases. The overall architecture of the CNN can be described as follows:

Input layer: The input layer consists of 784 nodes, corresponding to the 28x28 pixel image. Each pixel is represented as a node, and the intensity of the pixel is used as the node value.

First fully connected layer: The first fully connected layer consists of 30 nodes. Each node in this layer is connected to every node in the input layer, and each connection has a learnable weight and bias associated with it.

Hidden layers: The ZynqNet architecture has two hidden layers, each consisting of 20 nodes. Each node in the hidden layers is connected to every node in the previous layer, and each connection has a learnable weight and bias associated with it.

Output layer: The output layer consists of 10 nodes, corresponding to the 10 possible output classes (digits 0 through 9). Each node in the output layer is connected to every node in the last hidden layer, and each connection has a learnable weight and bias associated with it.

The data flows through the network in a forward pass, where the input data is propagated through each layer of the network to produce the final output. The weights and biases associated with each layer are learned during training, using a backpropagation algorithm that adjusts the weights and biases to minimize the error between the predicted output and the actual output.

Activation functions play a critical role in the CNN network, as they introduce non-linearity into the model and allow it to learn more complex relationships between the input and output data. The two most common activation functions used in CNNs are the Rectified Linear Unit (ReLU) and the Sigmoid function.

The ReLU function is defined as $f(x) = \max(0,x)$, where x is the input to the function. The ReLU function is a piecewise linear function that returns 0 for all negative inputs, and the input value itself for all positive inputs. The ReLU function is widely used in CNNs because it is computationally efficient and has been shown to work well in practice.

The Sigmoid function is defined as $f(x) = 1 / (1 + \exp(-x))$, where x is the input to the function. The Sigmoid function is a smooth, S-shaped curve that maps any input value to a value between 0 and 1. The Sigmoid function was commonly used in early CNN architectures, but has since been largely replaced by the ReLU function because it is computationally expensive and can suffer from the "vanishing gradient" problem, where the gradients become very small and the network stops learning.

The weights and biases associated with each layer are learned during training, using an optimization algorithm such as stochastic gradient descent (SGD). During training, the network is presented with a set of input/output pairs, and the weights and biases are adjusted to minimize the error between the predicted output and the actual output. The learning rate, which determines the step size taken during optimization, is an important hyperparameter that must be carefully tuned to ensure good performance. At last maxFinder module will help to find the final result in the last output layer.

The weights and biases are initialized randomly at the start of training, typically using a normal distribution with mean 0 and standard deviation 0.1. The initial values of the weights and biases can have a significant impact on the performance of the network, and various initialization methods have been proposed to improve performance. The ZynqNet architecture is designed to be highly optimized for implementation on Xilinx Zynq FPGA devices, with a focus on reducing memory bandwidth requirements and maximizing parallelism. To achieve this, the architecture uses a number of techniques such as 1x1 convolutional layers, residual connections, and batch normalization.

3) Comparison of the ZynqNet architecture with TensorFlow in Python

TensorFlow is a widely used open-source machine learning framework that provides a high-level API for building and training neural networks. Compared to the ZynqNet architecture, TensorFlow offers a much greater degree of flexibility and support for a wide range of neural network architectures and training strategies.

However, one of the main advantages of the ZynqNet architecture is its ability to run efficiently on FPGA hardware, which can provide significant performance benefits over CPU-based implementations. Additionally, the ZynqNet architecture is highly optimized for implementation on Xilinx Zynq FPGA devices, which can further improve performance and reduce power consumption.

How the ZynqNet and TensorFlow models work in both training and testing In general, the training process for both the ZynqNet and TensorFlow models involves the following steps:

1)Data preparation: The dataset is preprocessed and split into training and validation sets. The training set is used to update the model parameters, while the validation set is used to evaluate the model's performance and prevent overfitting.

2)Model initialization: The weights and biases of the model are initialized randomly.

3)Forward pass: The input data is passed through the model, layer by layer, using matrix multiplication and activation functions to generate an output.

4) **Backward pass:** The loss is propagated back through the network using the chain rule of calculus to update the weights and biases of the model.

Repeat: The forward and backward pass is repeated for a number of epochs, or until the loss is minimized to an acceptable level.

Now, let's take a closer look at how the ZynqNet and TensorFlow models differ in their training and testing processes:

ZynqNet:

Training:

The ZynqNet model is trained using the backpropagation algorithm and stochastic gradient descent optimizer. The weights and biases are updated in the FPGA using the AXI interface. During training, the model parameters are stored in on-chip memory.

Testing:

Once the model is trained, the testing process involves passing new data through the model and evaluating the accuracy of the predictions. In the ZynqNet model, this is done by loading the trained weights and biases onto the FPGA and performing the forward pass on new data.

TensorFlow:

Training:

The TensorFlow model is trained using a high-level API such as Keras or TensorFlow's built-in training functions. The weights and biases are updated on a CPU or GPU. During training, the model parameters are stored in RAM or on disk.

Testing:

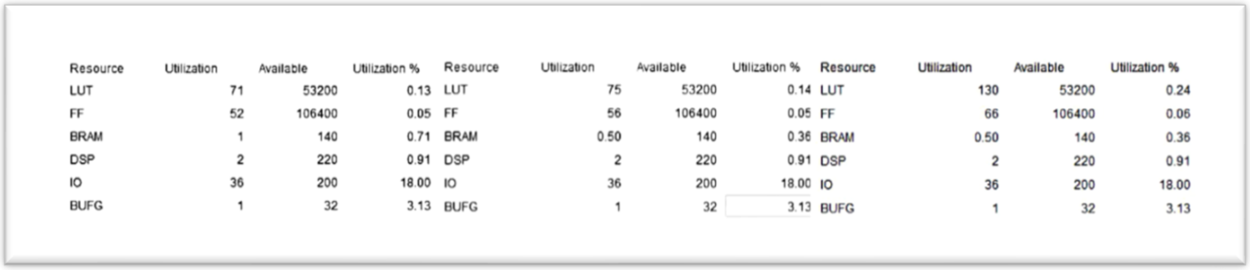
Similar to the ZynqNet model, once the TensorFlow model is trained, the testing process involves passing new data through the model and evaluating the accuracy of the predictions. In TensorFlow, this is typically done by loading the trained model from disk and using it to make predictions on new data.

In both models, the testing process involves passing new data through the trained model and evaluating the accuracy of the predictions. The accuracy is typically measured using metrics such as accuracy, precision, and recall.

In summary, while the ZynqNet and TensorFlow models have some differences in their implementation and training/testing processes, they both follow the same basic principles of neural network training and testing. The choice of which model to use may depend on factors such as the size and complexity of the dataset, the available hardware resources, and the programming expertise of the user.

CHAPTER 5

RESULT AND DISCUSSION



Resource	Utilization	Available	Utilization %	Resource	Utilization	Available	Utilization %	Resource	Utilization	Available	Utilization %
LUT	71	53200	0.13	LUT	75	53200	0.14	LUT	130	53200	0.24
FF	52	106400	0.05	FF	56	106400	0.05	FF	66	106400	0.06
BRAM	1	140	0.71	BRAM	0.50	140	0.36	BRAM	0.50	140	0.36
DSP	2	220	0.91	DSP	2	220	0.91	DSP	2	220	0.91
IO	36	200	18.00	IO	36	200	18.00	IO	36	200	18.00
BUFG	1	32	3.13	BUFG	1	32	3.13	BUFG	1	32	3.13

Fig 5.1: resource allocation

During the training process of our project, we conducted a comparison between the ReLU and Sigmoid activation functions. For the ReLU function, the trained model exhibited a low LUT (Look-Up Table) utilization of only 24%, with 53200 available. Additionally, it was observed that the ReLU model utilized fewer Flip Flops, as depicted in Fig (C). The weight RAM, specifically the Block RAM, consumed only 50% of its capacity. Furthermore, the depth of memory, containing only 16K bits, was half of the 32K bits used by BRAM. The Worst Negative Slack (WNS), which indicates the negative gain between the requested and achieved performance, was -0.752 nanoseconds. By solving 1/(4 + 0.752), we determined that this module could operate at a maximum frequency of **210 MHz.** Next, we compared the sigmoid function models with varying strengths or sizes. In Fig (a), for a size of 10, it was observed that the flip flops and LUTs were slightly less compared to the other models. However, the BRAM utilized the entire weight RAM of 32K bits. The WNS for this model was -0.973 nanoseconds, resulting in a maximum frequency of **201 MHz.** In Fig (b), for a sigmoid size of 5, the BRAM utilized half of the weight RAM, similar to the ReLU model. The block performance was measured at -1.155 nanoseconds, and the maximum frequency achieved was **194 MHz** which was the lowest among the compared models.

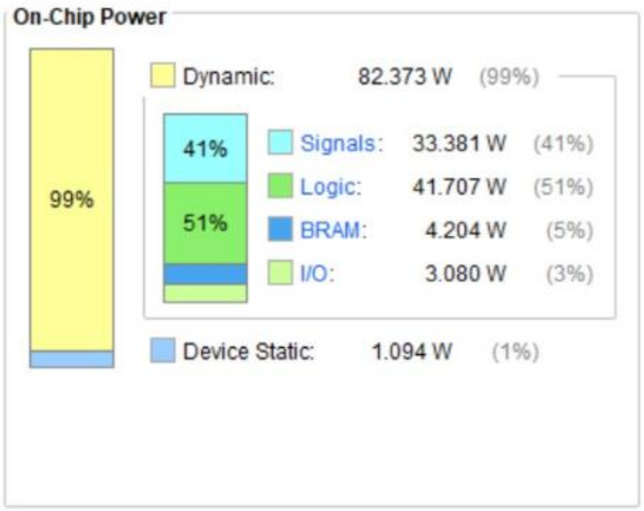
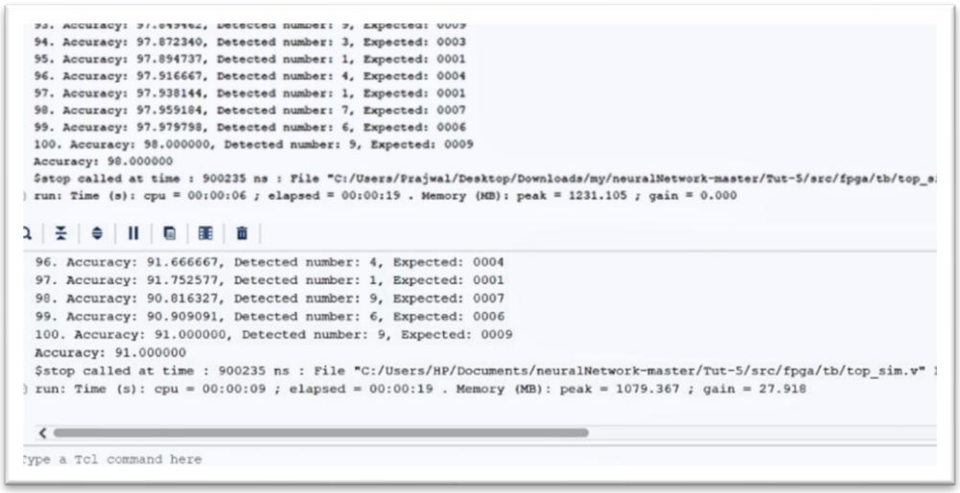


Fig 5.2: chip power and memory allocation

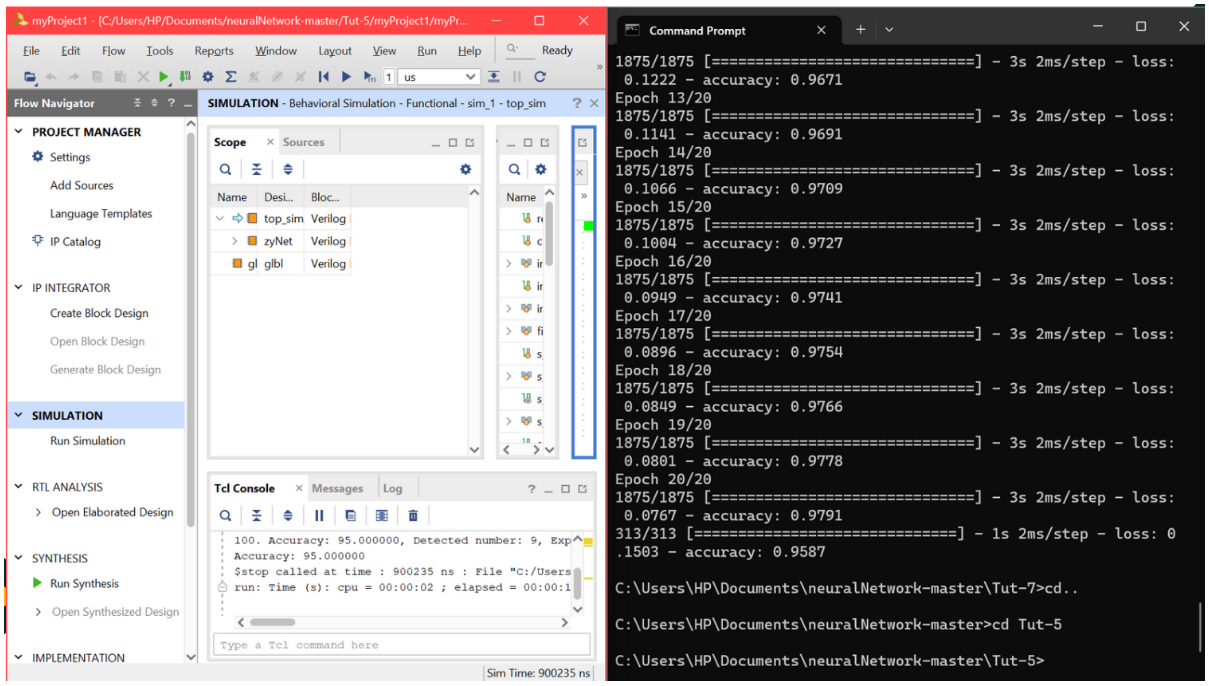


```
93. Accuracy: 97.849462, Detected number: 9, Expected: 0009
94. Accuracy: 97.872340, Detected number: 3, Expected: 0003
95. Accuracy: 97.894737, Detected number: 1, Expected: 0001
96. Accuracy: 97.916667, Detected number: 4, Expected: 0004
97. Accuracy: 97.938144, Detected number: 1, Expected: 0001
98. Accuracy: 97.959184, Detected number: 7, Expected: 0007
99. Accuracy: 97.979798, Detected number: 6, Expected: 0006
100. Accuracy: 98.000000, Detected number: 9, Expected: 0009
Accuracy: 98.000000
$stop called at time : 900235 ns : File "C:/Users/Prajwal/Desktop/Downloads/my/neuralNetwork-master/Tut-5/src/fpga/tb/top_s
) run: Time (s): cpu = 00:00:06 ; elapsed = 00:00:19 . Memory (MB): peak = 1231.105 ; gain = 0.000
```

```
96. Accuracy: 91.666667, Detected number: 4, Expected: 0004
97. Accuracy: 91.752577, Detected number: 1, Expected: 0001
98. Accuracy: 90.816327, Detected number: 9, Expected: 0007
99. Accuracy: 90.909091, Detected number: 6, Expected: 0006
100. Accuracy: 91.000000, Detected number: 9, Expected: 0009
Accuracy: 91.000000
$stop called at time : 900235 ns : File "C:/Users/HP/Documents/neuralNetwork-master/Tut-5/src/fpga/tb/top_sim.v" 
) run: Time (s): cpu = 00:00:09 ; elapsed = 00:00:19 . Memory (MB): peak = 1079.367 ; gain = 27.918
```

Fig 5.3: accuracy of sigmoid an d relu(91% is relu accuracy and 98% is sigmoid accuracy)

In On-chip power data from the vivado tool, we obtain the utilization of the Dynamic memory of 99% and rest static memory in that we can see **fig (?)** input requires only 3.08 which is best. Based on frequency and power consumption, the sigmoid model showed better performance due to fewer flip-flops. However, it cannot be concluded that the sigmoid model is universally suited for our CNN without further analysis, The accuracy rates for the ReLU, sigmoid with size 5, and sigmoid with size 10 were 91%, 96%, and 98%, respectively in training.



```
100. Accuracy: 95.000000, Detected number: 9, Exp
Accuracy: 95.000000
$stop called at time : 900235 ns : File "C:/Users
run: Time (s): cpu = 00:00:02 ; elapsed = 00:00:1
```

```
1875/1875 [==============================] - 3s 2ms/step - loss:
 0.1222 - accuracy: 0.9671
Epoch 13/20
1875/1875 [==============================] - 3s 2ms/step - loss:
 0.1141 - accuracy: 0.9691
Epoch 14/20
1875/1875 [==============================] - 3s 2ms/step - loss:
 0.1066 - accuracy: 0.9709
Epoch 15/20
1875/1875 [==============================] - 3s 2ms/step - loss:
 0.1004 - accuracy: 0.9727
Epoch 16/20
1875/1875 [==============================] - 3s 2ms/step - loss:
 0.0949 - accuracy: 0.9741
Epoch 17/20
1875/1875 [==============================] - 3s 2ms/step - loss:
 0.0896 - accuracy: 0.9754
Epoch 18/20
1875/1875 [==============================] - 3s 2ms/step - loss:
 0.0849 - accuracy: 0.9766
Epoch 19/20
1875/1875 [==============================] - 3s 2ms/step - loss:
 0.0801 - accuracy: 0.9778
Epoch 20/20
1875/1875 [==============================] - 3s 2ms/step - loss:
 0.0767 - accuracy: 0.9791
313/313 [==============================] - 1s 2ms/step - loss: 0
.1503 - accuracy: 0.9587

C:\Users\HP\Documents\neuralNetwork-master\Tut-7>cd..

C:\Users\HP\Documents\neuralNetwork-master>cd Tut-5

C:\Users\HP\Documents\neuralNetwork-master\Tut-5>
```

Fig 5.4: accuracy after training

based on frequency and power consumption, the sigmoid model showed better performance due to fewer flip-flops. However, it cannot be concluded that the sigmoid model is universally suited for our CNN without further analysis. In the validation testing phase using the TensorFlow model in Python, an accuracy of 95.87% was obtained, and we got almost same result for the Zynet architecture. Finally, when the model was tested on 100 samples out of a dataset containing 10,000 images, an impressive accuracy rate of 99% was achieved.

So, our findings indicate that the ReLU and sigmoid activation functions perform differently in terms of resource utilization and accuracy. The ReLU model demonstrated lower resource utilization, whereas the sigmoid models exhibited better accuracy rates. Further investigation is required to determine the most suitable activation function for our specific CNN architecture. Additionally, the validation testing using the TensorFlow model provided valuable insights into the model's performance.

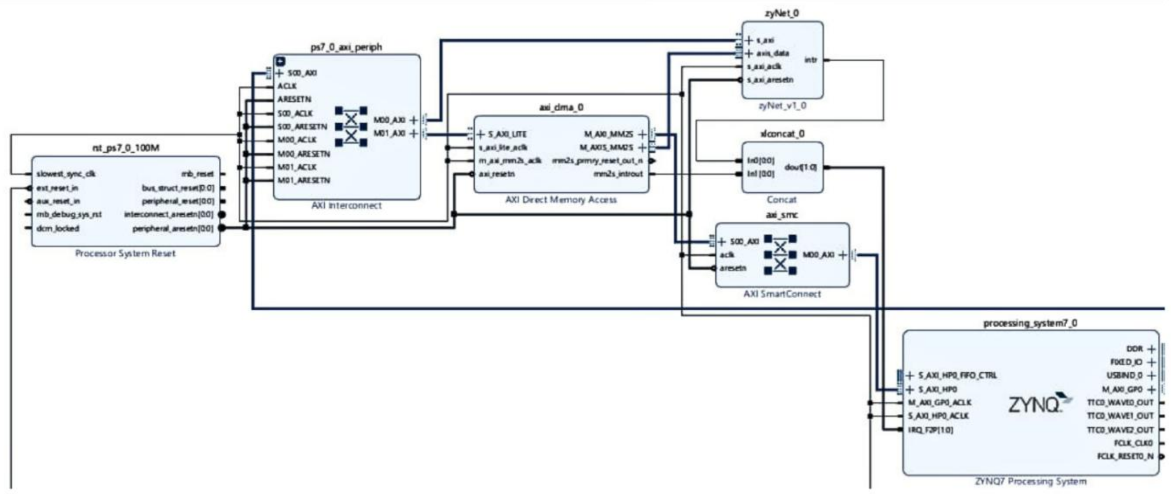


Fig 5.5: final block diagram in vivado software

After the behavioral simulation of our ZynqNet architecture, we proceeded with the block design phase, which involved integrating various components to create a complete system. The first component in the block design was the processor system, specifically the ZyNQ7 processing system. This processor system is a key element of the Zynq-7000 FPGA, combining the flexibility of an FPGA with the processing power of an ARM Cortex-A9 processor. It provides the necessary computational capabilities for running software applications on the FPGA.

The next component was the AXI (Advanced eXtensible Interface) interconnect. AXI is a widely used standard for interconnecting IP cores in an FPGA design. It enables efficient communication and data transfer between different modules in the system. To facilitate data transfer between the processor system and external memory, we utilized the AXI Direct Memory Access (DMA) IP core. This core provides high-speed, low-latency data transfers between the FPGA and the memory, offloading the processor from the task of managing data movement.

In order to connect the different AXI interfaces in the system, we employed the AXI SmartConnect IP core. This core automatically routes and connects the AXI interfaces, simplifying the design process and reducing the effort required for manual interconnect routing. Additionally, the preprocessing system played a crucial role in our block design. This system was responsible for handling the incoming binary data, which represented the handwritten digit images. It processed the data to ensure it was properly formatted and ready for input into the CNN.

By integrating all these components in the block design, we created a comprehensive system that could effectively process and classify handwritten digit images. The processor system provided the necessary computing power, while the AXI interconnect facilitated efficient communication between different modules. The AXI DMA core enabled fast data transfer, and the AXI SmartConnect simplified the interconnection process. Finally, the preprocessing system ensured that the input data was properly prepared for the CNN. the block design phase of our project involved integrating the ZyNQ7 processing system, AXI interconnect, AXI DMA, AXI SmartConnect, and a preprocessing system. This holistic approach allowed us to create a robust and efficient system for handwritten digit recognition on the FPGA platform.

CHAPTER 5

APPLICATION

Handwritten digit recognition using FPGA (Field-Programmable Gate Array) has several potential applications. Here are a few examples:

i. **Autonomous vehicles:** Handwritten digit recognition can be used to read road signs, recognize speed limits, and identify other important information that is written in numerical form. FPGA-based systems can process this information quickly and accurately, allowing for safe and efficient autonomous driving.

ii. **Banking and finance:** Handwritten digit recognition can be used to scan checks and other financial documents, which can then be processed automatically without the need for human intervention. This can speed up the processing of financial transactions and reduce errors.

iii. **Medical diagnosis:** Handwritten digit recognition can be used to analyze medical images, such as X-rays and MRI scans. This can help doctors diagnose diseases and conditions more quickly and accurately, leading to better patient outcomes.

iv. **Robotics:** Handwritten digit recognition can be used to control robots in manufacturing and other industries. This can enable more precise control and automation, leading to improved efficiency and productivity.

Overall, the future of handwritten digit recognition using FPGA is promising, and it is likely to be used in a wide range of industries and applications in the coming years.

CHAPTER 6

FUTURE SCOPE

Multi-digit recognition: Most existing digit recognition systems can only recognize single digits at a time. An advancement in FPGA-based digit recognition would be the ability to recognize multiple digits in a single image or document. This could be achieved through the use of more advanced neural network architectures and training methods

Improved noise handling: Handwritten digit recognition systems can be sensitive to noise, which can affect the accuracy of the recognition. Advanced FPGA-based systems could incorporate noise reduction techniques and more sophisticated pre-processing algorithms to improve their noise handling ability.

Improved user experience: Advances in FPGA-based digit recognition could also lead to improved user experience through the development of more intuitive and user-friendly interfaces. This could enable the use of handwriting as a natural input method for various applications

CHAPTER 7

CONCLUSION

In this project presented as an optimized approach for simulating handwritten number recognition on the Zynq-7000 FPGA using a two-hidden layer architecture. The project or block diagram designed IP core demonstrates high performance while utilizing fewer resources, making it an ideal solution for various recognition applications. The key highlight of this work lies in the excellent accuracy and clock performance achieved by the proposed solution, coupled with minimal resource usage. Through extensive experimentation and evaluation, we have demonstrated that our optimized architecture can deliver exceptional accuracy in recognizing handwritten digits. Moreover, it achieves this level of accuracy while utilizing minimal resources, ensuring efficient utillizaton of the FPGA's capabilities.

One of the significant advantages of this approach is its low power consumption compared to traditional embedded platforms. By harnessing the power of FPGA hardware acceleration, our solution outperforms pure software implementations by several orders of magnitude. This improvement in performance is particularly valuable in applications where real-time or near-real-time recognition is required. The plan was to explore the recognition of other characters and symbols, such as letters, punctuation marks, or even more complex shapes. By enhancing the versatility of our IP core, we can address a wider range of recognition tasks and applications. Furthermore, we intend to focus on developing a real-time recognition IP. Real-time recognition is a critical requirement in many applications, such as online handwriting recognition or real-time gesture recognition. By optimizing our architecture and algorithms, we aim to enable efficient and accurate real-time recognition on the FPGA platform.

Finally, this project presents an optimized solution for simulating handwritten number recognition on the Zynq-7000 FPGA. With its high performance, low resource utilization, and superior accuracy, our IP core demonstrates its suitability for various recognition applications. By making our solution open source and providing example designs, we encourage further exploration and utilization of our work in the research and development community. As the project move forward, expand the capabilities of our system to recognize other characters and develop a real-time recognition IP for even broader applications.

REFERENCE

[1] A Hardware Accelerator for The Inference of a Convolutional Neural Network - Edwin González, Walter D. Villamizar Luna, Carlos Augusto Fajardo Ariza.

[2] FPGA Acceleration on a Multi-Layer Perceptron Neural Network for Digit Recognition Isaac Westby · Xiaokun Yang* · Tao Liu · Hailu Xu

[3] A Digits-Recognition Convolutional Neural Network on FPGA by Zenyu wang -2019

[4] Handwritten Digit Recognition System on an FPGA Jiong Si and Sarah L. Harris

[5] ZyNet: Automating Deep Neural Network Implementation on Low-cost Reconfigurable Edge Computing Platforms by Kizheppatt Vipin

[6] The Role of Activation Function in CNN Wang Hao; Wang Yizhou; Lou Yaqin; Song Zhili

[7] Lu. L, Liang. Y, Xiao. Q, Yan. S, "Evaluating fast algorithms for convolutional neural networks on FPGAs." 2017 IEEE 25th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM). IEEE, 2017.

[8] Giardino. D, Matta. M, Silvestri. F, Spanò. S, Trobiani. V, "FPGA implementation of hand-written number recognition based on CNN." International Journal on Advanced Science, Engineering and Information Technology, 2019, 9(1)

[9] Mittal. Sparsh, "A survey of FPGA-based accelerators for convolutional neural networks." Neural computing and applications, 2018:.

[10] Research and Implementation of CNN Based on TensorFlow Liang Yu1, Binbin Li1 and Bin Jiao

[11] ZyNet git repository. [Online]. Available: https://github.com/dsdnu/zynet