VIETNAM NATIONAL UNIVERSITY – HOCHIMINH CITY

INTERNATIONAL UNIVERSITY

SCHOOL OF COMPUTER SCIENCE & ENGINEERING

IMPLEMENTATION OF MACHINE LEARNING FOR MNIST DIGITS RECOGNITION ON FPGA

BY

NGUYEN MINH DUC

A THESIS PROJECT SUBMITTED TO THE SCHOOL OF COMPUTER SCIENCE & ENGINEERING IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR THE DEGREE OF BACHELOR OF ENGINEERING OF INFORMATION TECHNOLOGY

HO CHI MINH CITY, VIET NAM

2025

IMPLEMENTATION OF MACHINE LEARNING

FOR MNIST DIGITS RECOGNITION ON FPGA

BY

NGUYEN MINH DUC

Under the guidance and approval of the committee, and approved by its members, this prethesis has been accepted in partial fulfillment of the requirements for the degree.

APPROVED BY:

_____ ,

Assoc. Prof. Dinh Duc Anh Vu

_____

Assoc. Prof. Vo Minh Thanh

HONESTY DECLARATION

My name is Nguyen Minh Duc. I want to declare that, aside from the mentioned references, the thesis neither uses language, ideas, nor other original material from anyone; this also is not submitted to any other educational and research programs or institutions before. I am fully acknowledged that any writings in this senior contradicted to the above statement will automatically lead to the rejection from the IT program at the International University – Vietnam National University Ho Chi Minh City.

Date: ../05/2023

Student's Signature

TURNITIN DECLARATION

Name of Student: Nguyen Minh Duc

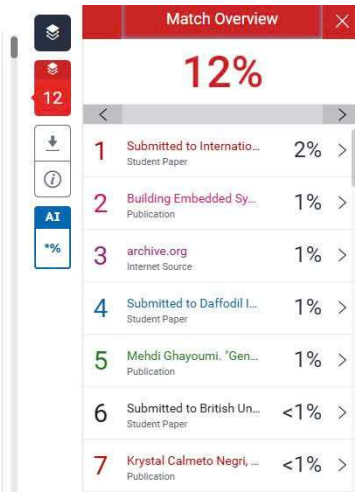
Date: 02/06/2023

INTERNATIONAL UNIVERSITY

SCHOOL OF COMPUTER SCIENCE & ENGINEERING

IMPLEMENTATION OF MACHINE LEARNING

FOR MNIST DIGITS RECOGNITION ON FPGA



Advisor Signature **Student Signature**

ACKNOWLEDGMENT

I would like to extend my sincere gratitude and deepest appreciation to the individuals and institutions whose support and contributions have helped me to the completion of this research endeavor.

First and foremost, I express my heartfelt thanks to my primary supervisor, Assoc. Prof. Dinh Duc Anh Vu, for his thoughtful suggestion and insightful feedback inspiring me towards the topic of the Special Study of the Field course. His expertise and encouragement to academic excellence have enriched the quality of this course.

I am also grateful for the valuable support from my dedicated mentor, Assoc. Prof. Vo Minh Thanh for his unwavering guidance, constructive feedback and continuous encouragement throughout the entire research process that greatly enhanced the rigor and depth of this study.

I extend my appreciation to Electronics Lab (LA2. 201) at the International University, for the effortable collaboration. The providing of access to facilities and fully functional equipment greatly contributed to the success of this project.

Finally, I am deeply grateful to my family for their emotional encouraging and understanding. Their love and belief in my abilities sustained me through the highs and lows of this journey.

In conclusion, this research stands as a collaborative effort, and I am indebted to the individuals and institutions mentioned above. Their contributions have left an indelible mark on this work, and for that, I truly appreciated.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

ABBREVIATIONS AND NOTATIONS

CNN	Convolutional Neural Network
CPU	Central Processing Unit
FPGA	Field-Programmable Gate Arrays
GPU	Graphics Processing Unit
HEX	Hexadecimal
HPS	Hard Processor System
LED	Light Emitting Diode
MLP	Multi-Layer Perceptron
MNIST	Modified National Institute of Standards and Technology
ReLU	Rectified Linear Unit
ResNet	Residual Network
SGD	Stochastic Gradient Descent
VGG	Visual Geometry Group

ABSTRACT

The goal of this project is to develop and implement a machine learning model for the recognition of handwritten digits ranging from 0 to 9, utilizing the Altera DE2 FPGA Board. The primary objective is to create an efficient system that processes input from the MNIST handwritten dataset and outputs accurate digit predictions displayed on HEX number LEDs on the Altera DE2 Board. The project begins with the preprocessing of the handwritten data to ensure optimal input quality for CNN. Following this, the CNN model is trained through two different models (VGG-16 and simple based CNN) to observe which achieves high accuracy and reliability in digit recognition. The chosen model is then trained and exported the weights and bias and sent to the DE2 Kit implementing in Verilog, enabling real-time digit recognition and display. This work not only demonstrates the feasibility of implementing machine learning solutions on FPGA hardware but also contributes to the field of digital signal processing and embedded systems. The results indicate that the proposed system successfully predicts handwritten digits with a high degree of accuracy, showcasing the potential for further advancements in hardware-accelerated machine learning applications.

Keywords: Altera DE2, FPGA, MNIST, HEX, LEDs, CNN, Verilog, Resnet50, VGG-16.

CHAPTER I

INTRODUCTION

The field of hardware acceleration for machine learning algorithms has gained significant attention in recent years due to the increasing computational demands of modern neural networks. Field-Programmable Gate Arrays (FPGAs) offer an appropriate platform for implementing these algorithms with advantages in power efficiency, speed, and customizability compared to traditional CPU and GPU implementations.

This project implements the MNIST handwritten digit dataset for classification on the Altera DE2 FPGA development board. The MNIST dataset, consisting of 28×28 pixels inside grayscale images of handwritten digits (0-9), has been a standard in the machine learning community since the beginning. By implementing neural network classifier directly into the hardware, the ability of FPGAs in accelerating is demonstrated inferencing in resource-constrained environments.

The Altera DE2 board, model Cyclone II EP2C35F672C6N, provides an approachable platform for this implementation with its abundant logic elements, embedded memory blocks, and various I/O interfaces. My implementation enhaces the board's RS232 output capabilities to tranmit serial input digits data and recognize to output the results while utilizing the onboard SDRAM to store neural network parameters.

The architecture employs a simplified neural network structure optimized for hardware implementation, with fixed-point arithmetic to balance accuracy and resource utilization. Through careful design and utilization, the system achieves real-time inference performance while maintaining accuracy compared to software implementations.

Objective 1: Theories and related works research

This objective requires theories about neural networks with specific types, the algorithm of digit recoginition using a trained model in Python. Theories and works about HDL language (Verilog HDL) and each layer's protocol design is also studied thoroughly to implement and test on the FPGA board.

Objective 2: Training CNN model

This objective focuses on choosing the appropriate model between three different models , vgg16, based simple CNN - to extract the weights and bias for convolutional and fully connected layers from the MNIST dataset using Torch, Keras and Tensorflow in Python coding language. The training is used for handwritten digit classification to implement and observe on a hardware FPGA board.

Objective 3: HDL design

This objective concentrates on designing the layers and sub-elements in the logical structure using Verilog HDL, each component plays an essential role in executing the protocol of digits recognition.

Objective 4: Finishing report

CHAPTER II

DESIGN SPECIFICATIONS AND STANDARDS

2.1. Hardware Specifications

Table 2.1. Hardware Specifications

Description	Name
Design hardware	Laptop Acer Nitro AN515-57
Verification hardware	Altera DE2 (Cyclone II EP2C35F672C6N)

2.2. Software Specifications

Table 2.2. Software Specifications

Description	Name
Hardware description language	Verilog
Software description language	Linux, Python
Coding and testing software	Quartus 9.0, ModelSim, Colab, Command Shell
Libraries	Tensorflow, Keras, Torch

CHAPTER III

PROJECT MANAGEMENT

3.1. Budget and Cost Management Plan

Table 3.1. Budget and Cost Management

	Name	**Cost**
Hardware	Laptop Acer Nitro AN515-57	25.000.000 VND
	Altera DE2 (Cyclone II EP2C35F672C6N)	Laboratory Supply
Software	Quartus 9.0	Free
	ModelSim	Free
	Colab	Free
	Command Shell	Free

CHAPTER IV

LITERATURE REVIEW

The chapter covers brief theories about background mathematical algorithms, logical structure, and CNN architecture. It also includes paper and references that relate to this project.

4.1. Theoretical background

4.1.1. Background Mathematical Algorithms

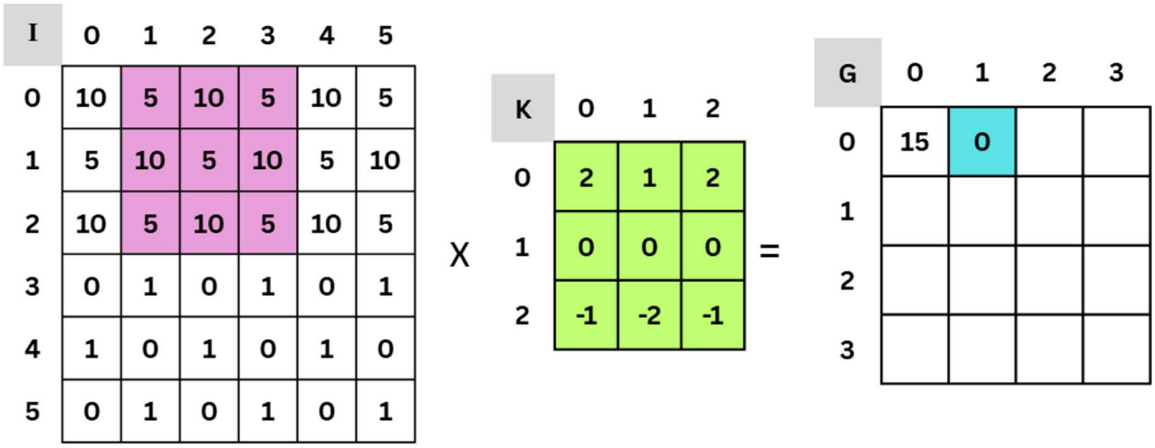
In this section, the fundamental mathematical concepts that are included in the architecture of Convolutional Neural Networks (CNNs). The concepts about convolution, max pooling, ReLU activation and fully connected layers are discussed and briefly explained below.

4.1.1.1. Convolution

A mathematical process called convolution is used to extract features from input photos. It requires executing element-wise multiplications and summations after sliding a filter (or kernel) across the input image. A feature map that emphasizes the existence of particular patterns in the image is the result. If G is the output of specific location at point (i, j), I is the input picture and K is the kernel, then the convolution operation is $I * K$ at a point (i, j) that can be expressed mathematically as follows:

$$\mathbf{G[i,j]} = \mathbf{(I * K)[i,j]} = \sum_{\boldsymbol{m}} \sum_{\boldsymbol{n}} \boldsymbol{I[i+m, j+n] \cdot K[m,n]}$$

The figure below illustrates one step of the convolution. The value of kernel can be changed to extract different features of the input photo.



G[0,1] = (I*K)[0,1] = 5*2 + 10*1 + 5*2 + 10*0 + 5*0 + 10*0 + 5*(-1) + 10*(-2) + 5*(-1) = 0

Figure 4.1.1.1. Convolutional Algorithm Example

4.1.1.2. ReLU Activation Function

The Rectified Linear Unit (ReLU) is a crucial activation function in CNNs. It is utilized to add non-linearity into the model and described as a function that enables the network to recognize intricate patterns by setting all negative values in the feature map to zero. ReLU activation reduces the vanishing gradient issue and speeds up convergence. The ReLU function is defined and plotted as below:

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

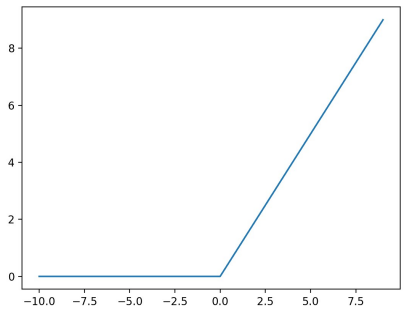


Figure 4.1.1.2. ReLU Activation Function Plot Graph

4.1.1.3. Pooling

A downsampling technique called pooling lowers the complexity of feature maps while keeping significant features. Max-pooling is one of the most popular types of pooling techniques, it chooses the highest value from a feature map region. This process facilitates the achievement of spatial invariance while lowering computing cost. A 2x2 max-pooling operation on a feature map will choose the highest value from every 2x2 region, as we can see in the following example for each colored 2x2 region.

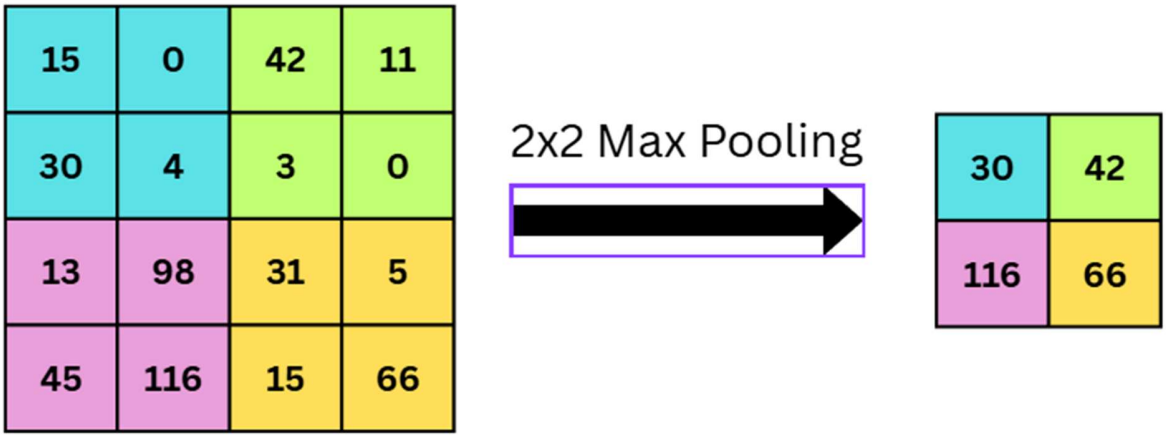


Figure 4.1.1.3. Max-pooling Example

4.1.1.4. Fully Connected Layer

The CNN architecture uses fully connected (FC) layers at the end to carry out categorization. Every neuron in an FC layer is linked to every other neuron in the layer above it. A bias term is then added after the outputs from the earlier layers have been flattened into a single vector and multiplied by a weight matrix. Mathematically, this process is defined as the function below, where x is the input vector, W is the weight matrix, b is the bias vector and y are the output vector. A follow figure demonstrates the functionality of the layer is also shown.

$$y = Wx + b$$

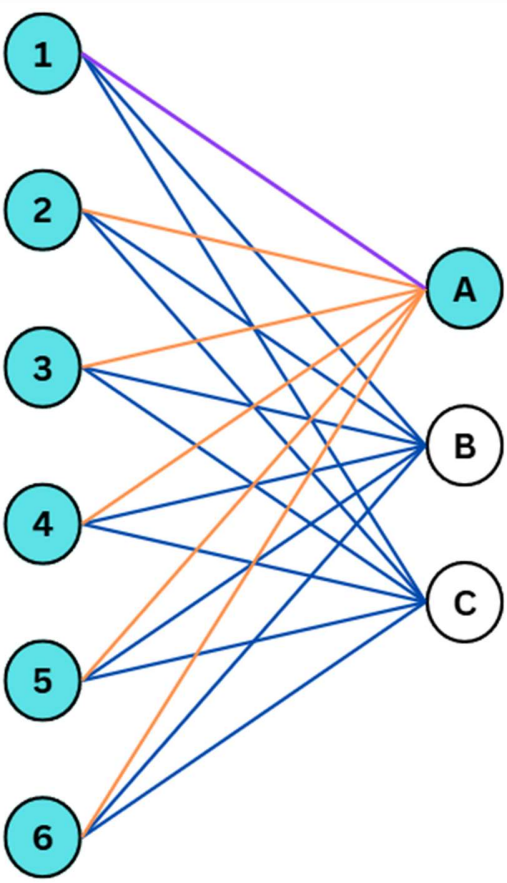


Figure 4.1.1.4. Fully Connected Layer Illustration

4.1.2. Logical Structure

The graph below displays the overall network's structure. The input image will pass through two main layer levels, each of which consists of a Convolution with different number and type of kernels, and a Max-ReLU activation layer. After that, the entire network's output will pass through the MLP layer to produce a network interpretation. Using two 5x5 kernels, the first convolution layer will convert a given 28x28 image into two channels of a 24x24 image. It will then become two channels of a 12x12 image after passing through a max-pooling layer and a ReLU activation

layer. The second convolution layer will then use three 2x3x3 kernels to transfer it to three channels of a 10x10 image. After that, it will pass through another max-pooling and ReLU activation layer, resulting in a 5x5 image with three channels. After that, it will be flattened into a 1x75 array. A fully connected layer at the end will transform the array into a 1x10 array, and the system's output will be the index pointing to the highest probability of the digit in the array.

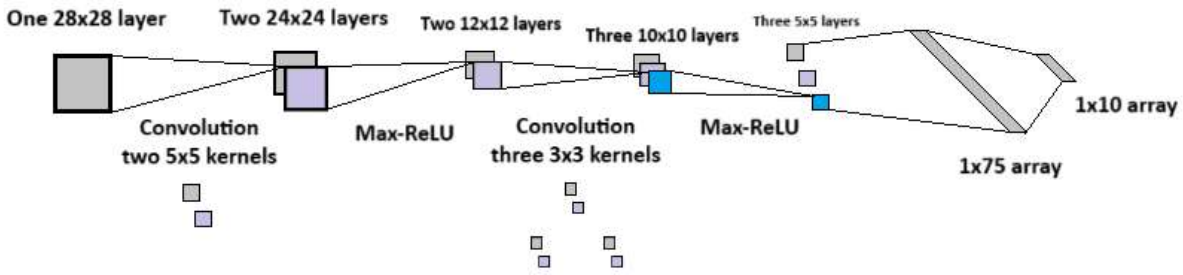


Figure 4.1.2. Logical Structure of CNN

4.1.3. CNN Architecture

4.1.3.1. MNIST Dataset

Large collections of handwritten numbers from the Modified National Institute of Standards and Technology (MNIST) database are frequently used to train different image processing programs. Additionally, the database is frequently utilized for machine learning testing and training. By modifying the samples from the original NIST datasets, it was produced. The researchers believed that NIST was not a good fit for machine learning experiments since its training dataset was derived from American Census Bureau employees, whereas its testing dataset was derived from American high school students. In addition, the NIST black and white photos were anti-aliased and normalized into a 28x28 pixel bounding box, which added grayscale levels. There are 10,000 test images and 60,000 training images in the MNIST database. While NIST's training dataset provided half of the training set and half of the test set, and NIST's testing dataset

provided the other half of the training set and the other half of the test set. A list of some of the techniques tested on the database is maintained by its original designers. They obtained an error rate of 0.8% in their original study by using a support-vector machine. In 2017, EMNIST - an expanded dataset similar to MNIST - was released. It comprises 40,000 testing photos and 240,000 training images of handwritten letters and numbers.

Figure 4.1.3.1. MNIST Example Dataset

4.1.3.2. Simple CNN Model

Since Pytorch is one of the most easy-to-use libraries, we used it to create the basic CNN based on the logical structure above. This step can also be completed using other libraries such as Keras or Tensorflow. The basic CNN with two main convolution layers and one fully connected layer after experimenting with the parameters. Since the MNIST datasets are grayscale images, the first convolution layer enlarges a single channel input into two channels of output. The size of the kernel is 5x5, its stride is 1, and its padding is 0. The input and output of the second convolution

layer are two and three channels, respectively. Each convolution layer is followed by a max pooling layer and a ReLU layer. In order to make the data flow of FPGA easier to handle, these numbers are chosen to guarantee that no padding is required at any layer. The model can attain 96% or higher accuracy on the MNIST validation set after being trained for 10 epochs with a batch size of 600. With the intention of facilitating hardware validation and debugging, test hooks are also implemented to provide logging outputs at every layer.

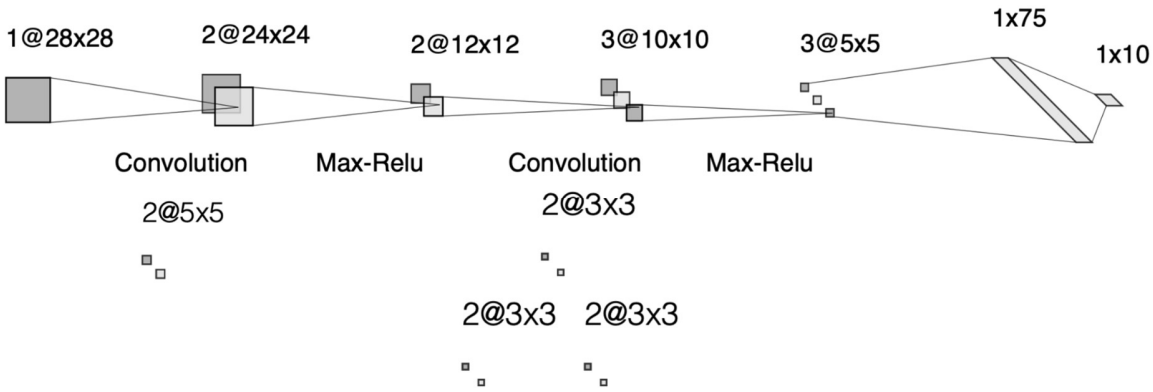


Figure 4.1.3.2. Simple CNN Architecture

4.2. Related works

4.2.1. Image classification of MNIST using VGG16

The Visual Geometry Group (VGG) at the University of Oxford proposed the VGG-16 model, CNN architecture. With 16 layers - 13 convolutional layers and 3 fully connected layers - it is distinguished by its depth. VGG-16 is well known for its ease of use and efficiency, as well as its superior performance on a range of computer vision tasks, such as object recognition and image categorization. The architecture of the model consists of a stack of convolutional layers gradually increasing in depth, followed by max-pooling layers. The model can learn complex hierarchical representations of visual characteristics thanks to this approach, producing predictions that are reliable and accurate.

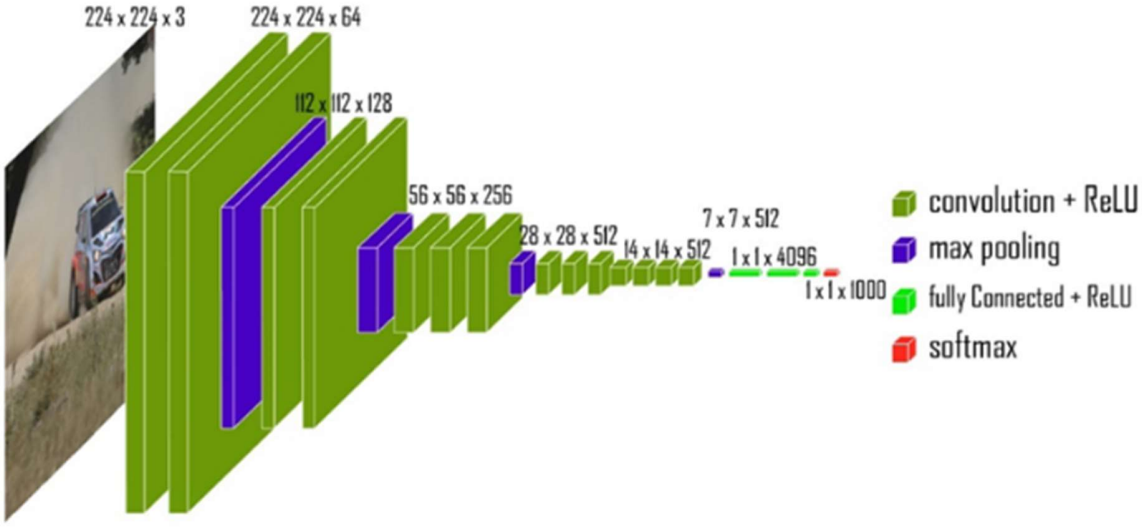


Figure 4.2.1. VGG-16 Architecture

CHAPTER V

METHODOLOGY

In this thesis project, Convolutional Neural Networks (CNNs) is improved to accelerate handwritten digit recognition by completing the architecture on FPGA. First, by thoroughly testing each network layer, Verilog is used as HDL and the verification of each module is done by using ModelSim. After ensuring the time is accurate, the Python scripts are utilized to verify its functionality. A data flow mechanism is constructed from the input image through each of these layers to the output, integrating them into a coherent system. Furthermore, PyTorch is used to train a simple CNN model using the MNIST dataset and extracted the weights and bias for the fully connected and convolutional layers. The hardware system was primed for classification tasks by loading these weights into it. By thoroughly evaluating the accuracy of the system using actual handwritten samples as well as digits from the MNIST dataset, a great increase in performance and efficiency by shifting calculation from software to specialized hardware, allowing quicker and more precise digit recognition.

5.1. Training CNN Model

To evaluate which CNN model is the most appropriate based on timing and simplicity criteria, three different CNN models are trained and compared. By using Pytorch, a simple CNN model is designed individually according to the logical structure; one pretrained CNN models are used for the digit recognition tasks with Keras and Tensorflow. Besides comparing the training time and the simplicity of the architecture, the loss and accuracy probabilities are also the standard to decide which model is used for this project.

5.1.1. Simple CNN Model

To extract the weights and bias from the CNN model used for the project, the Pytorch library is first imported. Next, the MNIST dataset is loaded to get the train and test database. Then, the neural network is designed based on the logical structure of the project including two main convolutional layers and one fully connected layer. The parameters of the neural network are optimized using Adam optimizer algorithm. After that, the data is trained through 10 epochs, the batch size is 600, each epoch contains 100 steps, the accuracy and loss probabilities are also observed to check the integrity of the training section. The model is tested through an example in the MNIST database and saved to obtain the weights and bias of each layer to load into and implemented on FPGA. The workflow and architecture of the network is shown in Figure 5.1.1.1 and 5.1.1.2 below.

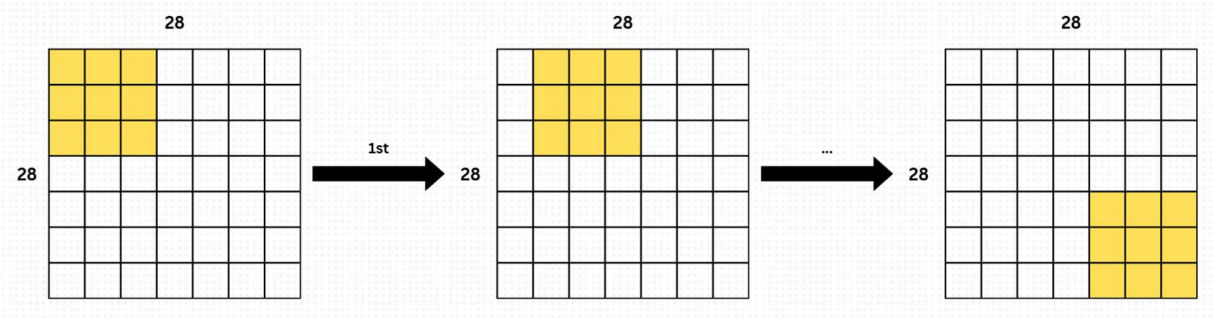


Figure 5.1.1.1. Arithmetic Process of Convolutional Layer

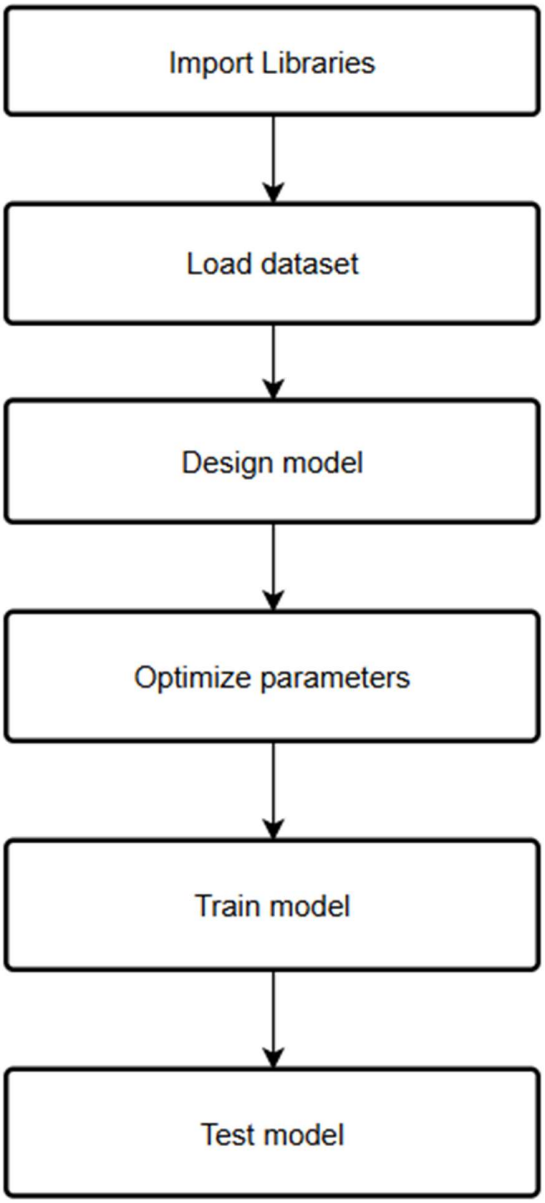


Figure 5.1.1.2. Simple CNN Model Design Workflow

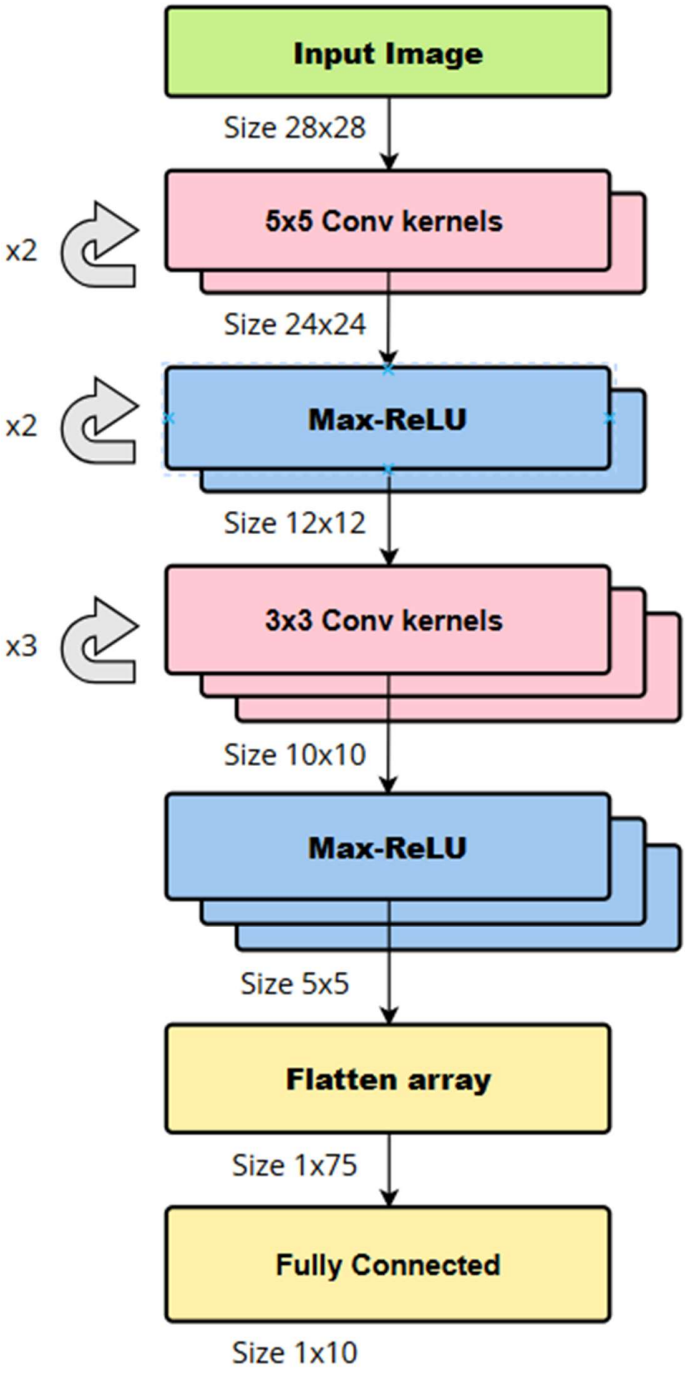


Figure 5.1.1.3. Simple CNN Model Architecture

5.1.2. VGG-16 Model

To understand the reason using the simple CNN model instead of different pretrained models, the VGG-16 model is chosen to compare the time consumption for training, the accuracy and the ability to design and implement the compatible model on the FPGA. Similar to the previous model, the libraries needed are first imported, however, the Tensorflow and Keras libraries are used due to the convenience. Next, the MNIST database is loaded to obtain the train and test dataset. However, the data loaded needs to be processed before training the model by reshaping into tensor format to match the requirement of VGG-16 model. Instead of building the neural network, the pretrained model is loaded directly from the library and then customized internal layers for better result in training. The customized model is compiled and optimized using Adam optimizer algorithm. After that, the model is fitted with 20 epochs, the batch size is 128, each epoch contains 375 steps. The next step is evaluating the model through 10000 test images. Although the digit recognition task from several examples gives higher accuracy than the simple one, the training speed and the complexity of the model with many layers are not suitable for memory inside FPGA. Figure 5.1.2.1 and 5.1.2.2 demonstrated the execution steps and architecture of the VGG-16 model.

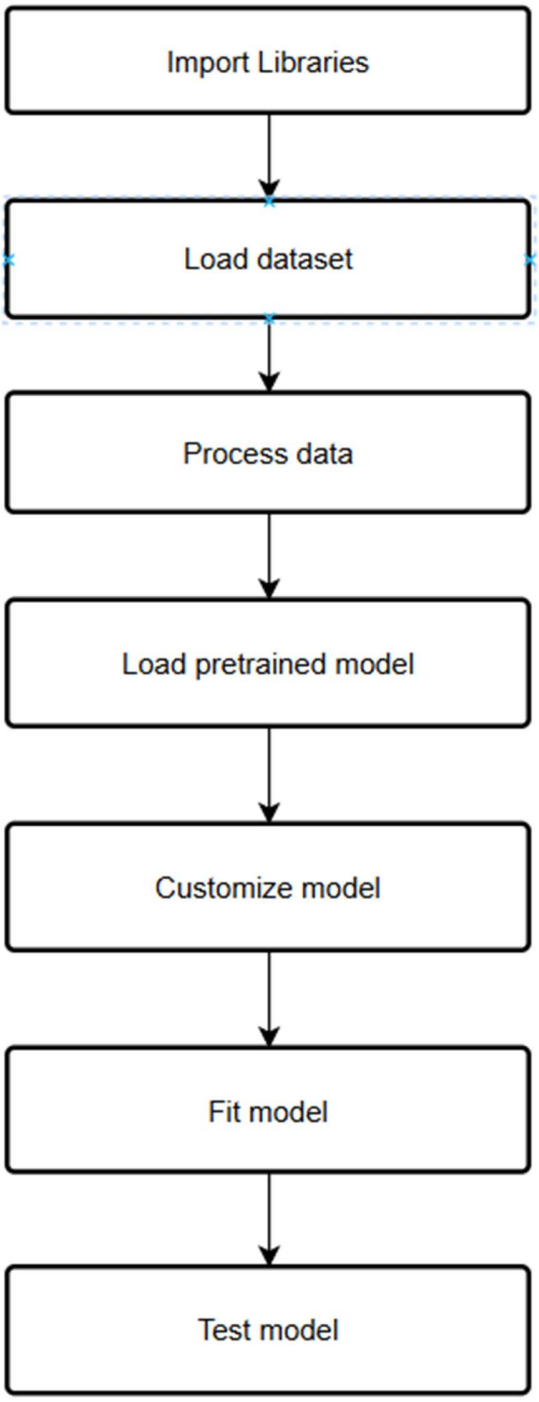


Figure 5.1.2.1. VGG-16 Model Design Workflow

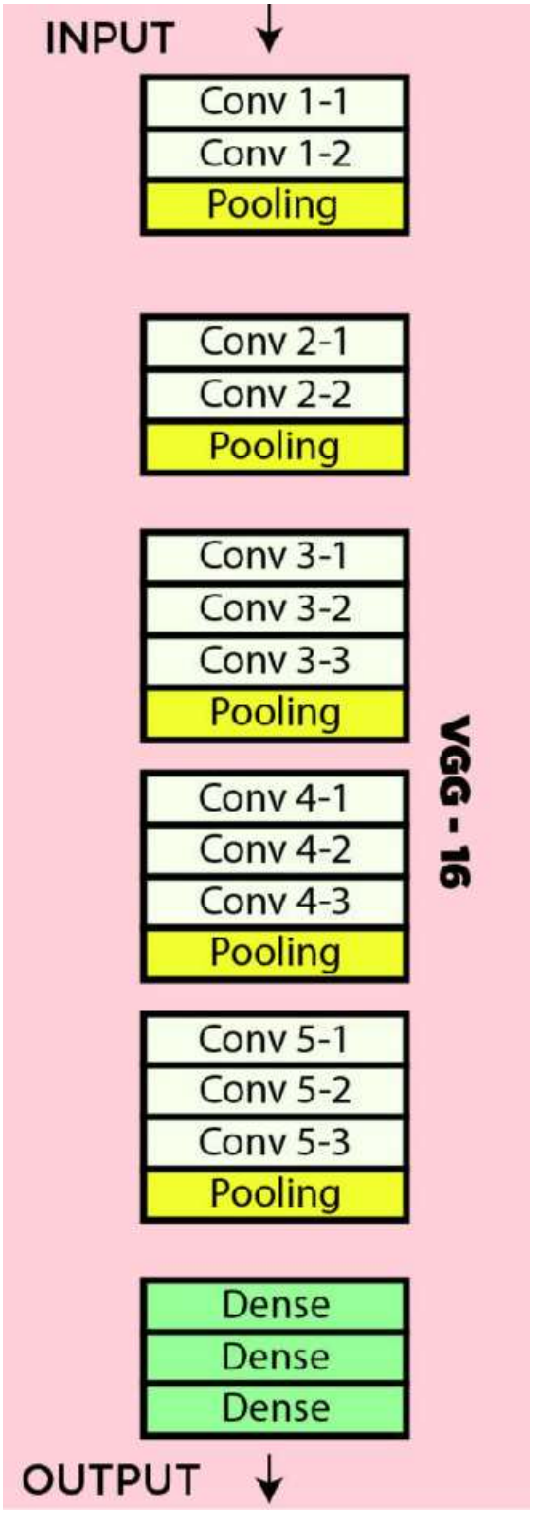


Figure 5.1.2.2. VGG-16 Model Architecture

5.2. Image Conversion

The raw image is generated by using Paint app with white color brush on black background on Windows 11. For simplicity, the image is converted into greyscale array format, the element inside the array is then flattenned to 1D and normalized to 0-255 range. After that, using the Q8.8 notation of number format to convert to signed float fixed point format with 7 digit before and 8 digit after the decimal point. Finally, the data is saved under .dat format.

Q8.8	Sign bit	7-bit integer part							8-bit fraction part						
Bit Value	-2^7	2^6	2^5	2^4	2^3	2^2	2^1	2^0	2^{-1}	2^{-2}	2^{-3}	2^{-4}	2^{-5}	2^{-6}	2^{-8}

Figure 5.2.1. Q8.8 bits declaration

5.3. Intel HEX File Generation

To load data of weights or bias into M4K RAM, which is the memory block inside the DE2 Cyclone II Board, the .dat file is not compatible, so it needs to be in the Intel HEX format, which is a file format conveying binary in ASCII text form, to be compatible with the DE2 board. Each line generated in the .hex file represents a 16-bit signed float fixed point under hexadecimal radix, which is four hexadecimal numbers, for example 00FF. A python program converting the weights or bias of each layer from .txt file exported from the model trained before into .hex file is also designed in each kernel from float format into a hex file holding 16-bit signed fixed-point hexadecimal representing the data which has 1-bit signed, 7-bit digit and 8-bit decimal.

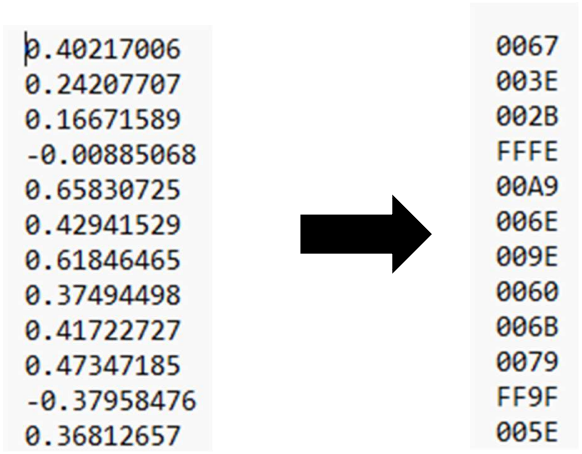


Figure 5.3.1. Q8.8 Conversion from Float to Hexadecimal

5.4. Storage Unit

5.4.1. Load weights into code using $readmemh

Using the $readmemh function, the weights and biases learned from the CNN model in Python are exported into hexadecimal (.hex) files and subsequently imported into the Verilog design. This method eliminates the requirement to hardcode precomputed parameters into the source code by enabling direct access to them by the hardware implementation. At simulation or synthesis time, internal memory arrays are read for the fixed-point representations of the model parameters included in each .hex file. The CNN on FPGA can replicate the same behavior as the software-trained model by storing weights and biases in this manner, which allows the Verilog modules to execute inference using actual model data. Scalability, modularity, and effective incorporation of machine learning models into digital hardware are guaranteed by this approach.

5.4.2. Load weights directly onto FPGA board

Another way to load weights and biases directly onto the FPGA board without recompilation is using a UART receiver module that is implemented to transfer data via an RS232 serial connection.

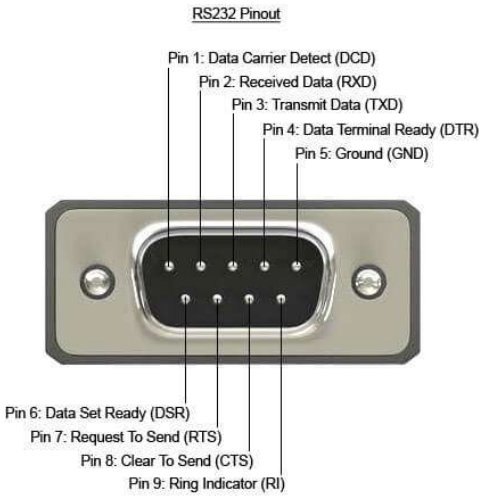


Figure 5.4.2.1. Interface of RS232 Port

In this approach, a pre-trained CNN model's parameters (weights and biases) are sent from a host computer to the FPGA in real time. The RS232 protocol enables reliable, low-speed communication over a simple two-wire interface (TX and RX), making it suitable for direct FPGA integration. On the FPGA side, the UART receiver captures incoming serial data and stores it into internal registers or RAM blocks, which are then used by the CNN computation modules. This method allows dynamic updates of model parameters, enabling reconfiguration or testing of different models without needing to resynthesize the hardware design. It provides flexibility and is particularly useful for debugging, tuning, or deploying multiple models using the same FPGA bitstream. The board and laptop are connected and communicate with each other by a RS232-to-USB Cable in the figure below.

Figure 5.4.2.2. RS232-to-USB Cable

5.5. CNN Model Layers

As listed above, there are two main convolutional layers and one fully connected layer. Inside the main convolutional layer, it contains one convolution and one Max-ReLU activate function. Some algorithms are used for better compiling such as sliding window, fixed point multipliers. Only fully connected layer module and top module requires FSM design.

5.5.1. First Convolutional Layer

Table 5.5.1. First Convolutional Module Port Declaration

Name	Port type	Bit width
IN_CHANNELS	Parameter	1
OUT_CHANNELS	Parameter	2
IN_IMG_SIZE	Parameter	28
OUT_IMG_SIZE	Parameter	24
KERNEL_SIZE	Parameter	5

DATA_WIDTH	Parameter	16
SUM_WIDTH	Parameter	DATA_WIDTH*2 + 4
clk	Input Wire	1
reset_n	Input Wire	1
start	Input Wire	1
image_in_linear	Input Wire Signed	[DATA_WIDTH * IN_IMG_SIZE * IN_IMG_SIZE * IN_CHANNELS - 1:0]
weights_linear	Input Wire Signed	[DATA_WIDTH*KERNEL_SIZE*KERNEL_SIZE* IN_CHANNELS*OUT_CHANNELS-1:0]
biases_linear	Input Wire Signed	[DATA_WIDTH*OUT_CHANNELS-1:0]
map	Output Reg Signed	[DATA_WIDTH*OUT_IMG_SIZE* OUT_IMG_SIZE*OUT_CHANNELS-1:0]
finish	Output Reg	1

5.5.2. First Max Pooling and ReLU Layer

Table 5.5.2. First MaxPool2x2ReLU Module Port Declaration

Name	**Port type**	**Bit width**
IMG_WIDTH	Parameter	24
IMG_HEIGHT	Parameter	24
CHANNELS	Parameter	2
POOL_SIZE	Parameter	2
DATA_WIDTH	Parameter	16
clk	Input Wire	1

reset_n	Input Wire	1
start	Input Wire	1
data_in	Input Wire Signed	[IMG_WIDTH*IMG_HEIGHT *CHANNELS*DATA_WIDTH-1:0]
data_out	Output Reg Signed	[(IMG_WIDTH/2)*(IMG_HEIGHT/2) *CHANNELS*DATA_WIDTH-1:0]
finish	Output Reg	1

5.5.3. First Convolutional Block

Table 5.5.3. First Convolutional Block Module Port Declaration

Name	**Port type**	**Bit width**
IN_IMG_SIZE	Parameter	28
OUT_IMG_SIZE	Parameter	24
KERNEL_SIZE	Parameter	5
DATA_WIDTH	Parameter	16
IN_ CHANNELS	Parameter	1
OUT_ CHANNELS	Parameter	2
clk	Input Wire	1
reset_n	Input Wire	1
start	Input Wire	1
image_in_linear	Input Wire Signed	[DATA_WIDTH * IN_IMG_SIZE * IN_IMG_SIZE * IN_CHANNELS - 1:0]

weights_linear	Input Wire Signed	[DATA_WIDTH*KERNEL_SIZE*KERNEL_SIZE* IN_CHANNELS*OUT_CHANNELS-1:0]
biases_linear	Input Wire Signed	[DATA_WIDTH*OUT_CHANNELS-1:0]
block_out_linear	Output Wire Signed	[DATA_WIDTH*(OUT_IMG_SIZE/2) *(OUT_IMG_SIZE/2)*OUT_CHANNELS-1:0]
finish_block	Output Wire	1

5.5.4. Second Convolutional Layer

Table 5.5.4. Second MaxPool2x2ReLU Module Port Declaration

Name	**Port type**	**Bit width**
IN_CHANNELS	Parameter	2
OUT_CHANNELS	Parameter	3
IN_IMG_SIZE	Parameter	12
OUT_IMG_SIZE	Parameter	10
KERNEL_SIZE	Parameter	3
DATA_WIDTH	Parameter	16
SUM_WIDTH	Parameter	DATA_WIDTH*2 + 4
clk	Input Wire	1
reset_n	Input Wire	1
start	Input Wire	1
image_in_linear	Input Wire Signed	[DATA_WIDTH * IN_IMG_SIZE * IN_IMG_SIZE * IN_CHANNELS - 1:0]

weights_linear	Input Wire Signed	[DATA_WIDTH*KERNEL_SIZE*KERNEL_SIZE* IN_CHANNELS*OUT_CHANNELS-1:0]
biases_linear	Input Wire Signed	[DATA_WIDTH*OUT_CHANNELS-1:0]
map	Output Reg Signed	[DATA_WIDTH*OUT_IMG_SIZE* OUT_IMG_SIZE*OUT_CHANNELS-1:0]
finish	Output Reg	1

5.5.5. Second Max Pooling and ReLU Layer

Table 5.5.5. Second MaxPool2x2ReLU Module Port Declaration

Name	**Port type**	**Bit width**
IMG_WIDTH	Parameter	10
IMG_HEIGHT	Parameter	10
CHANNELS	Parameter	2
POOL_SIZE	Parameter	2
DATA_WIDTH	Parameter	16
clk	Input Wire	1
reset_n	Input Wire	1
start	Input Wire	1
data_in	Input Wire Signed	[IMG_WIDTH*IMG_HEIGHT *CHANNELS*DATA_WIDTH-1:0]
data_out	Output Reg Signed	[(IMG_WIDTH/2)*(IMG_HEIGHT/2) *CHANNELS*DATA_WIDTH-1:0]
finish	Output Reg	1

5.5.6. Second Convolutional Block

Table 5.5.6. Second Convolutional Block Module Port Declaration

Name	Port type	Bit width
IN_IMG_SIZE	Parameter	12
OUT_IMG_SIZE	Parameter	10
KERNEL_SIZE	Parameter	3
DATA_WIDTH	Parameter	16
IN_CHANNELS	Parameter	2
OUT_ CHANNELS	Parameter	3
clk	Input Wire	1
reset_n	Input Wire	1
start	Input Wire	1
image_in_linear	Input Wire Signed	[DATA_WIDTH * IN_IMG_SIZE * IN_IMG_SIZE * IN_CHANNELS - 1:0]
weights_linear	Input Wire Signed	[DATA_WIDTH*KERNEL_SIZE*KERNEL_SIZE* IN_CHANNELS*OUT_CHANNELS-1:0]
biases_linear	Input Wire Signed	[DATA_WIDTH*OUT_CHANNELS-1:0]
block_out_linear	Output Wire Signed	[DATA_WIDTH*(OUT_IMG_SIZE/2) *(OUT_IMG_SIZE/2)*OUT_CHANNELS-1:0]
finish_block	Output Wire	1

5.5.7. Flatten Layer

Table 5.5.7. Flatten Layer Module Port Declaration

Name	Port type	Bit width
FILTER_WIDTH	Parameter	5
FILTER_HEIGHT	Parameter	5
CHANNELS	Parameter	3
DATA_WIDTH	Parameter	16
clk	Input Wire	1
reset_n	Input Wire	1
start	Input Wire	1
data_in	Input Wire Signed	[FILTER_HEIGHT*FILTER_WIDTH *CHANNELS*DATA_WIDTH-1:0]
vector_out	Output Reg Signed	[FILTER_HEIGHT*FILTER_WIDTH *CHANNELS*DATA_WIDTH-1:0]
finish	Output Reg	1

5.5.8. Fully Connected Layer

For the better understanding of the code, table 5.2.1 below clearly explains the state machine of fully connected layer including the state number, the state name and the description which describe the roles of each state.

Table 5.5.8.1. Fully Connected Layer State Machine

State	Name	Description

0	IDLE_STATE	Idle state waiting for the start signal and reseting the default value.
1	COMPUTE_STATE	Executing the algorithm of fully connected layer, using matrix dot and sliding window.
2	WRITE_OUT_STATE	Write each sum result of the calculation into an 1D output array.

Table 5.5.8.2. Fully Connected Layer Module Port Declaration

Name	**Port type**	**Bit width**
IN _SIZE	Parameter	75
OUT _SIZE	Parameter	10
DATA_WIDTH	Parameter	16
clk	Input Wire	1
reset_n	Input Wire	1
start	Input Wire	1
image_in_linear	Input Wire Signed	[DATA_WIDTH * IN_IMG_SIZE * IN_IMG_SIZE * IN_CHANNELS - 1:0]
weights_linear	Input Wire Signed	[DATA_WIDTH*KERNEL_SIZE*KERNEL_SIZE* IN_CHANNELS*OUT_CHANNELS-1:0]
biases_linear	Input Wire Signed	[DATA_WIDTH*OUT_CHANNELS-1:0]
map	Output Reg Signed	[DATA_WIDTH*OUT_IMG_SIZE* OUT_IMG_SIZE*OUT_CHANNELS-1:0]
finish	Output Reg	1

5.5.9. ArgMax Layer

Table 5.5.9.1. ArgMax Layer State Machine

State	Name	Description
0	IDLE_STATE	Idle state waiting for the start signal to reset the default value and load data into array.
1	SET_UP_STATE	Setting up the data for the next state.
2	COMPARE_STATE	Executing the algorithm of ArgMax, using comparison to get the largest value.
3	FINISH_STATE	Write each sum result of the calculation into an 1D output array.

Table 5.5.9.2. ArgMax Layer Module Port Declaration

Name	Port type	Bit width
IN_SIZE	Parameter	10
DATA_WIDTH	Parameter	16
clk	Input Wire	1
reset_n	Input Wire	1
start	Input Wire	1
class_in	Input Wire Signed	[IN_SIZE*DATA_WIDTH-1:0]
index_out	Output Reg	[3:0]
finish_argmax	Output Reg	1

5.5.10. Top Convolutional Neural Network

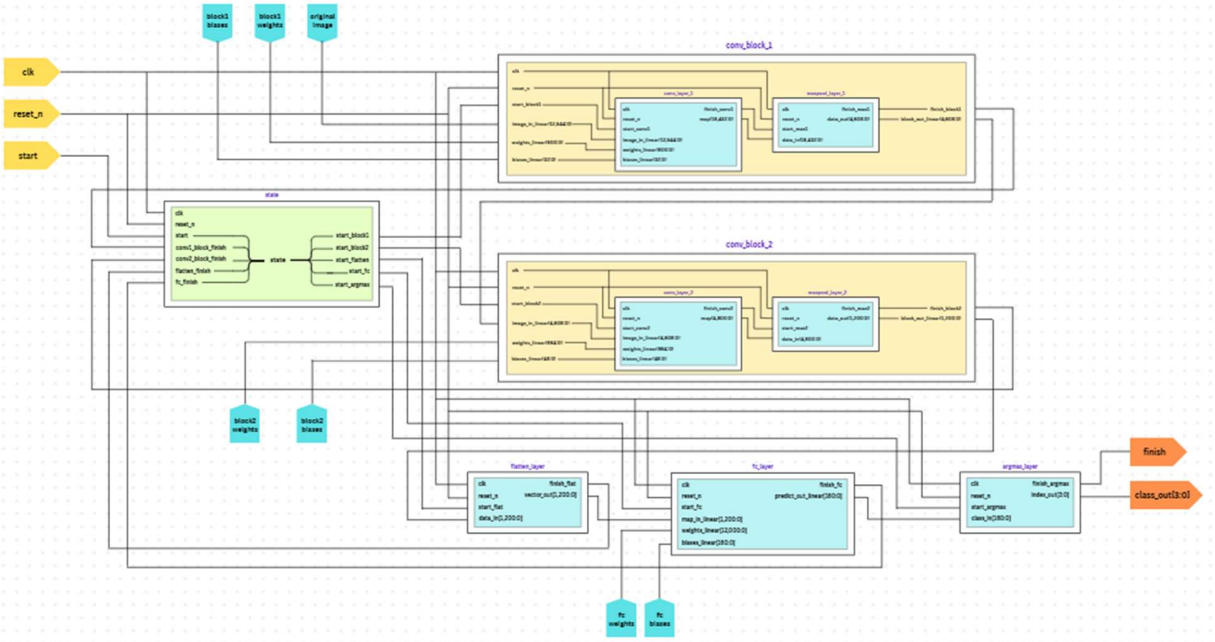


Figure 5.5.10.1. RTL Diagram of Convolutional Top Module

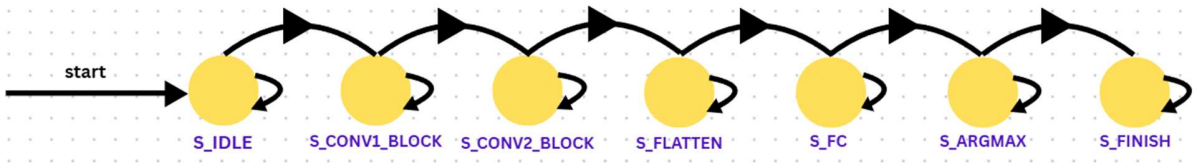


Figure 5.5.10.2. State Machine of Convolutional Top Module

Table 5.5.10. Top Convolutional Neural Network Module Port Declaration

Name	Port type	Bit width
IMG_CHANNELS	Parameter	1
IMG_WIDTH	Parameter	28
IMG_HEIGHT	Parameter	28
DATA_ WIDTH	Parameter	16

CONV1_OUT_CHANNELS	Parameter	2
CONV2_OUT_CHANNELS	Parameter	3
CONV1_ KERNEL	Parameter	5
CONV2_ KERNEL	Parameter	3
clk	Input Wire	1
reset_n	Input Wire	1
start	Input Wire	1
class_out	Output Reg	[3:0]
finish_out	Output Reg	1

CHAPTER VI

RESULTS

This chapter contains the results of training two CNN models on Colab using Torch, Keras and Tensorflow. The weights and bias of the chosen CNN is clearly shown and exported to .txt file for hexadecimal conversion. The image processing step to transform the file into Intel HEX file to be compatible with the FPGA is also explained. Each layer in CNN is first designed and synthesized using Quartus 9.0, then tested and verified by indiviual testbench using ModelSim-Altera 6.4a. The waveforms are displayed to observe the behavior and evaluate the precision.

6.1. Training CNN Model

6.1.1. Simple CNN Model

Starting by configuring the type of device in Colab, the CPU is used because of the basic architecture of the model, it does not require a high-accelerate device to compile and train.

```
In [69]: import torch
         # Device configuration
         device = torch.device('cpu')
         device

Out[69]: device(type='cpu')
```

Figure 6.1.1.1. Device Configuration

After that, the MNIST dataset is imported from "torchvision" library, the 'train_data' is the only data being trained; therefore, the train boolean value of the 'test_data' is set to 'False'. Both datasets are transformed into Pytorch tensor preparing for the consumption by Pytorch models through the 'ToTensor()' function. It also handles the scaling of the image pixel value to the range

[0.0, 1.0]. For many deep learning models to provide consistent training and improved performance, this normalization is essential.

```
Dataset MNIST
    Number of datapoints: 60000
    Root location: data
    Split: Train
    StandardTransform
Transform: ToTensor()
Dataset MNIST
    Number of datapoints: 10000
    Root location: data
    Split: Test
    StandardTransform
Transform: ToTensor()
torch.Size([60000, 28, 28])
```

Figure 6.1.1.2. Dataset Preparation

After loading the MNIST dataset, the batch size is set to 600 for both types of data and they are suffled for reputated verification.

```
In [74]: from torch.utils.data import DataLoader
         train_dataloader = DataLoader(train_data, batch_size=600, shuffle=True)
         test_dataloader = DataLoader(test_data, batch_size=600, shuffle=True)
```

Figure 6.1.1.3. Load Data Preparation

The logical structure is implemented using the ‘nn’ library, which is the neural network. The ‘Sequential’ function is used to design two main convolutional layers, each containing a convolution, a ReLU and a Max Pool activate function. The CNN module is printed to check the setting of each layer.

```
CNN(
  (conv1): Sequential(
    (0): Conv2d(1, 2, kernel_size=(5, 5), stride=(1, 1))
    (1): ReLU()
    (2): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
  )
  (conv2): Sequential(
    (0): Conv2d(2, 3, kernel_size=(3, 3), stride=(1, 1))
    (1): ReLU()
    (2): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
  )
  (out): Linear(in_features=75, out_features=10, bias=True)
)
```

Figure 6.1.1.4. Logical Structure Design

Next, optimizing the data is an important part of maximizing the ultility of the data for training. In this section, Adam optimizer with learning rate of 0.01 is chosen because of the significant outperformance in terms of both training and convergence accuracy compared with Stochastic Gradient Descent (SGD) optimizer.

```
Out[78]: Adam (
         Parameter Group 0
             amsgrad: False
             betas: (0.9, 0.999)
             capturable: False
             differentiable: False
             eps: 1e-08
             foreach: None
             fused: None
             lr: 0.01
             maximize: False
             weight_decay: 0
         )
```

Figure 6.1.1.5. Data Optimization

The model is trained through 10 epochs; each epoch has 100 steps. The loss probability is printed to observe the loss in each epoch. The total training time is 133.191s. Each step inside is commented and explained in the for loop.

```
Epoch [1/10], Step [100/100], Loss: 0.2163
Epoch [2/10], Step [100/100], Loss: 0.1585
Epoch [3/10], Step [100/100], Loss: 0.0964
Epoch [4/10], Step [100/100], Loss: 0.0865
Epoch [5/10], Step [100/100], Loss: 0.1841
Epoch [6/10], Step [100/100], Loss: 0.1079
Epoch [7/10], Step [100/100], Loss: 0.0934
Epoch [8/10], Step [100/100], Loss: 0.0679
Epoch [9/10], Step [100/100], Loss: 0.0790
Epoch [10/10], Step [100/100], Loss: 0.0862
```

Figure 6.1.1.6. Model Training

After training, the model is tested and verified by printing out the accuracy probability using the test data. The accuracy probability results in 0.96 or higher.

```
Test Accuracy of the model on the 10000 test images: 0.97
Test Accuracy of the model on the 10000 test images: 0.97
Test Accuracy of the model on the 10000 test images: 0.98
Test Accuracy of the model on the 10000 test images: 0.97
Test Accuracy of the model on the 10000 test images: 0.98
Test Accuracy of the model on the 10000 test images: 0.97
Test Accuracy of the model on the 10000 test images: 0.97
Test Accuracy of the model on the 10000 test images: 0.97
Test Accuracy of the model on the 10000 test images: 0.96
Test Accuracy of the model on the 10000 test images: 0.97
Test Accuracy of the model on the 10000 test images: 0.97
Test Accuracy of the model on the 10000 test images: 0.98
Test Accuracy of the model on the 10000 test images: 0.98
Test Accuracy of the model on the 10000 test images: 0.97
Test Accuracy of the model on the 10000 test images: 0.97
Test Accuracy of the model on the 10000 test images: 0.98
Test Accuracy of the model on the 10000 test images: 0.98
```

Figure 6.1.1.7. Accuracy Testing

The model's state_dict, which lists the shapes of all learned parameters (weights and biases) in CNN. Each line corresponds to a specific layer. The first layer, conv1.0.weight, has shape [2, 1, 5, 5], meaning it has 2 filters, each with 1 input channel and a kernel size of 5×5. Its bias term conv1.0.bias has shape [2], assigning one bias per filter. The second convolutional layer, conv2.0.weight, has shape [3, 2, 3, 3], representing 3 filters operating on 2 input channels with 3×3 kernels. The corresponding biases conv2.0.bias have 3 values, one per filter. Finally, the fully

connected output layer out.weight connects 75 input features to 10 output classes (shape [10, 75]), with out.bias contributing a separate bias value for each of the 10 output classes. These tensor shapes confirm the CNN structure and determine how the parameters are loaded and mapped in the FPGA design and are indicated in Figure 6.1.1.8 below.

```
Model state_dict:
conv1.0.weight   torch.Size([2, 1, 5, 5])
conv1.0.bias     torch.Size([2])
conv2.0.weight   torch.Size([3, 2, 3, 3])
conv2.0.bias     torch.Size([3])
out.weight       torch.Size([10, 75])
out.bias         torch.Size([10])
```

Figure 6.1.1.8. Model State Result

Figure 6.1.1.9 displays the actual trained parameters of the first convolutional layer (conv1.0) after training. The conv1.0.weight tensor has a shape of [2, 1, 5, 5], meaning there are 2 filters, each applied to 1 input channel with a kernel size of 5×5. The figure shows the values of these filters as floating-point numbers, representing how each filter responds to specific spatial patterns in the input image. These weights are used during convolution to extract low-level features such as edges or textures. Below the weights, the conv1.0.bias tensor contains two values, one bias per filter. These biases are added after the convolution operation to help the network better fit the data. These parameters will later be quantified and converted into fixed-point hexadecimal values to be loaded into the FPGA for hardware inference.

```
Parameters
conv1.0.weight
 tensor([[[[ 0.1027, -0.1600, -0.1231, -0.0204,  0.1641],
          [ 0.1342,  0.4724,  0.7942,  0.4118,  0.1202],
          [ 0.2299,  0.5250,  0.6427,  0.3914,  0.5267],
          [ 0.0911,  0.0792, -0.0830,  0.3659,  0.5461],
          [-0.0836,  0.1277,  0.2579,  0.2825,  0.4228]]],


        [[[ 0.4518,  0.1350, -0.0849,  0.2220,  0.9772],
          [ 0.6005,  0.5837,  0.6235,  0.5671,  0.0920],
          [ 0.2693,  0.8749,  1.1374,  0.4500,  0.0737],
          [-1.1765, -0.9229, -1.0997, -0.6078, -0.0682],
          [-0.4302, -0.8660, -0.5325, -0.0360,  0.0271]]]])
conv1.0.bias
 tensor([-0.0329,  0.0210])
```

Figure 6.1.1.9. First Convolutional Layer Weights and Bias

Figure 6.1.1.10 presents the trained weights and bias values for the second convolutional layer conv2.0 in the CNN model. The conv2.0.weight tensor has the shape [3, 2, 3, 3], meaning there are 3 output filters, each operating on 2 input channels with a kernel size of 3×3. This layer takes as input the output feature maps from the first convolutional layer and extracts higher-level features. Each filter is represented by two 3×3 matrices (for the 2 input channels), and their values define how the filter responds to spatial patterns across the channels. Below the weights, the conv2.0.bias tensor contains 3 values, one for each filter, which are added after the convolution operation to adjust the activation levels. These floating-point parameters are essential for maintaining the learned behavior of the CNN when deployed in hardware. Before implementation on FPGA, they are converted to fixed-point format and stored in memory (e.g., HEX or binary format) to be loaded using $readmemh or via UART communication.

```
conv2.0.weight
 tensor([[[[ 0.2220, -0.5584, -0.5548],
           [-0.2858, -0.6273,  0.3947],
           [-0.3715, -0.1783,  0.6155]],

          [[ 0.1321,  0.4670,  0.2784],
           [ 0.6525,  0.1232, -0.7473],
           [-0.3478, -0.2348,  0.0567]]],


         [[[-0.7856, -0.7645, -0.6510],
           [ 0.0502, -0.2146, -0.1427],
           [ 0.4796,  0.5612,  0.1099]],

          [[ 0.0987,  0.7185,  0.9346],
           [ 0.0759, -0.2976,  0.2438],
           [ 0.2201, -0.1353, -0.1484]]],


         [[[-0.3146,  0.0650, -0.0919],
           [ 0.2711,  0.1592,  0.1563],
           [-0.1413,  0.0043,  0.0559]],

          [[ 0.1131,  0.2231,  0.0074],
           [ 0.5374,  0.4241,  0.0101],
           [ 0.1828,  0.2742,  0.3573]]]])
conv2.0.bias
 tensor([ 0.5919,  0.0174, -0.0912])
```

Figure 6.1.1.10. Second Convolutional Layer Weights and Bias

The weight matrix of the output fully connected (dense) layer out.weight in the CNN model is declared in Figure 6.1.1.11. This layer connects the flattened feature vector from the previous layer (of size 75) to the 10 output classes (digits 0–9), and the weight tensor has the shape [10, 75]. Each row represents the weights associated with one output neuron, and each column corresponds to a feature from the flatten layer. The values are represented in scientific notation, indicating precise floating-point numbers learned during training. These weights determine how strongly each feature contributes to a specific class prediction. Along with the associated out.bias in Figure 6.1.1.12, these parameters complete the final computation stage of the CNN, producing the class logits. For FPGA deployment, the weights are quantized to fixed-point representation and stored

in memory. They are then loaded into the design using $readmemh or through UART to perform real-time digit classification.

```
out.weight
 tensor([[ 1.2590e-01,  1.7423e-01, -1.1697e-01,  7.8632e-02, -2.2715e-01,
           1.6016e-01,  1.8257e-01, -4.7618e-01, -1.5495e-01, -8.3346e-01,
          -5.9894e-02, -3.3716e-01,  1.4041e-02,  1.4780e-02, -8.8312e-01,
          -5.0943e-01, -3.9915e-01, -3.7103e-01, -2.6598e-01,  2.7425e-01,
          -3.0650e-01, -5.8047e-01,  2.5383e-01,  4.7415e-01,  7.3430e-01,
          -3.0098e-02, -1.2895e-02,  4.8265e-02,  1.2132e-01, -7.3929e-02,
           1.4242e-01,  2.0702e-01, -2.1031e-01,  1.0990e-01, -1.1270e-01,
          -5.9445e-01, -7.1520e-01,  9.6179e-02, -3.0214e-01, -8.0249e-01,
          -1.0584e+00, -5.1217e-01,  4.0718e-01,  1.3079e-02, -8.0991e-01,
          -1.7587e-01,  6.8624e-02,  6.1310e-01, -8.5488e-02, -2.5026e-01,
          -1.0341e-03, -3.2017e-01,  7.5627e-03,  1.1876e-01, -9.1084e-03,
           6.0919e-02,  2.2486e-01,  2.3304e-01,  1.7631e-01,  2.0531e-01,
           1.3548e-01, -5.4571e-02, -5.8380e-01, -2.8579e-01,  3.9145e-02,
           1.6843e-01, -1.3098e-01, -5.3251e-01, -3.3119e-01,  2.1391e-01,
           2.0365e-02,  2.5871e-01,  3.0231e-01,  2.2347e-01,  8.2336e-02],
```

Figure 6.1.1.11. Output Layer Weights

```
out.bias
 tensor([ 0.2748,  0.6083, -0.1970, -0.6693,  0.5291, -0.2911,  0.0876, -0.0701,
          0.1496, -0.2314])
```

Figure 6.1.1.12. Output Layer Biases

6.1.2. VGG-16 model

Figure 6.1.2.1 shows the initial step of the Python-based deep learning pipeline, which involves importing all necessary libraries. General-purpose libraries such as numpy, matlib, os and random are used for numerical operations, visualization, file management, and reproducibility, respectively. The ‘tqdm’ library is included to provide progress bars for iterative operations. TensorFlow and its high-level API Keras are imported for building and training the CNN model, including tools for preprocessing, modeling, and visualization. In addition, the script imports VGG-16 and preprocess-input from Keras applications, indicating that transfer learning using the VGG-16 architecture is planned later in the workflow. The successful import of these libraries is

confirmed by the printed message “Importing libraries completed.”, ensuring the environment is ready for model development and training.

In [1]:
```
# importing libraries
import tensorflow
import numpy as np
import matplotlib.pyplot as plt
import os
import random
from tqdm import tqdm # for progress bar

# Libraries for TensorFlow
from tensorflow import keras
from tensorflow.keras.utils import to_categorical, plot_model
from tensorflow.keras.preprocessing import image
from tensorflow.keras import models, layers


# Library for Transfer Learning
from tensorflow.keras.applications import VGG16
from keras.applications.vgg16 import preprocess_input

print("Importing libraries completed.")
```

```
Importing libraries completed.
```

Figure 6.1.2.1. Libraries Importation

The dimensional structure of the MNIST dataset after the data acquisition step is shown in Figure 6.1.2.2. The training set consists of 60,000 grayscale images, each with a resolution of 28×28 pixels, resulting in a data shape of (60,000, 28, 28). The associated training labels are stored in a one-dimensional array of shape (60,000,), representing the class indices from 0 to 9. Similarly, the test set includes 10,000 images with the same spatial dimensions and a corresponding array of 10,000 labels. This format aligns with standard practices in image classification research, allowing for efficient preprocessing, model training, and evaluation. The model is configured by reshaping and resizing the dataset to compact with the digit recognition task, every step is demonstrated from Figure 6.1.2.3 to Figure 6.1.2.7.

```
Downloading data from https://storage.googleapis.com/tensorflow/tf-keras-datasets/mnist.npz
11490434/11490434 ━━━━━━━━━━━━━━━━━━━━ 0s 0us/step
```

```
(60000, 28, 28)
(60000,)
(10000, 28, 28)
(10000,)
```

Figure 6.1.2.2. Data Gathering

```
In [5]:
x_train = x_train.reshape(-1, 28,28,3)
x_test= x_test.reshape (-1,28,28,3)
x_train.shape,x_test.shape

Out[5]: ((60000, 28, 28, 3), (10000, 28, 28, 3))
```

Figure 6.1.2.3. Image Reshaping

```
In [6]:
from keras.preprocessing.image import img_to_array, array_to_img

x_train = np.asarray([img_to_array(array_to_img(im, scale=False).resize((48,48))) for im in x_train])
x_test = np.asarray([img_to_array(array_to_img(im, scale=False).resize((48,48))) for im in x_test])
x_train.shape, x_test.shape

Out[6]: ((60000, 48, 48, 3), (10000, 48, 48, 3))
```

Figure 6.1.2.4. Image Resizing

```
In [8]:
x=[] # to store array value of the images
x=x_train
y=[] # to store the labels of the images
y=y_train

test_images=[]
test_images=x_test
test_images_Original=[]
test_images_Original=x_test
test_image_label=[] # to store the labels of the images
test_image_label=y_test

val_images=[]
val_images=x_test
val_images_Original=[]
val_images_Original=x_test
val_image_label=[] # to store the labels of the images
val_image_label=y_test # to store the labels of the images

print("Preparing Dataset Completed.")

Preparing Dataset Completed.
```

Figure 6.1.2.5. Data Preparing

```
Training Dataset
(60000, 48, 48, 3)
(60000, 10)
Test Dataset
(10000, 48, 48, 3)
(10000, 10)
Validation Dataset
(10000, 48, 48, 3)
(10000, 10)
```

Figure 6.1.2.6. Data Verifying

```
Summary of default VGG16 model.

Downloading data from https://storage.googleapis.com/tensorflow/keras-applications/vgg16/vgg16_weights_tf_dim_ordering_tf_k
ernels.h5
553467096/553467096 ━━━━━━━━━━━━━━━━━━━━ 2s 0us/step
```

Figure 6.1.2.7. Model Loading

The architecture of the custom VGG-16 model is presented in Figure 6.1.2.8, highlighting its layer-wise configuration, output dimensions, and parameter counts. VGG-16 is a deep convolutional neural network composed of 13 convolutional layers, 5 max-pooling layers, and 3 fully connected (dense) layers. The model accepts input images of shape (224, 224, 3) and progressively reduces spatial dimensions through pooling operations while increasing the number of feature maps, reaching 512 channels by the fifth convolutional block. The final feature maps are flattened into a 25,088-dimensional vector before being passed to two dense layers of size 4,096 and a final classification layer of 1,000 outputs, suitable for ImageNet classification. The total number of trainable parameters is approximately 138 million, making VGG-16 a computationally intensive model, particularly in the fully connected layers. Its simplicity in design and high performance have made it a popular choice in transfer learning and computer vision applications.

Model: "vgg16"

Layer (type)	Output Shape	Param #
input_layer (InputLayer)	(None, 224, 224, 3)	0
block1_conv1 (Conv2D)	(None, 224, 224, 64)	1,792
block1_conv2 (Conv2D)	(None, 224, 224, 64)	36,928
block1_pool (MaxPooling2D)	(None, 112, 112, 64)	0
block2_conv1 (Conv2D)	(None, 112, 112, 128)	73,856
block2_conv2 (Conv2D)	(None, 112, 112, 128)	147,584
block2_pool (MaxPooling2D)	(None, 56, 56, 128)	0
block3_conv1 (Conv2D)	(None, 56, 56, 256)	295,168
block3_conv2 (Conv2D)	(None, 56, 56, 256)	590,080
block3_conv3 (Conv2D)	(None, 56, 56, 256)	590,080
block3_pool (MaxPooling2D)	(None, 28, 28, 256)	0
block4_conv1 (Conv2D)	(None, 28, 28, 512)	1,180,160
block4_conv2 (Conv2D)	(None, 28, 28, 512)	2,359,808
block4_conv3 (Conv2D)	(None, 28, 28, 512)	2,359,808
block4_pool (MaxPooling2D)	(None, 14, 14, 512)	0
block5_conv1 (Conv2D)	(None, 14, 14, 512)	2,359,808
block5_conv2 (Conv2D)	(None, 14, 14, 512)	2,359,808
block5_conv3 (Conv2D)	(None, 14, 14, 512)	2,359,808
block5_pool (MaxPooling2D)	(None, 7, 7, 512)	0
flatten (Flatten)	(None, 25088)	0
fc1 (Dense)	(None, 4096)	102,764,544
fc2 (Dense)	(None, 4096)	16,781,312
predictions (Dense)	(None, 1000)	4,097,000

Total params: 138,357,544 (527.79 MB)
Trainable params: 138,357,544 (527.79 MB)
Non-trainable params: 0 (0.00 B)

Figure 6.1.2.8. VGG-16 Model

The initial customization of the VGG16 model is illustrated in Figure 6.1.2.9 and Figure 6.1.2.10. In this step, the model is modified to better suit the target task by first defining a custom input layer and then removing the original top (fully connected) layers. This approach is commonly used in transfer learning, where the convolutional base of a pre-trained model is retained to leverage its feature extraction capabilities, while the classification head is replaced with a new

structure tailored to a specific dataset. The output also shows the downloading process of the pre-trained weights from the TensorFlow repository, confirming that the model is being loaded without the top layers. This allows the user to build and train a new classifier on top of the rich feature representations learned by VGG16 from large-scale datasets like ImageNet.

```
Summary of Custom VGG16 model.

1) We setup input layer.
2) We removed top (last) layer.

Downloading data from https://storage.googleapis.com/tensorflow/keras-applications/vgg16/vgg16_weights_tf_dim_ordering_tf_ker
nels_notop.h5
58889256/58889256 ━━━━━━━━━━━━━━━━━━━━ 0s 0us/step
```

Figure 6.1.2.9. Model First Customization

```
Summary of Custom VGG16 model.

1) We flatten the last layer and added 1 Dense layer and 1 output layer.
```

Figure 6.1.2.10. Model Second Customization

Figure 6.1.2.11 shows the architecture of the Custom VGG16 model, designed for image classification with input dimensions of 48×48×3. The model follows a simplified VGG16 structure, consisting of five convolutional blocks with 3×3 filters and increasing depth from 64 to 512 filters, each followed by max pooling layers that reduce spatial dimensions. After the final pooling layer, the output is flattened and passed to a dense layer with 10 output units for classification. The model contains approximately 14.7 million parameters, of which only 5,130 are trainable, indicating the use of a pre-trained convolutional base with only the final layer fine-tuned. This approach leverages transfer learning for improved efficiency and performance on smaller datasets.

Model: "CustomVGG16"

Layer (type)	Output Shape	Param #
input_layer_1 (InputLayer)	(None, 48, 48, 3)	0
block1_conv1 (Conv2D)	(None, 48, 48, 64)	1,792
block1_conv2 (Conv2D)	(None, 48, 48, 64)	36,928
block1_pool (MaxPooling2D)	(None, 24, 24, 64)	0
block2_conv1 (Conv2D)	(None, 24, 24, 128)	73,856
block2_conv2 (Conv2D)	(None, 24, 24, 128)	147,584
block2_pool (MaxPooling2D)	(None, 12, 12, 128)	0
block3_conv1 (Conv2D)	(None, 12, 12, 256)	295,168
block3_conv2 (Conv2D)	(None, 12, 12, 256)	590,080
block3_conv3 (Conv2D)	(None, 12, 12, 256)	590,080
block3_pool (MaxPooling2D)	(None, 6, 6, 256)	0
block4_conv1 (Conv2D)	(None, 6, 6, 512)	1,180,160
block4_conv2 (Conv2D)	(None, 6, 6, 512)	2,359,808
block4_conv3 (Conv2D)	(None, 6, 6, 512)	2,359,808
block4_pool (MaxPooling2D)	(None, 3, 3, 512)	0
block5_conv1 (Conv2D)	(None, 3, 3, 512)	2,359,808
block5_conv2 (Conv2D)	(None, 3, 3, 512)	2,359,808
block5_conv3 (Conv2D)	(None, 3, 3, 512)	2,359,808
block5_pool (MaxPooling2D)	(None, 1, 1, 512)	0
flatten (Flatten)	(None, 512)	0
dense_3 (Dense)	(None, 10)	5,130

Total params: 14,719,818 (56.15 MB)
Trainable params: 5,130 (20.04 KB)
Non-trainable params: 14,714,688 (56.13 MB)

Figure 6.1.2.11. Custom VGG-16 Model

Figures 6.1.2.12 and 6.1.2.13 indicate the training progress and result of the Custom VGG16 model, respectively. Figure 6.1.2.12 displays training and validation accuracy and loss values recorded across 20 epochs. The model exhibits rapid convergence, achieving over 90% training accuracy by the third epoch and consistently maintaining above 91% validation accuracy in subsequent epochs. The validation loss also decreases steadily, indicating effective learning without significant overfitting. By the final epoch, the model achieves a training accuracy of 93.19% and a validation accuracy of 91.97%, with corresponding losses of 0.2150 and 0.2851,

respectively. These results demonstrate that the model is well-generalized on unseen data. Figure 6.1.2.13 shows the final stage of training, where the fitting process reaches completion with all steps executed successfully. The progress bars indicate efficient batch processing, with the final iteration completing at a speed of approximately 8.09 iterations per second. Together, these figures confirm the stability, effectiveness, and performance efficiency of the Custom VGG16 model during the training phase.

```
Epoch 1/20
375/375 ━━━━━━━━━━━━━━━━━━━━ 28s 59ms/step - accuracy: 0.5321 - loss: 5.7297 - val_accuracy: 0.8612 - val_loss: 0.5961
Epoch 2/20
375/375 ━━━━━━━━━━━━━━━━━━━━ 33s 50ms/step - accuracy: 0.8716 - loss: 0.5191 - val_accuracy: 0.8996 - val_loss: 0.3744
Epoch 3/20
375/375 ━━━━━━━━━━━━━━━━━━━━ 22s 54ms/step - accuracy: 0.9006 - loss: 0.3510 - val_accuracy: 0.9068 - val_loss: 0.3394
Epoch 4/20
375/375 ━━━━━━━━━━━━━━━━━━━━ 21s 54ms/step - accuracy: 0.9148 - loss: 0.2843 - val_accuracy: 0.9140 - val_loss: 0.3007
Epoch 5/20
375/375 ━━━━━━━━━━━━━━━━━━━━ 20s 54ms/step - accuracy: 0.9195 - loss: 0.2685 - val_accuracy: 0.9199 - val_loss: 0.2787
Epoch 6/20
375/375 ━━━━━━━━━━━━━━━━━━━━ 19s 51ms/step - accuracy: 0.9234 - loss: 0.2530 - val_accuracy: 0.9231 - val_loss: 0.2657
Epoch 7/20
375/375 ━━━━━━━━━━━━━━━━━━━━ 22s 55ms/step - accuracy: 0.9239 - loss: 0.2435 - val_accuracy: 0.9143 - val_loss: 0.2853
Epoch 8/20
375/375 ━━━━━━━━━━━━━━━━━━━━ 21s 55ms/step - accuracy: 0.9265 - loss: 0.2311 - val_accuracy: 0.9200 - val_loss: 0.2753
Epoch 9/20
375/375 ━━━━━━━━━━━━━━━━━━━━ 40s 51ms/step - accuracy: 0.9297 - loss: 0.2254 - val_accuracy: 0.9148 - val_loss: 0.2845
Epoch 10/20
375/375 ━━━━━━━━━━━━━━━━━━━━ 19s 51ms/step - accuracy: 0.9303 - loss: 0.2270 - val_accuracy: 0.9181 - val_loss: 0.2744
Epoch 11/20
375/375 ━━━━━━━━━━━━━━━━━━━━ 21s 51ms/step - accuracy: 0.9285 - loss: 0.2304 - val_accuracy: 0.9222 - val_loss: 0.2645
Epoch 12/20
375/375 ━━━━━━━━━━━━━━━━━━━━ 19s 52ms/step - accuracy: 0.9294 - loss: 0.2207 - val_accuracy: 0.9227 - val_loss: 0.2616
Epoch 13/20
375/375 ━━━━━━━━━━━━━━━━━━━━ 22s 55ms/step - accuracy: 0.9283 - loss: 0.2243 - val_accuracy: 0.9211 - val_loss: 0.2734
Epoch 14/20
375/375 ━━━━━━━━━━━━━━━━━━━━ 41s 55ms/step - accuracy: 0.9292 - loss: 0.2267 - val_accuracy: 0.9223 - val_loss: 0.2729
Epoch 15/20
375/375 ━━━━━━━━━━━━━━━━━━━━ 39s 51ms/step - accuracy: 0.9318 - loss: 0.2186 - val_accuracy: 0.9199 - val_loss: 0.2714
Epoch 16/20
375/375 ━━━━━━━━━━━━━━━━━━━━ 19s 51ms/step - accuracy: 0.9306 - loss: 0.2205 - val_accuracy: 0.9233 - val_loss: 0.2717
Epoch 17/20
375/375 ━━━━━━━━━━━━━━━━━━━━ 22s 55ms/step - accuracy: 0.9326 - loss: 0.2130 - val_accuracy: 0.9230 - val_loss: 0.2682
Epoch 18/20
375/375 ━━━━━━━━━━━━━━━━━━━━ 41s 55ms/step - accuracy: 0.9318 - loss: 0.2182 - val_accuracy: 0.9220 - val_loss: 0.2726
Epoch 19/20
375/375 ━━━━━━━━━━━━━━━━━━━━ 39s 51ms/step - accuracy: 0.9307 - loss: 0.2222 - val_accuracy: 0.9233 - val_loss: 0.2758
Epoch 20/20
375/375 ━━━━━━━━━━━━━━━━━━━━ 19s 51ms/step - accuracy: 0.9319 - loss: 0.2150 - val_accuracy: 0.9197 - val_loss: 0.2851
Fitting the model completed.
```

Figure 6.1.2.12. Custom VGG-16 Model Fitting

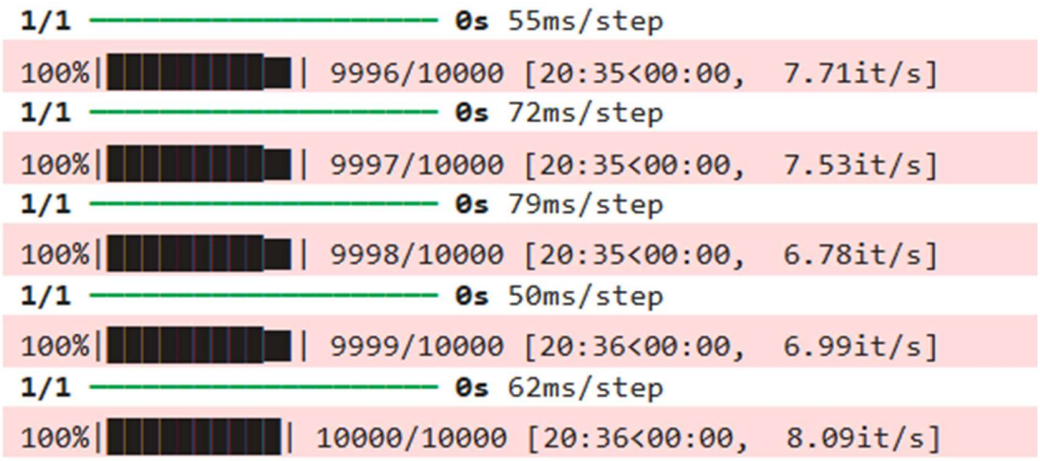


Figure 6.1.2.13. Training Result

Figure 6.1.2.14 illustrates example classification results using the VGG16 model in a transfer learning context. Each image represents a handwritten digit along with its predicted label, confidence score (as a percentage), and the true label shown in parentheses. The green vertical bar indicates correct classifications, while red bars denote misclassifications. Most predictions are highly confident, with many achieving 100% certainty. The overall accuracy appears visually consistent with strong performance across various digits, such as “Seven 100% (Seven)” and “Zero 100% (Zero)”. A few cases, such as “Nine 53% (Nine)” or “Two 95% (Two)”, show reduced confidence, which is typical for ambiguous or poorly written digits. These results highlight the effectiveness of transfer learning using VGG16 in digit recognition tasks, demonstrating that even a model pre-trained on natural images can successfully generalize to digit classification with minimal training modifications.

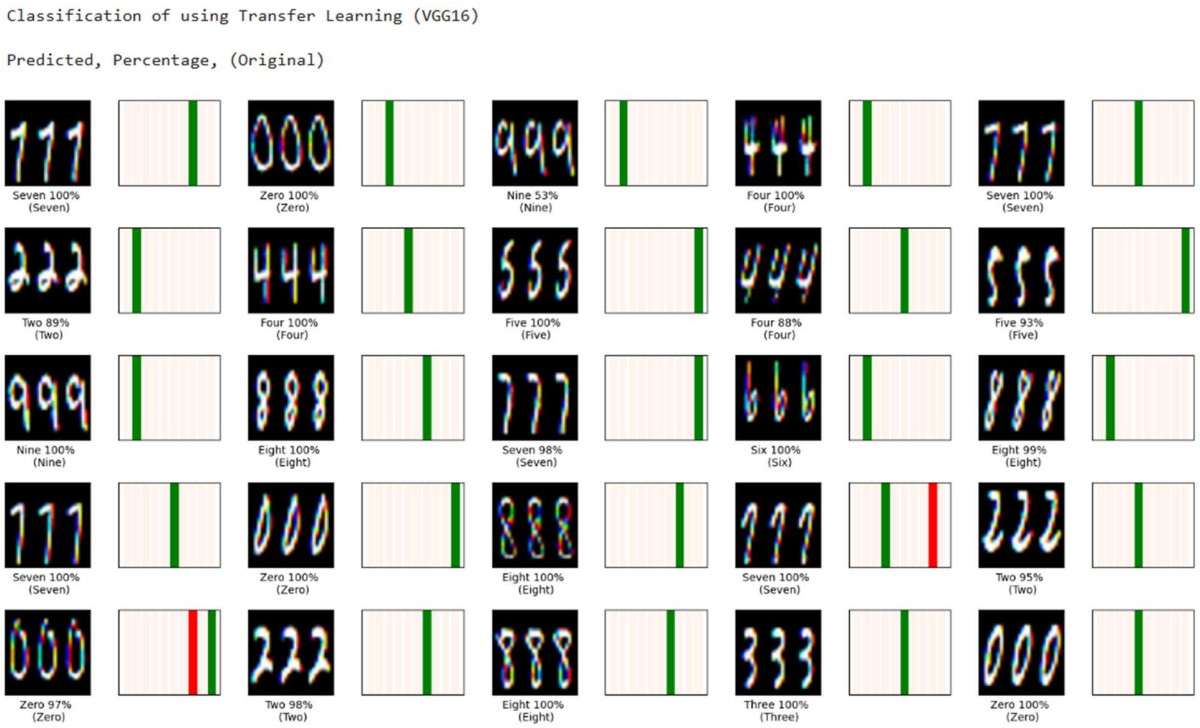


Figure 6.1.2.14. Example Classification

The model's training and validation performance are illustrated in Figures 6.1.2.15 and 6.1.2.16. The accuracy graph in Figure 6.1.2.15 shows a steady increase in both training and validation accuracy across 20 epochs, with both curves stabilizing above 90%. This indicates that the model effectively learns to classify the input data while generalizing well to unseen samples. The minimal gap between the two curves suggests that overfitting is largely avoided. Figure 6.1.2.16 depicts the corresponding loss curves, which show a rapid decline during the initial epoch and gradual convergence thereafter. The training loss continues to decrease slightly, while the validation loss stabilizes early on. Together, these figures demonstrate that the model has achieved strong convergence with high generalization capability, validating the effectiveness of the chosen architecture and training configuration.

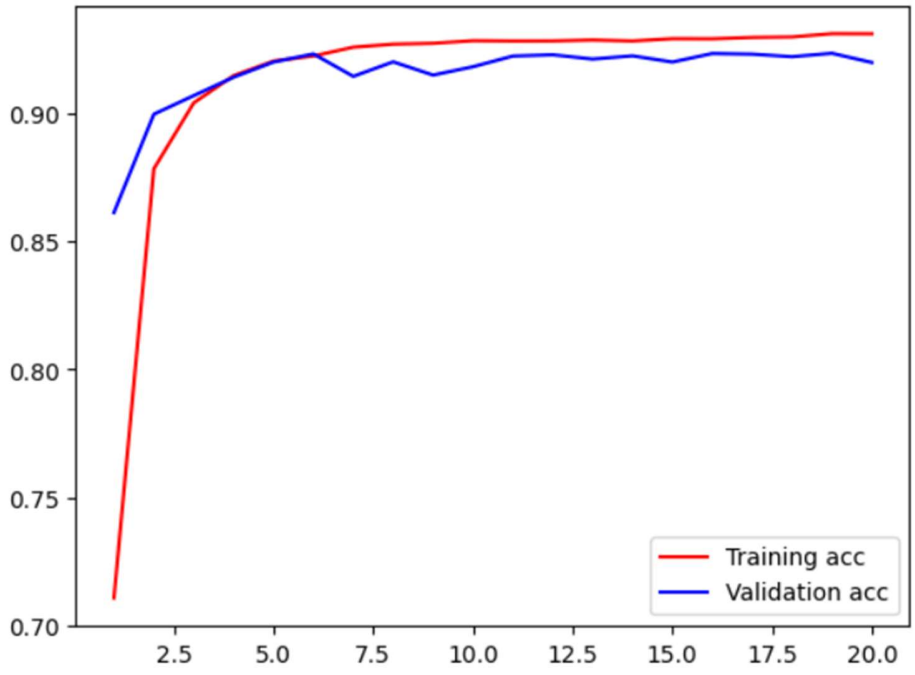


Figure 6.1.2.15. Training and Validation Accuracy

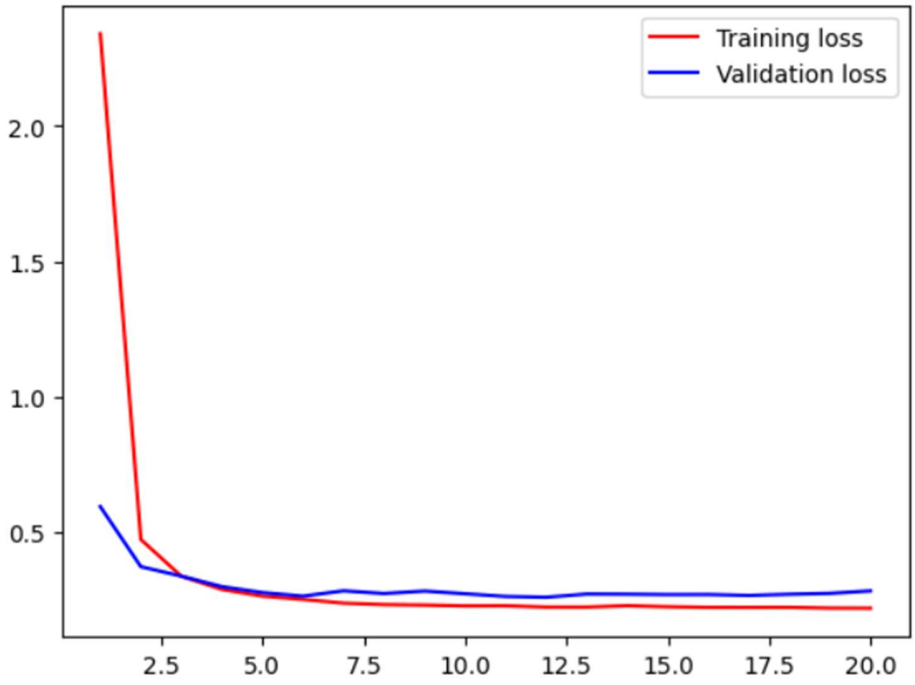


Figure 6.1.2.16. Training and Validation Loss

6.2. Image Conversion

Figure 6.2.1 confirms the successful execution of the image conversion script, which transforms an MNIST digit image into a .dat file suitable for FPGA input. The terminal output shows that the Python script image_signed_convert.py was run without errors, and the converted data was saved to the specified directory. The output file contains the processed pixel values of the image are shown in Figure 6.2.2., encoded in a format compatible with the Verilog-based image RAM on the FPGA. This conversion step is essential for bridging the software-trained model with the hardware implementation, ensuring that the input fed into the FPGA matches the format expected by the CNN pipeline during simulation or real-time inference.

Figure 6.2.1. Success converting image to .dat

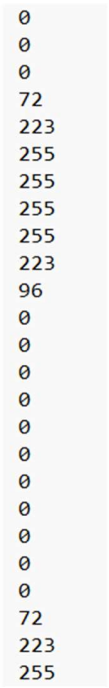


Figure 6.2.2. File .dat content

6.3. Intel HEX File Generation

Figure 6.3.1 displays the successful execution of the hex_generator.py script, which converts trained model parameters from .dat format into .hex files compatible with Verilog-based memory initialization. The terminal output confirms that separate HEX files were created for each layer's weights and biases, including conv1_weight.hex, conv1_bias.hex, conv2_weight.hex, conv2_bias.hex, dense_weight.hex, and dense_bias.hex. The number of values written into each file is also shown, matching the expected size of each layer based on the trained CNN model. These HEX files are later loaded into FPGA RAM using $readmemh, allowing the hardware to perform inference using the exact weights obtained during training. This step is essential for ensuring consistency between the software and hardware implementations of the neural network.

```
> python -u "c:\Users\Acer\Downloads\CNN-FPGA-Implementation-main\miscellaneous\hex_generator\weights_biases_hex_generator.py"
✅ HEX file written: C:/Users/Acer/Downloads/CNN-FPGA-Implementation-main/src/IntelHEX/conv1_weight.hex (50 values)
✅ HEX file written: C:/Users/Acer/Downloads/CNN-FPGA-Implementation-main/src/IntelHEX/conv1_bias.hex (2 values)
✅ HEX file written: C:/Users/Acer/Downloads/CNN-FPGA-Implementation-main/src/IntelHEX/conv2_weight.hex (54 values)
✅ HEX file written: C:/Users/Acer/Downloads/CNN-FPGA-Implementation-main/src/IntelHEX/conv2_bias.hex (3 values)
✅ HEX file written: C:/Users/Acer/Downloads/CNN-FPGA-Implementation-main/src/IntelHEX/dense_weight.hex (750 values)
✅ HEX file written: C:/Users/Acer/Downloads/CNN-FPGA-Implementation-main/src/IntelHEX/dense_bias.hex (10 values)
PS C:\Users\Acer>
```

Figure 6.3.1. Success converting .dat file to .hex file

```
FEA5
FF75
FFAC
FFC3
FF4C
0007
FF58
FF1B
FF5A
FFF3
0009
0087
0075
0025
000D
007B
```

Figure 6.3.2. File .hex of the first convolution weight after conversion

6.4. CNN Model Layers

6.4.1. First Convolutional Layer and ReLU

Figure 6.4.1.1 illustrates the simulation waveform of the convolutional layer and ReLU activation function module during the hardware-level implementation of the CNN model. This output was generated using ModelSim. The figure displays various control and data signals, including the clock (clk), reset (reset_n), start (start), image and weight memory buses, as well as the computed feature map output. The simulation confirms that image, weight and bias data are correctly read from memory and processed to the array. The finish signal transitions high once the computation completes, indicating successful execution of the convolution operation. The bottom section shows the binary content of the resulting feature map, verifying that the convolutional layer performs as expected.

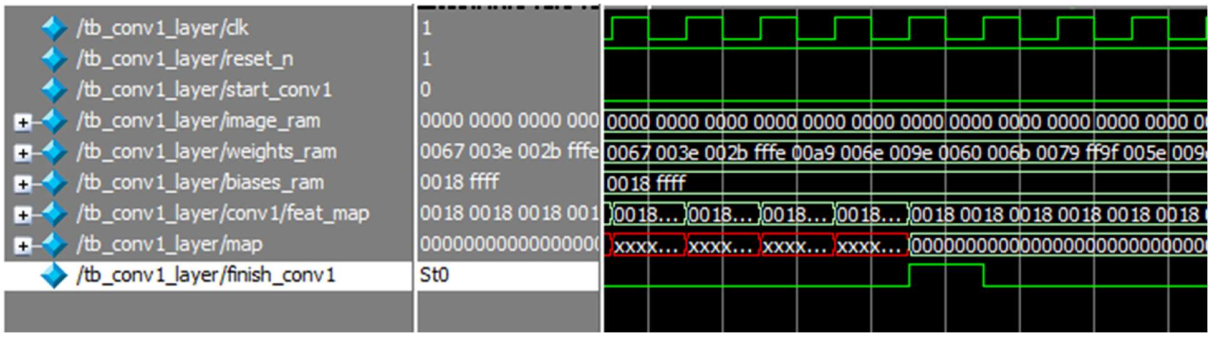


Figure 6.4.1.1. First Convolutional Layer Waveform

The memory initialization data for the convolutional layer simulation is shown in the three figures below. Figure 6.4.1.2 displays the contents of the image_ram, where pixel values from the input image have been stored in hexadecimal format. These values represent the fixed-point encoded grayscale intensity levels used for convolution. The structure confirms proper image loading, with non-zero values corresponding to the image pixel regions of the digit. Figure 6.4.1.3 shows the weights_ram, containing preloaded kernel weights for the convolutional filters. Each

16-bit value represents a fixed-point weight from the trained CNN model, formatted using Q0.8 notation. The organized layout confirms correct memory addressing for multi-channel convolution operations. Figure 6.4.1.4 presents the biases_ram, where the corresponding bias values for each output filter are stored. The presence of valid bias data, shown as 0018 and ffff, confirms that the hardware is correctly incorporating bias during convolution. Together, these memory snapshots confirm the successful loading and formatting of input image data, filter weights, and biases, validating the initialization stage before convolution execution in FPGA simulation.

```
sim:/tb_conv1_layer/image_ram @ 11555 ns
  0 : 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000
 16 : 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000
 32 : 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000
 48 : 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000
 64 : 0000 0000 0000 0000 0000 0000 4800 df00 ff00 ff00 ff00 ff00 df00 6000 0000 0000
 80 : 0000 0000 0000 0000 0000 0000 0000 0000 0000 4800 df00 ff00 ff00 ff00 ff00 ff00
 96 : ff00 ff00 ff00 ff00 ff00 ff00 ff00 ff00 ff00 f700 0000 0000 0000 0000 0000 0000
112 : 0000 0000 0000 0000 0000 df00 ff00 ff00 ff00 ff00 ff00 ff00 ff00 ff00 ff00 ff00
128 : ff00 ff00 ff00 ff00 ff00 ff00 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000
144 : 0000 7800 ff00 ff00 ff00 ff00 ff00 ff00 ff00 ff00 ff00 ff00 9700 0000 e700 ff00
160 : ff00 ff00 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000
176 : 0000 0000 0000 0000 0000 0000 0000 0000 0000 4800 ff00 ff00 ff00 9700 0000 0000
192 : 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000
208 : 0000 0000 0000 0000 4800 ef00 ff00 ff00 ff00 0000 0000 0000 0000 0000 0000 0000
224 : 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 4800
240 : ef00 ff00 ff00 ff00 9700 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000
256 : 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 4800 ef00 ff00 ff00 ff00 9700
272 : 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000
288 : 0000 0000 0000 0000 0000 0000 e700 ff00 ff00 ff00 9700 0000 0000 0000 0000 0000
304 : 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000
320 : 0000 4800 ff00 ff00 ff00 9700 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000
336 : 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 4800 ef00 ff00 ff00
352 : ff00 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000
368 : 0000 0000 0000 0000 0000 0000 0000 0000 e700 ff00 ff00 ff00 9700 0000 0000 0000
384 : 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000
400 : 0000 0000 0000 4800 ff00 ff00 ff00 9700 0000 0000 0000 0000 0000 0000 0000 0000
416 : 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 4800 ef00
432 : ff00 ff00 ff00 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000
448 : 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 e700 ff00 ff00 ff00 9700 0000
464 : 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000
480 : 0000 0000 0000 0000 0000 4800 ff00 ff00 ff00 9700 0000 0000 0000 0000 0000 0000
496 : 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000
512 : 0000 e700 ff00 ff00 ff00 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000
528 : 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 ff00 ff00 ff00
544 : 9700 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000
560 : 0000 0000 0000 0000 0000 0000 0000 0000 0000 ff00 ff00 ff00 0000 0000 0000 0000
576 : ...
  .
  .
  .
```

Figure 6.4.1.2. Image RAM of First Convolution Layer

```
sim:/tb_conv1_layer/weights_ram @ 11383 ns
 0 : 0067 003e 002b fffe
 4 : 00a9 006e 009e 0060
 8 : 006b 0079 ff9f 005e
12 : 009c 0059 0028 fea5
16 : ff75 ffac ffc3 ff4c
20 : 0007 ff58 ff1b ff5a
24 : fff3 0009 0087 0075
28 : 0025 000d 007b 00a4
32 : 0062 0031 ffcb 0077
36 : 0052 001c 003c fff1
40 : 002a 003a ffbf fffa
44 : 00e6 ffdc 005d ff98
48 : ff57 00bc
```

Figure 6.4.1.3. Weights RAM of First Convolution Layer

```
sim:/tb_conv1_layer/biases_ram @ 11341 ns
0018 ffff
```

Figure 6.4.1.4. Biases RAM of First Convolution Layer

Figure 6.4.1.5 illustrates the contents of the feature map RAM corresponding to the first convolutional layer in the simulation environment. Captured at 11,560 ns, this output represents the raw activation data produced by the convolution operation applied to the input image. Each row shows a memory address offset followed by a sequence of hexadecimal values, which correspond to two channels, each has 24x24 size value. These values reflect the result of applying filters to local regions of the input data, highlighting areas of high activation which typically correspond to features such as edges or textures. The repeating patterns like 0018, and higher-value activations such as fc18, af18, and cc1c, suggest both different grayscale color level activations across different spatial regions. There are no negative value because of the ReLU activation function. The simulation output verifies that the convolution logic is functioning correctly and storing results as expected in the dedicated RAM module.

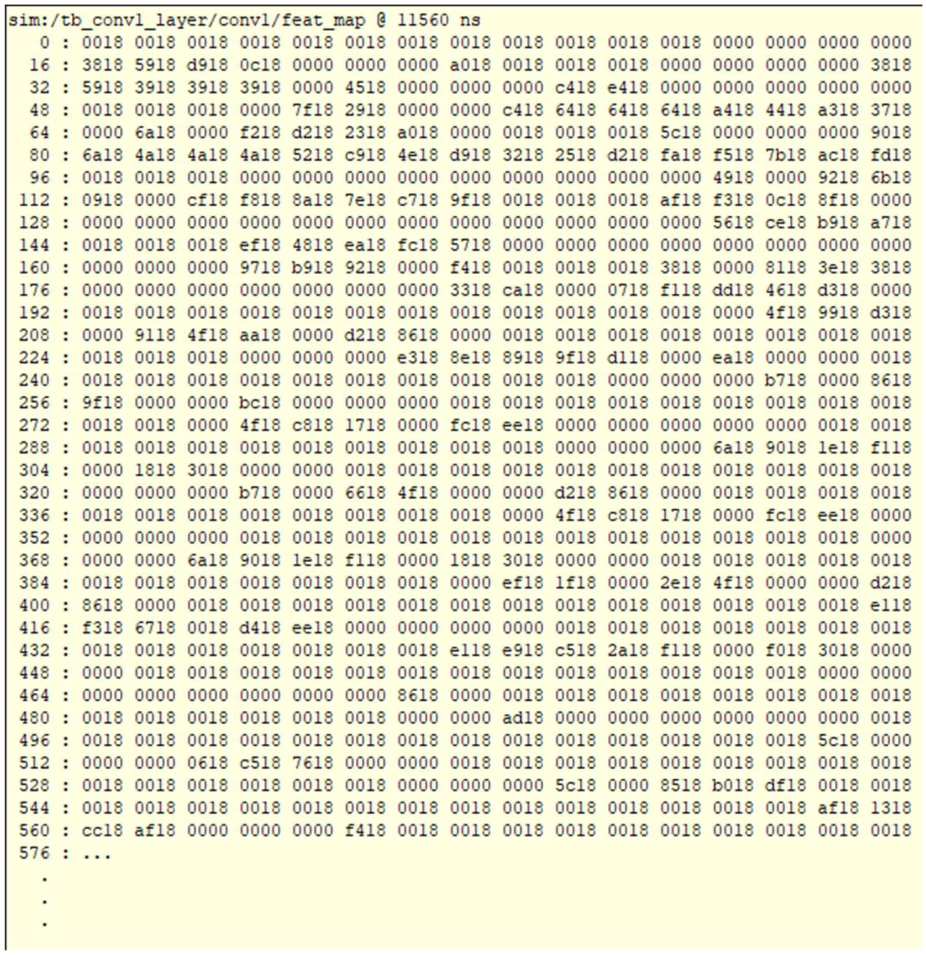


Figure 6.4.1.5. Feature Map RAM of First Convolution Layer

6.4.2. First Max Pooling Layer

Figure 6.4.2 displays the simulation waveform of the first MaxPool2x2ReLU module, highlighting the functional behavior of the max-pooling operation with integrated ReLU activation. The waveform shows key signals such as the clock (clk), reset (reset_n), start signal (start_max1), input data (data_in), intermediate signals (input_data and pooled_data from the MaxReLU submodule), and final output (data_out). Initially, the start_max1 signal is asserted low, and upon activation, the input_data receives values such as 3524 and 1215. The max pooling operation proceeds, extracting the maximum values from local 2x2 regions, and choosing the 3524

to be the output. Finally, the finish_max1 signal is asserted to indicate completion. This waveform confirms the correct timing, control, and data flow through the max-pooling and ReLU logic in the hardware implementation.

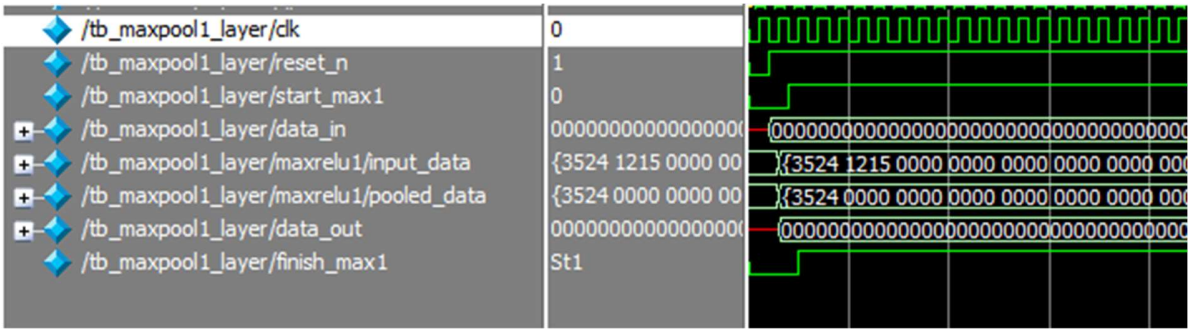


Figure 6.4.2. First MaxPool2x2ReLU Waveform

6.4.3. First Convolutional Block

Figure 6.4.3.1 illustrates the simulation waveform of the first convolutional block, which includes both convolution and activation (e.g., ReLU) stages. Key signals such as the system clock (clk), reset (reset_n), and start control signal are shown at the top, ensuring synchronized operation. Below these, memory buses for image input (image_in_linear), weights, and biases are displayed, confirming that data is correctly fetched for computation. As the convolution proceeds, the output data (block_out_linear) is expanded into linear format from two channels of pooled_data indicating in Figure 6.4.3.2 and Figure 6.4.3.3. These outputs represent the feature maps produced after the convolutional and activation operations. The assertion of the finish signal confirms that the block has successfully completed its operation. This simulation result validates the correct functional behavior of the first convolutional block in the hardware implementation of the CNN model.

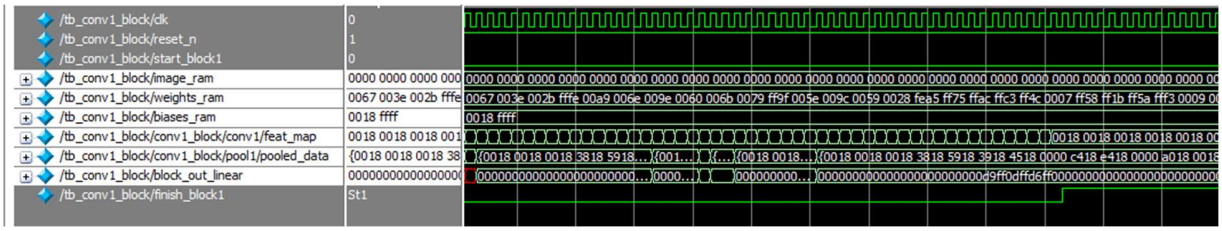


Figure 6.4.3.1. First Convolutional Block Waveform

```
sim:/tb_conv1_block/conv1_block/pool1/pooled_data[0] @ 11587 ns
  0 : 0018 0018 0018 3818 5918 3918 4518 0000
  8 : c418 e418 0000 a018 0018 5c18 7f18 9018
 16 : c418 6418 c918 d918 6a18 fa18 f518 fd18
 24 : 0018 af18 f318 8f18 0000 0000 4918 9218
 32 : 0918 f818 ce18 c718 0018 ef18 ea18 fc18
 40 : 0000 0000 0000 3318 ca18 f118 dd18 f418
 48 : 0018 0018 0018 0018 0018 0018 4f18 e318
 56 : 9f18 d118 ea18 8618 0018 0018 0018 0018
 64 : 0018 4f18 c818 fc18 ee18 bc18 0000 0018
 72 : 0018 0018 0018 0018 0018 b718 9018 f118
 80 : d218 8618 0018 0018 0018 0018 0018 0018
 88 : 4f18 c818 fc18 ee18 3018 0018 0018 0018
 96 : 0018 0018 0018 e118 f318 d418 ee18 d218
104 : 8618 0018 0018 0018 0018 0018 0018 e118
112 : e918 f118 f018 8618 0018 0018 0018 0018
120 : 0018 0018 0018 5c18 ad18 c518 7618 0018
128 : 0018 0018 0018 0018 0018 0018 0018 af18
136 : cc18 8518 f418 0018 0018 0018 0018 0018
```

Figure 6.4.3.2. First Filter of First Convolutional Block RAM

```
sim:/tb_conv1_block/conv1_block/pool1/pooled_data[1] @ 11618 ns
  0 : 0000 dfff 0000 7cff 9bff 1bff dfff ecff
  8 : d2ff 77ff dcff 83ff 0000 73ff 0000 52ff
 16 : 0000 0000 e7ff ecff b5ff e6ff b8ff fbff
 24 : 0000 d6ff cdff faff 0000 0000 28ff 54ff
 32 : 0000 55ff f8ff 2fff 0000 17ff acff a0ff
 40 : 0000 0000 a8ff f9ff 0000 9bff 66ff 0000
 48 : 0000 0000 0000 0000 0000 dfff 53ff e8ff
 56 : e1ff feff 0000 0000 0000 0000 0000 0000
 64 : 0000 dfff fcff deff cfff 0000 0000 0000
 72 : 0000 0000 0000 0000 dfff fcff aeff fcff
 80 : 0000 0000 0000 0000 0000 0000 0000 dfff
 88 : 53ff aaff fcff 37ff 0000 0000 0000 0000
 96 : 0000 0000 0000 53ff 77ff e6ff 37ff 0000
104 : 0000 0000 0000 0000 0000 0000 0000 dfff
112 : f7ff 80ff 0000 0000 0000 0000 0000 0000
120 : 0000 0000 0000 75ff 0000 b3ff 0000 0000
128 : 0000 0000 0000 0000 0000 0000 0000 d6ff
136 : 0dff d9ff 0000 0000 0000 0000 0000 0000
```

Figure 6.4.3.3. Second Filter of First Convolutional Block RAM

6.4.4. Second Convolutional Layer and ReLU

The waveform shown in Figure 6.4.4.1 represents the simulation output of the second convolutional layer (conv2_layer) during model training on FPGA. This visualization includes key control signals such as the clock (clk), reset (reset_n), and start trigger (start), as well as memory interfaces for image, weight, and bias data. Once the start signal is asserted, image and kernel data are sequentially fetched from their respective memory arrays (image_ram, weights_ram, biases_ram), as shown in Figure 6.4.4.2, 6.4.4.3 and 6.4.4.4. The result of this operation is reflected in the map output at the bottom of the waveform, where a dense matrix of fixed-point values indicates the computed feature map. The transition of the finish signal to high marks the completion of the convolution process. This confirms that the second convolutional layer has functionally completed its forward pass, successfully consuming inputs and producing the corresponding output map. The result is essential for validating the sequential execution and correctness of deeper layers in the FPGA-accelerated CNN inference pipeline.

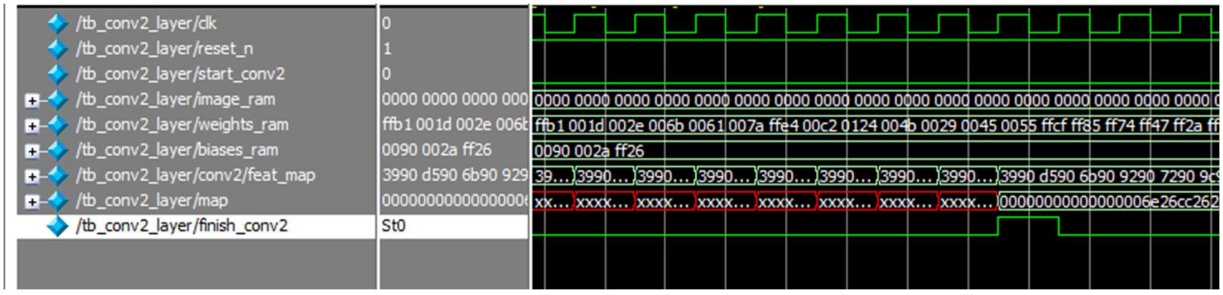


Figure 6.4.4.1. Second Convolutional Layer Waveform

```
sim:/tb_conv2_layer/image_ram @ 3060 ns
  0 : 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000
 16 : 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000
 32 : 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000
 48 : 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000
 64 : 0000 0000 0000 0000 0000 0000 4800 df00 ff00 ff00 ff00 ff00 df00 6000 0000 0000
 80 : 0000 0000 0000 0000 0000 0000 0000 0000 0000 4800 df00 ff00 ff00 ff00 ff00 ff00
 96 : ff00 ff00 ff00 ff00 ff00 ff00 ff00 ff00 ff00 f700 0000 0000 0000 0000 0000 0000
112 : 0000 0000 0000 0000 0000 df00 ff00 ff00 ff00 ff00 ff00 ff00 ff00 ff00 ff00 ff00
128 : ff00 ff00 ff00 ff00 ff00 ff00 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000
144 : 0000 7800 ff00 ff00 ff00 ff00 ff00 ff00 ff00 ff00 ff00 ff00 9700 0000 e700 ff00
160 : ff00 ff00 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000
176 : 0000 0000 0000 0000 0000 0000 0000 0000 0000 4800 ff00 ff00 ff00 9700 0000 0000
192 : 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000
208 : 0000 0000 0000 0000 4800 ef00 ff00 ff00 ff00 0000 0000 0000 0000 0000 0000 0000
224 : 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 4800
240 : ef00 ff00 ff00 ff00 9700 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000
256 : 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 4800 ef00 ff00 ff00 ff00 9700
272 : 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000
```

Figure 6.4.4.2. Image RAM of Second Convolution Layer

```
sim:/tb_conv2_layer/weights_ram @ 3010 ns
 0 : ffb1 001d 002e 006b
 4 : 0061 007a ffe4 00c2
 8 : 0124 004b 0029 0045
12 : 0055 ffcf ff85 ff74
16 : ff47 ff2a fff0 ff3f
20 : ffdd ff4a ffc8 ff78
24 : ffbb ffd3 ff6f 003a
28 : 006b 00ac 000a 0074
32 : 008b fff2 ffed 0070
36 : 003e fff5 ffdb 00b1
40 : 0022 ffff 00b0 006d
44 : 0054 fff4 ff82 ff6b
48 : 0012 0016 0015 0057
52 : 004d 005a
```

Figure 6.4.4.3. Weights RAM of Second Convolution Layer

```
sim:/tb_conv2_layer/biases_ram @ 2968 ns
0090 002a ff26
```

Figure 6.4.4.4. Biases RAM of Second Convolution Layer

Figure 6.4.4.5 presents the contents of the feature map RAM generated by the second convolutional layer at a simulation timestamp of 3070 ns. This output shows the memory structure storing the convolutional and ReLU activations in hexadecimal format, representing processed pixel values after the second convolution operation. Compared to the first layer, this feature map

exhibits a wider distribution and higher variation in values, such as f490, cf2a, and af26, indicating more complex feature extraction due to increased filter depth and receptive field. The presence of zeroes suggests regions with low activation, possibly filtered out due to low feature significance. This feature map is a crucial intermediate result, feeding into deeper layers and capturing higher-level representations of the input data. The successful capture and organization of these values into RAM verify the correct implementation of the convolution module and memory handling in the hardware simulation.

```
sim:/tb_conv2_layer/conv2/feat_map @ 3070 ns
  0 : 3990 d590 6b90 9290 7290 9c90 e390 5790 5790 5790 7190 0000 fe90 0000 0000 0000
 16 : 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 1b90 0b90 6d90 0090 0090
 32 : 0090 0090 0000 0000 9490 5790 4f90 c490 0090 0090 0090 0090 6890 4390 aa90 0000
 48 : 0000 c390 0000 0000 0000 0000 9a90 5c90 0000 0000 0000 f490 0000 0000 0000 0000
 64 : f490 8190 1a90 1e90 2990 4090 c590 ca90 5590 8590 c590 3f90 0000 0000 0000 0000
 80 : e190 0000 0000 0990 6e90 0000 0000 9e90 0000 0000 0000 0000 0000 0000 0000 0000
 96 : 4590 a790 7c90 0000 f32a a32a 1c2a 0000 0000 0000 0000 0000 0000 0000 0c2a cf2a
112 : 3a2a 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 2f2a 0000 0000 0000
128 : 0000 d92a 002a 002a 002a 002a 182a 152a d12a 0000 0000 0000 002a 002a 002a 002a
144 : 602a 6c2a 392a 2f2a 0000 0000 d12a 112a 032a 232a 0000 0000 0000 182a 652a 0000
160 : 4c2a 6c2a 762a 762a 0000 7f2a 0000 032a af2a 0000 0a2a 4e2a a82a 0000 0000 0000
176 : 0000 392a 012a 502a 0000 0000 6d2a 0000 0000 0000 fa2a 6a2a 5b2a 002a 0000 862a
192 : 0000 0000 0000 0000 0000 d02a 552a 9c2a 0000 0000 6526 0000 0000 0000 0000 0000
208 : 0000 0000 0000 0000 0000 0000 e126 f626 0c26 1e26 1e26 1e26 ad26 7826 e226 3e26
224 : 6e26 d726 dc26 0000 0000 0000 0000 0000 0000 0000 e726 1a26 e426 0000 0000 0000
240 : 0000 0000 0000 0000 0000 0000 0000 6e26 5426 5a26 0000 0000 0000 0000 c226 2f26
256 : ff26 e726 ca26 3c26 0000 0000 0000 0000 2c26 c226 7e26 0000 0000 0000 0000 0000
272 : 0000 0000 0000 0000 3526 0000 0000 0000 0000 0000 0000 0000 0000 0000 2d26 0000
288 : 0000 0000 6126 af26 4d26 2826 cc26 6e26 0000 0000 0000 0000
```

Figure 6.4.4.5. Feature Map RAM of Second Convolution Layer

6.4.5. Second Max Pooling Layer

Figure 6.4.5.1 displays the simulation waveform of the second MaxPooling layer with ReLU activation, confirming the successful execution of the MaxPool2x2 operation. The control signals include the clock (clk), reset (reset_n), and start trigger (start). Initially, the data_in signal contains undefined values (x), which is expected before valid input is received. Upon activation of the pooling operation, the data_out signal transitions from undefined to valid fixed-point values,

representing the downsampled feature map after the maximum operation is applied to 2×2 regions of the input. The assertion of the finish signal indicates the end of the pooling process. This simulation demonstrates that the MaxPooling layer not only performs spatial downsampling but also maintains the non-linearity introduced by ReLU, ensuring proper feature propagation through the CNN hardware pipeline.

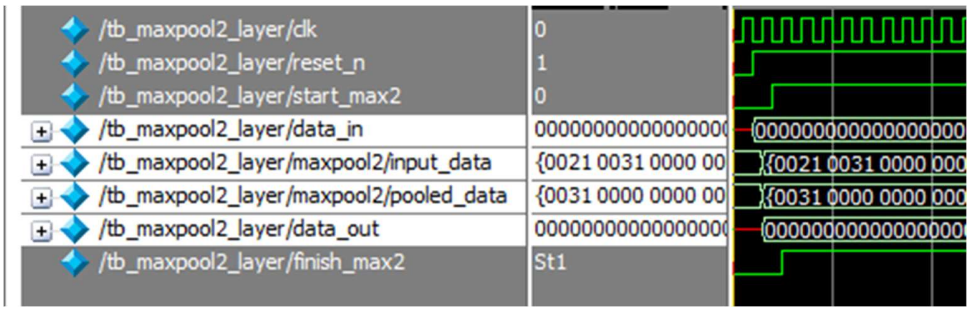


Figure 6.4.5.1. Second MaxPool2x2ReLU Layer Waveform

6.4.6. Second Convolutional Block

Figure 6.4.6.1 presents the waveform simulation of the second convolutional block, which includes convolution, bias addition, and ReLU activation. The signals shown include control inputs such as the clock (clk), reset (reset_n), and start trigger, alongside memory buses for image, weight, and bias data. The image input (image_in_linear) and kernel weights (weights_linear) are correctly fetched from RAM, as observed in the waveforms and memory traces. As the module operates, valid output values begin to populate the block_out_linear bus, replacing the initial undefined (z) or idle states. These values represent the output feature maps illustrated in Figure 6.4.6.2, which has 3 channels with 5x5 size. The finish signal transitions high at the end of the operation, indicating that the entire block has successfully completed processing. This waveform validates the correct execution of the second convolutional block within the FPGA CNN pipeline,

demonstrating synchronized memory access, correct computation, and timely signaling across all stages of the block.

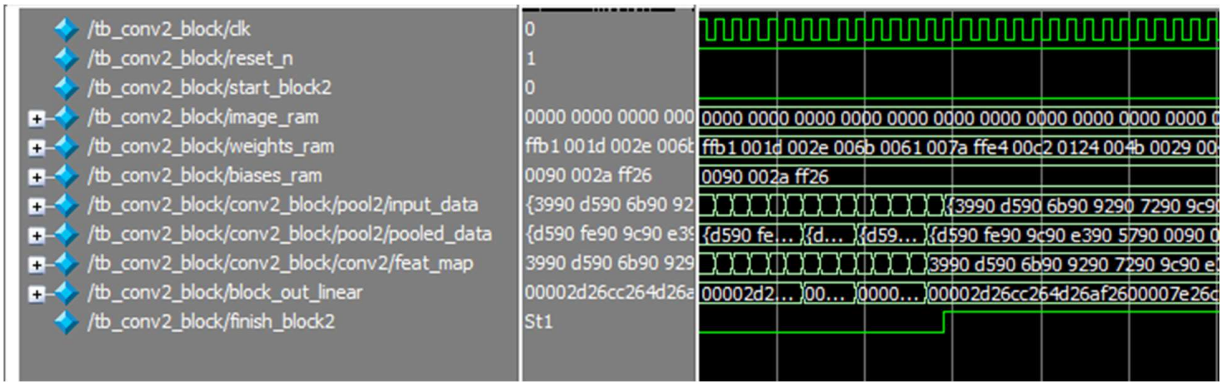


Figure 6.4.6.1. Second Convolutional Block Layer Waveform

```
sim:/tb_conv2_block/conv2_block/pool2/pooled_data @ 3080 ns
0 : d590 fe90 9c90 e390 5790 0090 0090 0000 9490 c490 0090 0090 9a90 aa90 f490 ca90 8590 f490 1e90 4090 e190 0990 6e90 a790 7c90
1 : f32a 3a2a 0000 0000 0000 002a 002a 2f2a d12a d92a d12a 232a 6c2a 392a 652a 6c2a a82a 7f2a 392a af2a 862a 6d2a 0000 fa2a 9c2a
2 : 0000 6526 f626 1e26 1e26 ad26 e226 e726 e426 0000 0000 0000 c226 ff26 ca26 0000 0000 c226 7e26 0000 af26 4d26 cc26 2d26 0000
```

Figure 6.4.6.2. Three Filters of Second Convolutional Block RAM

6.4.7. Flatten Layer

Figure 6.4.7.1 shows the waveform of the flattening operation within the FPGA-based CNN model. The flatten layer receives a multi-dimensional feature map through the data_in signal and converts it into a one-dimensional vector at vector_out, suitable for input to the fully connected (dense) layer. The control signals such as clk, reset_n, and start are used to initiate and synchronize the process. At the beginning, the input and output lines are undefined (x) while the module awaits valid input and activation. Once the start signal is asserted and data becomes valid, the vector_out signal begins transitioning to non-zero values, demonstrating successful serialization of the input tensor. The finish signal is then asserted, indicating the completion of the flattening process. This confirms that the module correctly transforms spatial feature maps into a flattened format, enabling downstream processing by dense layers in the CNN inference pipeline.

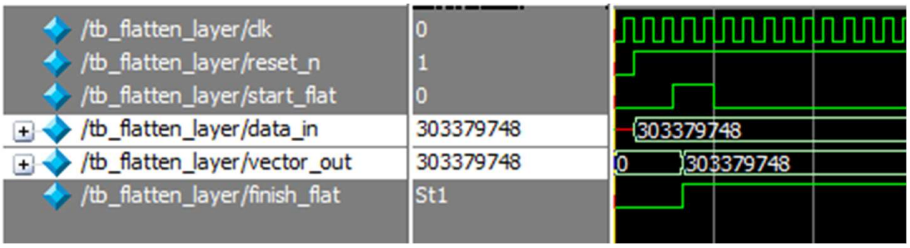


Figure 6.4.7. Flatten Layer Waveform

6.4.8. Fully Connected Layer

Figure 6.4.8 illustrates the simulation waveform of the fully connected (FC) layer, capturing the timing and data flow during the classification phase of the neural network. The clock (clk) and control signals (reset_n, start_fc, finish_fc) manage the operation's sequencing, ensuring synchronized processing. Input data is sourced from image_ram, while the FC computations are driven by weights_ram and biases_ram. The register log_reg shows the intermediate computed values, such as 25622, 1299363, and -3347, reflecting the accumulation and bias addition performed during matrix multiplication. The final prediction output, shown in the predict_out_linear signal, is displayed as a hexadecimal result, representing the encoded class probabilities or logits. The assertion of the finish_fc signal indicates the end of the computation, verifying the correct implementation and functionality of the FC layer in hardware.

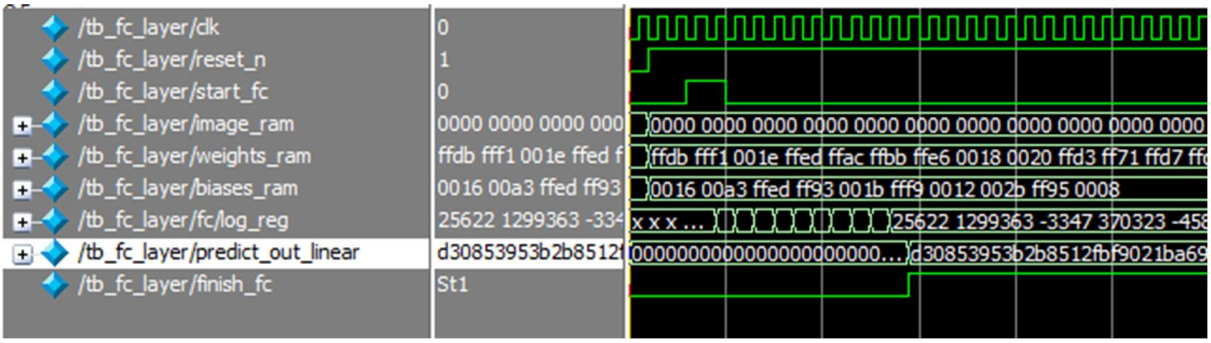


Figure 6.4.8. Fully Connected Layer Waveform

6.4.9. ArgMax Layer

Figure 6.4.9 presents the waveform of the ArgMax layer, which is responsible for determining the predicted class by identifying the index of the maximum value from the fully connected layer's output. The simulation begins with the start_argmax signal initiating the operation, while the clk signal drives the synchronous logic. The input data, input_data_linear, contains a vector of classification scores, such as, representing potential class logits. As the operation progresses, the val_array holds the probability for each digit. The index_out signal eventually outputs the index of the maximum value—in this case is 7—which indicates the predicted class. The finish_argmax signal is asserted once the comparison is complete, confirming the successful execution of the ArgMax operation and enabling downstream modules to use the classification result.

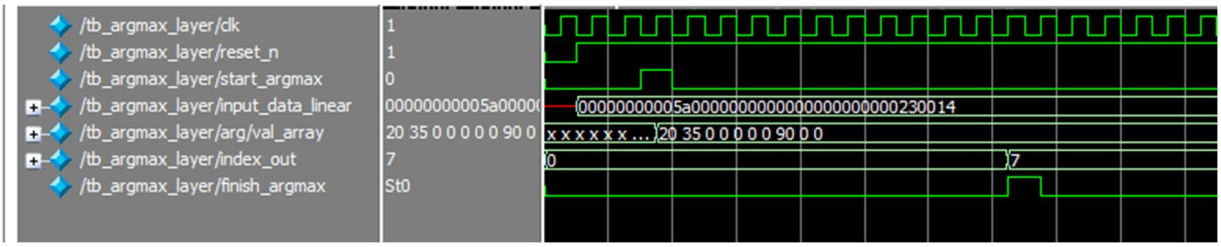


Figure 6.4.9. ArgMax Layer Waveform

6.4.10. Top Module Layer

For each digit, three tests are executed between 16 bits and 32 bits to verify the validation of the design. Each decimal value in the val_array represents the probability of the digit prediction, the index at the highest value is the predicted digit output. Some test cases are not accurate, however, the difference between the probabilities is not significantly distinct. The 32-bit tests have a higher accuracy than the 16-bit tests due to more precision in calculation, in 32-bit tests, the fraction part of the value is 8-digit longer than the 16-bit tests. Sixty figures from 6.4.10.1 to 6.4.10.60 demonstrated digits recognition test, each digit has 3 different image input written by hand.

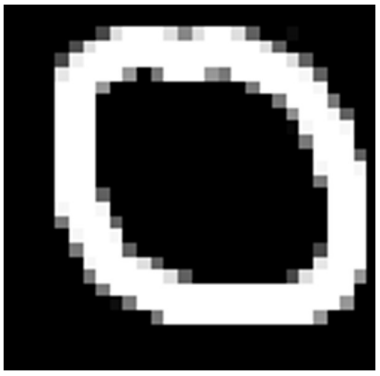


Figure 6.4.10.1. Raw Image Input – Number 0 First Test (16 bits)

Figure 6.4.10.2. Top Module Layer Waveform – Number 0 First Test (16 bits)

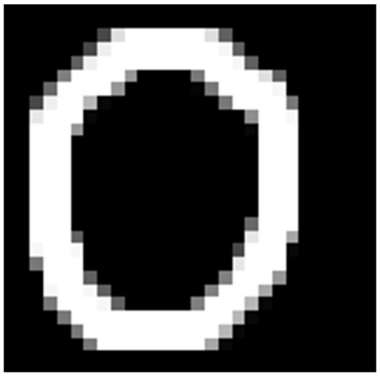


Figure 6.4.10.3. Raw Image Input – Number 0 Second Test (32 bits)

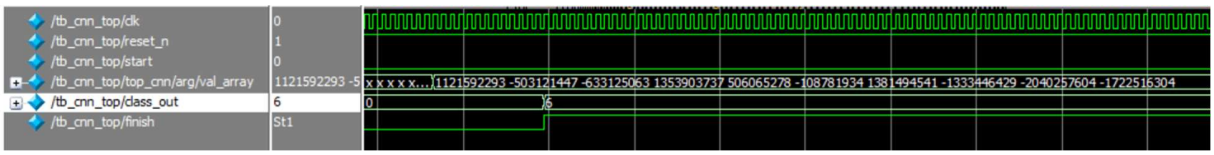


Figure 6.4.10.4. Top Module Layer Waveform – Number 0 Second Test (32 bits)

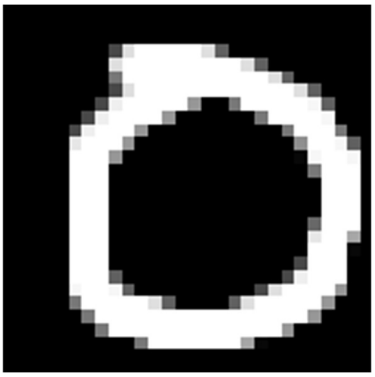


Figure 6.4.10.5. Raw Image Input – Number 0 Third Test (32 bits)

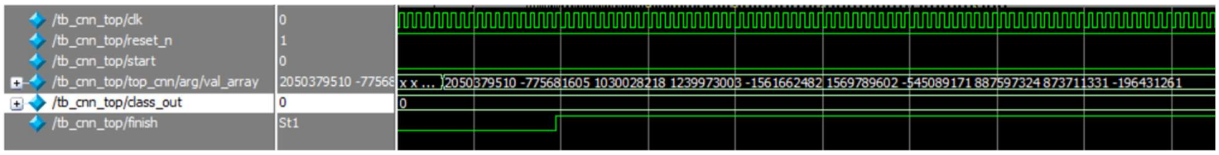


Figure 6.4.10.6. Top Module Layer Waveform – Number 0 Third Test (32 bits)

Figure 6.4.10.7. Raw Image Input – Number 1 First Test (16 bits)

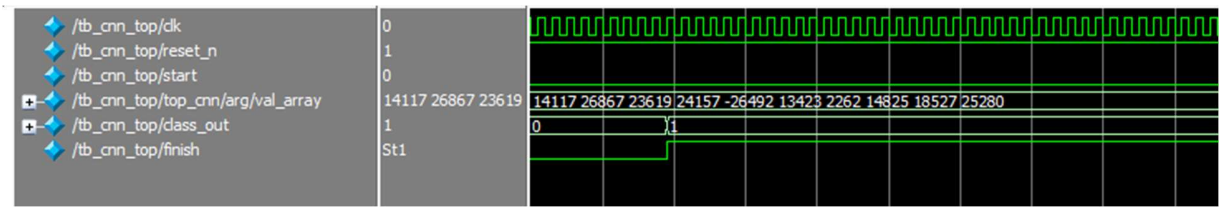


Figure 6.4.10.8. Top Module Layer Waveform – Number 1 First Test (16 bits)

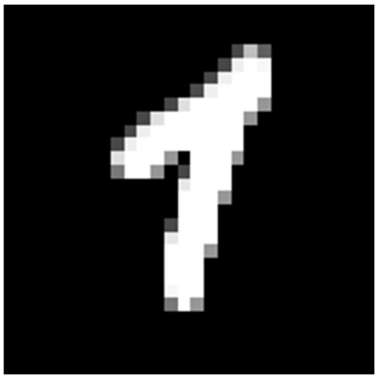


Figure 6.4.10.9. Raw Image Input – Number 1 Second Test (32 bits)

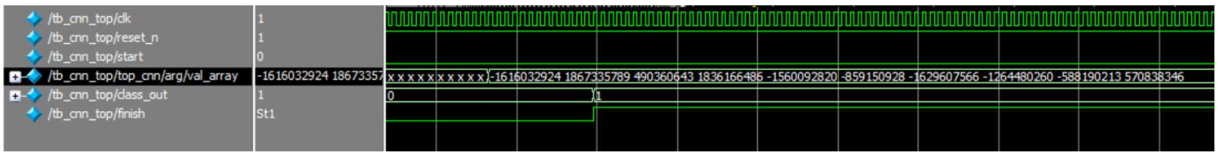


Figure 6.4.10.10. Top Module Layer Waveform – Number 1 Second Test (32 bits)

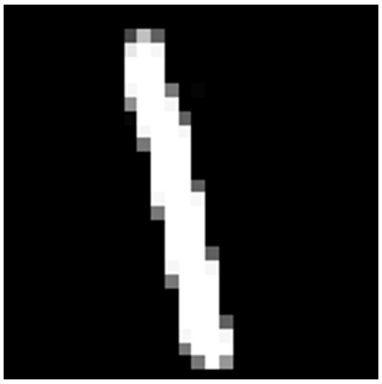


Figure 6.4.10.11. Raw Image Input – Number 1 Third Test (32 bits)

Figure 6.4.10.12. Top Module Layer Waveform – Number 1 Third Test (32 bits)

Figure 6.4.10.13. Raw Image Input – Number 2 First Test (16 bits)

Figure 6.4.10.14. Top Module Layer Waveform – Number 2 First Test (16 bits)

Figure 6.4.10.15. Raw Image Input – Number 2 Second Test (16 bits)

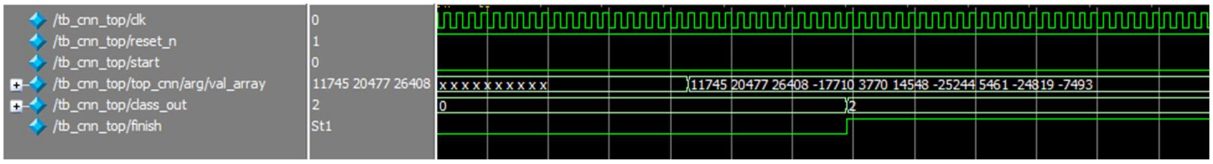


Figure 6.4.10.16. Top Module Layer Waveform – Number 2 Second Test (16 bits)

Figure 6.4.10.17. Raw Image Input – Number 2 Third Test (32 bits)

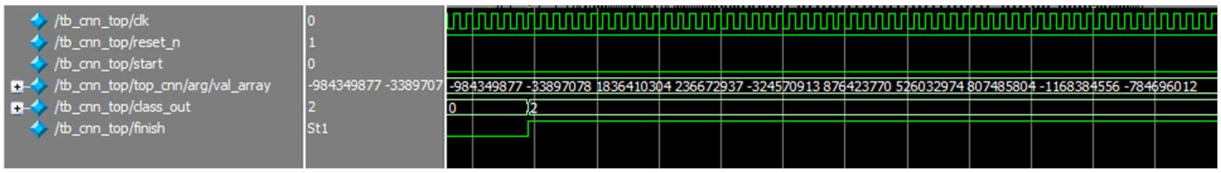


Figure 6.4.10.18. Top Module Layer Waveform – Number 2 Third Test (32 bits)

Figure 6.4.10.19. Raw Image Input – Number 3 First Test (16 bits)

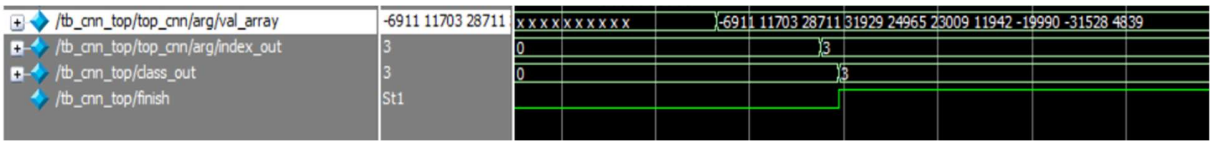


Figure 6.4.10.20. Top Module Layer Waveform – Number 3 First Test (16 bits)

Figure 6.4.10.21. Raw Image Input – Number 3 Second Test (16 bits)

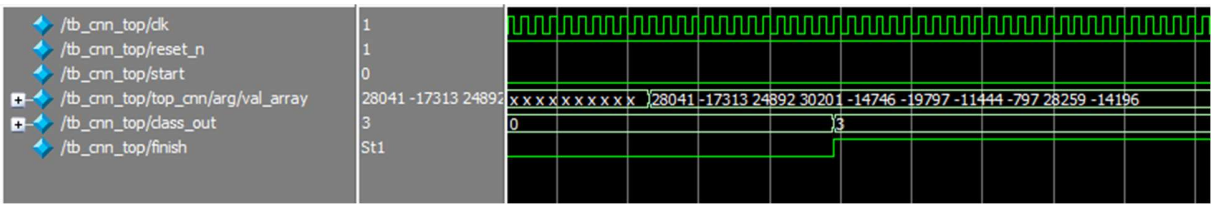


Figure 6.4.10.22. Top Module Layer Waveform – Number 3 Second Test (16 bits)

Figure 6.4.10.23. Raw Image Input – Number 3 Third Test (32 bits)

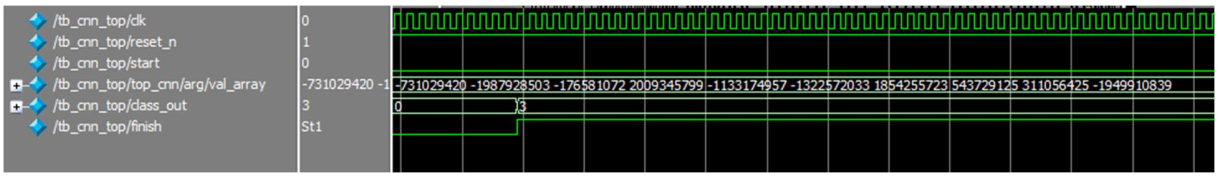


Figure 6.4.10.24. Top Module Layer Waveform – Number 3 Third Test (32 bits)

Figure 6.4.10.25. Raw Image Input – Number 4 First Test (16 bits)

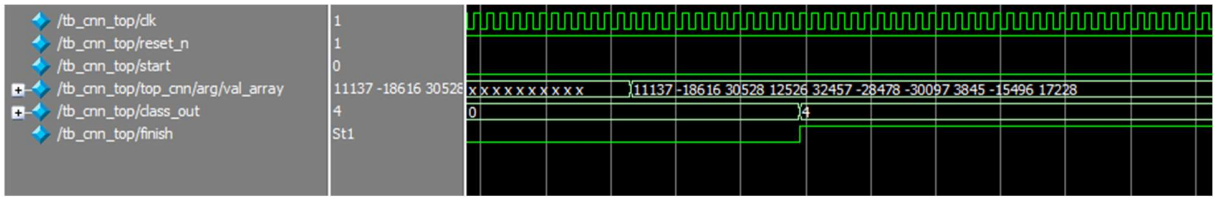


Figure 6.4.10.26. Top Module Layer Waveform – Number 4 First Test (16 bits)

Figure 6.4.10.27. Raw Image Input – Number 4 Second Test (32 bits)

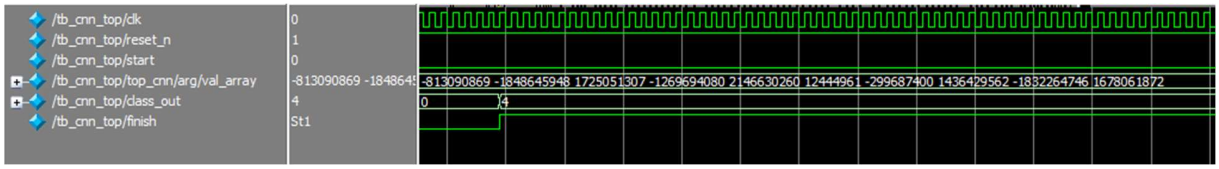


Figure 6.4.10.28. Top Module Layer Waveform – Number 4 Second Test (32 bits)

Figure 6.4.10.29. Raw Image Input – Number 4 Third Test (32 bits)

Figure 6.4.10.30. Top Module Layer Waveform – Number 4 Third Test (32 bits)

Figure 6.4.10.31. Raw Image Input – Number 5 First Test (16 bits)

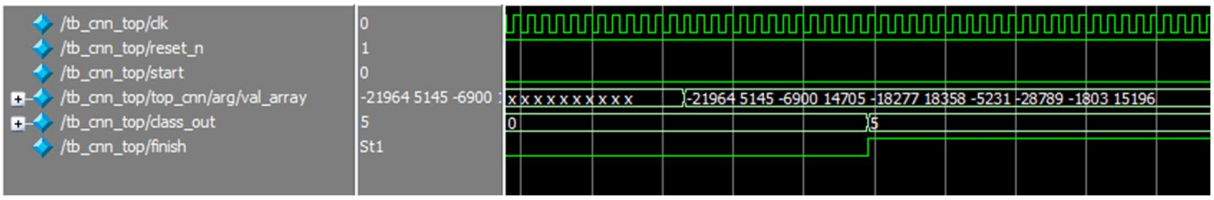


Figure 6.4.10.32. Top Module Layer Waveform – Number 5 First Test (16 bits)

Figure 6.4.10.33. Raw Image Input – Number 5 Second Test (32 bits)

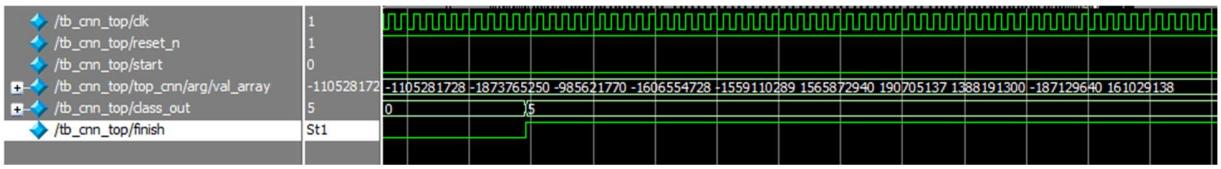


Figure 6.4.10.34. Top Module Layer Waveform – Number 5 Second Test (32 bits)

Figure 6.4.10.35. Raw Image Input – Number 5 Third Test (32 bits)

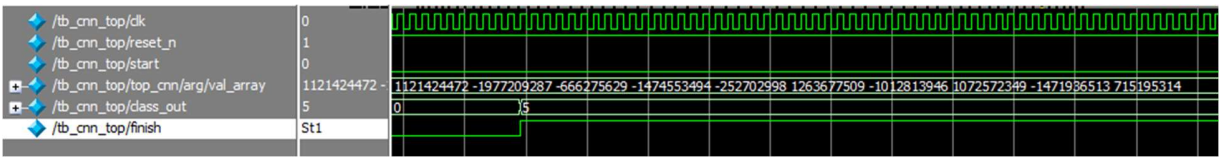


Figure 6.4.10.36. Top Module Layer Waveform – Number 5 Third Test (32 bits)

Figure 6.4.10.37. Raw Image Input – Number 6 First Test (16 bits)

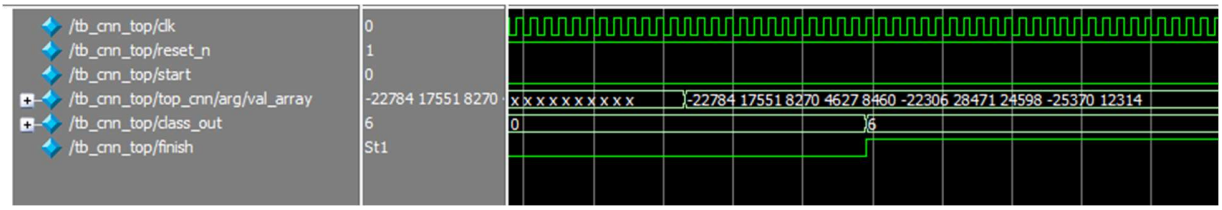


Figure 6.4.10.38. Top Module Layer Waveform – Number 6 First Test (16 bits)

Figure 6.4.10.39. Raw Image Input – Number 6 Second Test (16 bits)

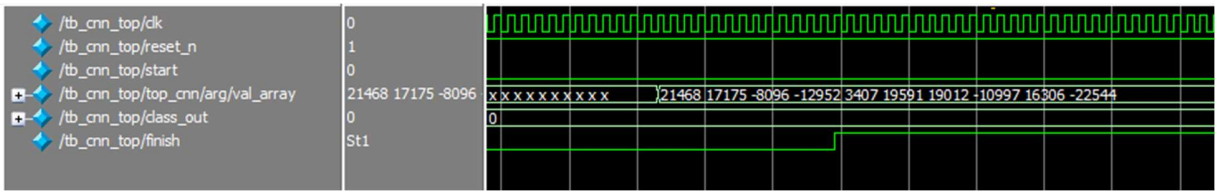


Figure 6.4.10.40. Top Module Layer Waveform – Number 6 Second Test (16 bits)

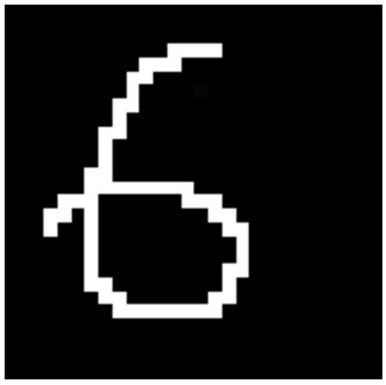


Figure 6.4.10.41. Top Module Layer Waveform – Number 6 Third Test (32 bits)

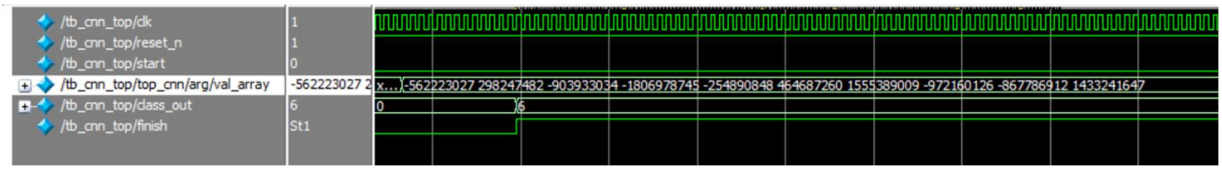


Figure 6.4.10.42. Top Module Layer Waveform – Number 6 Third Test (32 bits)

Figure 6.4.10.43. Raw Image Input – Number 7 First Test (16 bits)

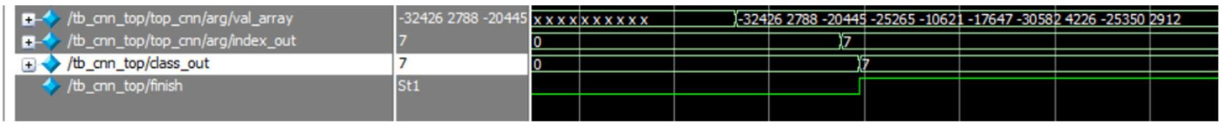


Figure 6.4.10.44. Top Module Layer Waveform – Number 7 First Test (16 bits)

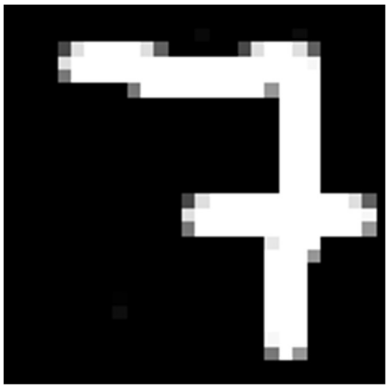


Figure 6.4.10.45. Raw Image Input – Number 7 Second Test (16 bits)

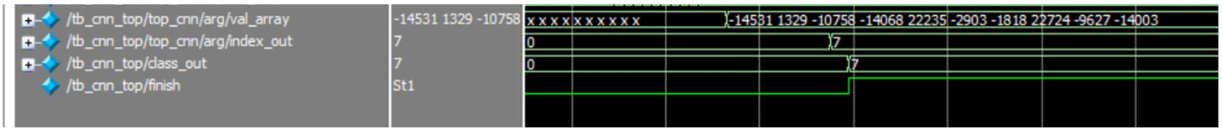


Figure 6.4.10.46. Top Module Layer Waveform – Number 7 Second Test (16 bits)

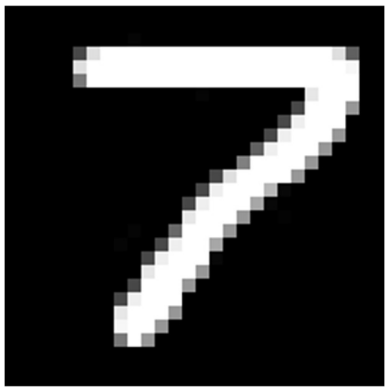


Figure 6.4.10.47. Raw Image Input – Number 7 Third Test (32 bits)

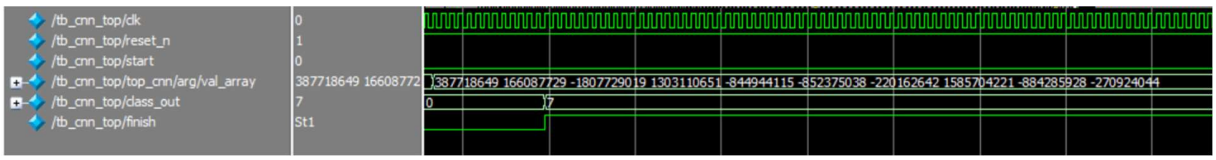


Figure 6.4.10.48. Top Module Layer Waveform – Number 7 Third Test (32 bits)

Figure 6.4.10.49. Raw Image Input – Number 8 First Test (16 bits)

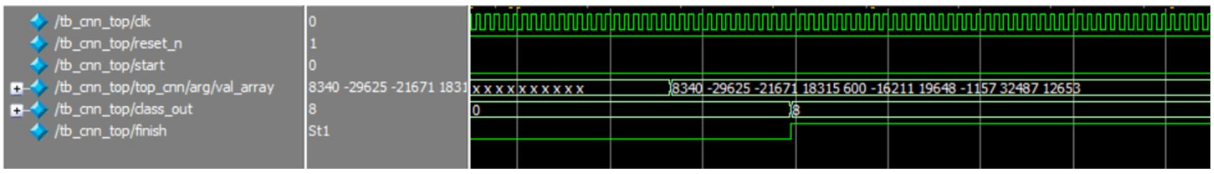


Figure 6.4.10.50. Top Module Layer Waveform – Number 8 First Test (16 bits)

Figure 6.4.10.51. Raw Image Input – Number 8 Second Test (16 bits)

Figure 6.4.10.52. Top Module Layer Waveform – Number 8 Second Test (16 bits)

Figure 6.4.10.53. Raw Image Input – Number 8 Third Test (32 bits)

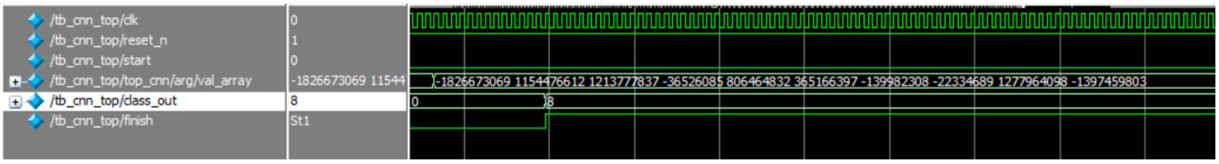


Figure 6.4.10.54. Top Module Layer Waveform – Number 8 Third Test (32 bits)

Figure 6.4.10.55. Raw Image Input – Number 9 First Test (16 bits)

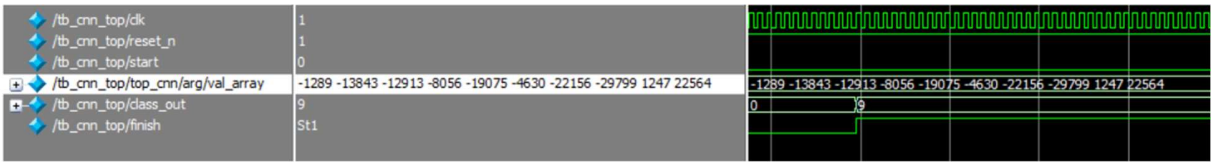


Figure 6.4.10.56. Top Module Layer Waveform – Number 9 First Test (16 bits)

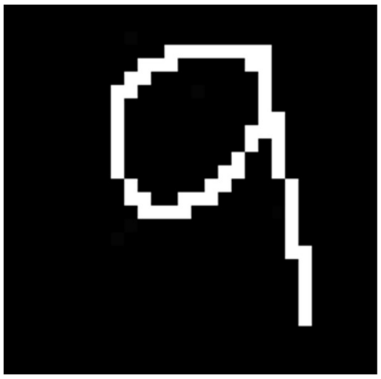


Figure 6.4.10.57. Raw Image Input – Number 9 Second Test (32 bits)

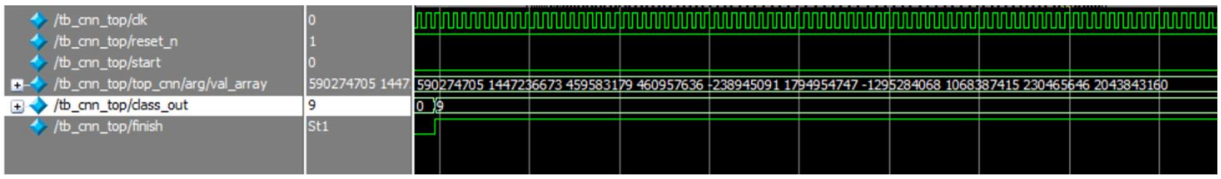


Figure 6.4.10.58. Top Module Layer Waveform – Number 9 Second Test (32 bits)

Figure 6.4.10.59. Raw Image Input – Number 9 Third Test (32 bits)

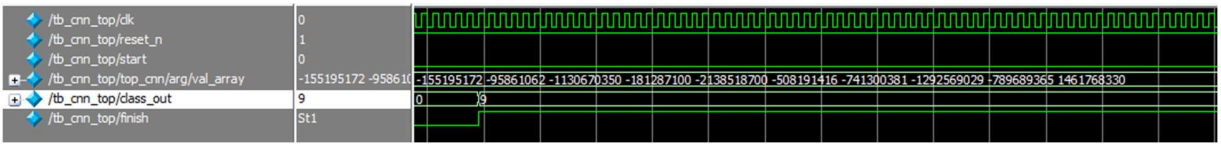


Figure 6.4.10.60. Top Module Layer Waveform – Number 9 Third Test (32 bits)

CHAPTER VII

CONCLUSION AND FUTURE WORK

7.1. Conclusion

In conclusion, the successful design and implementation of the handwritten digit recognition system on the Altera DE2 FPGA Board demonstrates the effectiveness of integrating machine learning with hardware acceleration. By utilizing both a simple CNN and VGG16 for model evaluation, the project identified a reliable architecture capable of delivering high-accuracy predictions. The process of training, exporting, and deploying model weights and biases onto the FPGA using Verilog was executed effectively by observing the waveform. The outcome validates the feasibility of FPGA-based CNN deployment and highlights the potential for future developments in embedded machine learning and real-time digital signal processing applications.

7.2. Future work

For future work, several enhancements can be made to improve the system's functionality, accuracy, and usability. First, the top-level Verilog module can be further optimized to improve clarity, modularity, and signal routing, making debugging and integration more efficient. Implementing a more structured and synchronized control flow using finite state machines (FSMs) will help ensure more predictable operation and easier waveform analysis during simulation. Expanding the input interface to accept live digit input through external devices such as a camera or touchscreen, rather than static images, would enhance system interactivity. Furthermore, deploying more advanced quantization and compression techniques for weights and activations could reduce memory usage while maintaining accuracy, making the design even more resource efficient.

CHAPER VIII

BUSINESS, SOCIAL AND ETHICAL CONSIDERATIONS

The implementation of machine learning models on FPGA hardware, such as in this handwritten digit recognition system, presents various business, social, and ethical considerations. From a business perspective, deploying efficient, low-power CNN inference on FPGA platforms offers significant advantages in edge computing, embedded systems, and real-time applications. This technology can be commercially applied in banking (e.g., check processing), postal services, automated data entry, and educational tools, where accurate and fast digit recognition is critical. The use of reconfigurable hardware also reduces long-term costs by allowing updates without full hardware replacement.Socially, such systems can improve accessibility and automation in various sectors, promoting digital literacy and reducing manual workloads. They may also contribute to inclusive technology, such as assistive devices for individuals with disabilities or tools for non-digital native populations. However, care must be taken to ensure these systems are deployed equitably and do not inadvertently replace human roles in ways that cause social disruption without proper reskilling initiatives.Ethically, it is crucial to ensure transparency, reliability, and fairness in the development and deployment of AI-driven systems. The dataset used must be diverse and representative to avoid bias in recognition accuracy. In summary, while the project offers promising technical and business opportunities, it must be guided by careful consideration of social impact, ethical responsibility, and sustainable commercial deployment.

REFERENCES

[1] Chris Spear and Greg Tumbush*, System Verilog for Verification. A Guide to Learning the Testbench Language Features,* Third edition, New York, United State, 2014.

[2] Nennie Farina Mahat, *Design of a 9-bit UART Module Based On Verilog HDL*, Kualar Lumpur, Malaysia, 2012

[3] Mason Leblanc, *Convolutional Neural Networks: A Comprehensive Guide to the Foundations, Architectures, and Applications of CNNs in Deep Learning and AI,* October 4, 2024.

[4] Akshi Kumar, *Handwritten Digit Recognition Using Deep Learning,* September 11, 2017.

[5] Thi Dieu Linh Nguyen, Elena Verdú, Anh Ngoc Le, Maria Ganzha, *Intelligent Systems and Networks,* August 19, 2023.

APPENDICES

A. Simple CNN Model

Device Configuration

In [69]:
```
import torch
# Device configuration
device = torch.device('cpu')
device
```

Out[69]:
```
device(type='cpu')
```

Import Dataset

In [70]:
```
from torchvision import datasets
from torchvision.transforms import ToTensor
train_data = datasets.MNIST(
    root = 'data',
    train = True,
    transform = ToTensor(),
    download = True,
)
test_data = datasets.MNIST(
    root = 'data',
    train = False,
    transform = ToTensor()
)
```

Load data preparing to train

In [74]:
```
from torch.utils.data import DataLoader
train_dataloader = DataLoader(train_data, batch_size=600, shuffle=True)
test_dataloader = DataLoader(test_data, batch_size=600, shuffle=True)
```

In [75]:
```
import torch.nn as nn
class CNN(nn.Module):
    def __init__(self):
        super(CNN, self).__init__()
        self.conv1 = nn.Sequential(
            nn.Conv2d(
                in_channels=1,
                out_channels=2,
                kernel_size=5,
                stride=1,
                padding=0,
            ),
            nn.ReLU(),
            nn.MaxPool2d(kernel_size=2),
        )
        self.conv2 = nn.Sequential(
            nn.Conv2d(2, 3, 3, 1, 0),
            nn.ReLU(),
            nn.MaxPool2d(2),
        )
        # Fully connected layer, output 10 classes
        self.out = nn.Linear(3 * 5 * 5, 10)
    def forward(self, x):
        x = self.conv1(x)
        x = self.conv2(x)
        # flatten the output of conv2 to (batch size, 32 * 7 * 7)
        x = x.view(x.size(0), -1)
        output = self.out(x)
        return output, x    # return x for visualization
```

In [78]:
```
from torch import optim
optimizer = optim.Adam(cnn.parameters(), lr = 0.01)
optimizer
```

In [79]:

```
from torch.autograd import Variable
num_epochs = 10
def train(num_epochs, cnn, loaders):

    cnn.train()

    # Train model
    total_step =  len(train_dataloader)
    for epoch in range(num_epochs):
         for i, (images, labels) in enumerate(train_dataloader):
             # Give batch data, normalize x when iterate train_loader
             b_x = Variable(images)   # batch x
             b_y = Variable(labels)   # batch y
             output = cnn(b_x)[0]
             loss = loss_func(output, b_y)
             # clear gradients for this training step
             optimizer.zero_grad()

             # backpropagation, compute gradients
             loss.backward()
             # apply gradients
             optimizer.step()

             if (i+1) % 100 == 0:
                 print ('Epoch [{}/{}], Step [{}/{}], Loss: {:.4f}'
                        .format(epoch + 1, num_epochs, i + 1, total_step, loss.item()))
                 # pass
          # pass
      # pass
train(num_epochs, cnn, train_dataloader)
```

In [80]:

```
def test():
  # Test the model
  cnn.eval()
  with torch.no_grad():
      correct = 0
      total = 0
      for i, (images, labels) in enumerate(test_dataloader):
          test_output, last_layer = cnn(images)
          pred_y = torch.max(test_output, 1)[1].data.squeeze()
          accuracy = (pred_y == labels).sum().item() / float(labels.size(0))
          pass
          print('Test Accuracy of the model on the 10000 test images: %.2f' % accuracy)
          pass
test()
```

In [87]:

```
print('Model state_dict:')
for param_tensor in cnn.state_dict():
    print(param_tensor, "\t", model.state_dict()[param_tensor].size())

# Print optimizer's state_dict
print('Parameters')
for name, param in model.named_parameters():
    if param.requires_grad:
        print(name, "\n", param.data)
```

```
Parameters
conv1.0.weight
 tensor([[[[ 0.1027, -0.1600, -0.1231, -0.0204,  0.1641],
          [ 0.1342,  0.4724,  0.7942,  0.4118,  0.1202],
          [ 0.2299,  0.5250,  0.6427,  0.3914,  0.5267],
          [ 0.0911,  0.0792, -0.0830,  0.3659,  0.5461],
          [-0.0836,  0.1277,  0.2579,  0.2825,  0.4228]]],


        [[[ 0.4518,  0.1350, -0.0849,  0.2220,  0.9772],
          [ 0.6005,  0.5837,  0.6235,  0.5671,  0.0920],
          [ 0.2693,  0.8749,  1.1374,  0.4500,  0.0737],
          [-1.1765, -0.9229, -1.0997, -0.6078, -0.0682],
          [-0.4302, -0.8660, -0.5325, -0.0360,  0.0271]]]])
conv1.0.bias
 tensor([-0.0329,  0.0210])
```

```
conv2.0.weight
 tensor([[[[ 0.2220, -0.5584, -0.5548],
           [-0.2858, -0.6273,  0.3947],
           [-0.3715, -0.1783,  0.6155]],

          [[ 0.1321,  0.4670,  0.2784],
           [ 0.6525,  0.1232, -0.7473],
           [-0.3478, -0.2348,  0.0567]]],


         [[[-0.7856, -0.7645, -0.6510],
           [ 0.0502, -0.2146, -0.1427],
           [ 0.4796,  0.5612,  0.1099]],

          [[ 0.0987,  0.7185,  0.9346],
           [ 0.0759, -0.2976,  0.2438],
           [ 0.2201, -0.1353, -0.1484]]],


         [[[-0.3146,  0.0650, -0.0919],
           [ 0.2711,  0.1592,  0.1563],
           [-0.1413,  0.0043,  0.0559]],

          [[ 0.1131,  0.2231,  0.0074],
           [ 0.5374,  0.4241,  0.0101],
           [ 0.1828,  0.2742,  0.3573]]]])
conv2.0.bias
 tensor([ 0.5919,  0.0174, -0.0912])
```

```
out.weight
 tensor([[ 1.2590e-01,  1.7423e-01, -1.1697e-01,  7.8632e-02, -2.2715e-01,
           1.6016e-01,  1.8257e-01, -4.7618e-01, -1.5495e-01, -8.3346e-01,
          -5.9894e-02, -3.3716e-01,  1.4041e-02,  1.4780e-02, -8.8312e-01,
          -5.0943e-01, -3.9915e-01, -3.7103e-01, -2.6598e-01,  2.7425e-01,
          -3.0650e-01, -5.8047e-01,  2.5383e-01,  4.7415e-01,  7.3430e-01,
          -3.0098e-02, -1.2895e-02,  4.8265e-02,  1.2132e-01, -7.3929e-02,
           1.4242e-01,  2.0702e-01, -2.1031e-01,  1.0990e-01, -1.1270e-01,
          -5.9445e-01, -7.1520e-01,  9.6179e-02, -3.0214e-01, -8.0249e-01,
          -1.0584e+00, -5.1217e-01,  4.0718e-01,  1.3079e-02, -8.0991e-01,
          -1.7587e-01,  6.8624e-02,  6.1310e-01, -8.5488e-02, -2.5026e-01,
          -1.0341e-03, -3.2017e-01,  7.5627e-03,  1.1876e-01, -9.1084e-03,
           6.0919e-02,  2.2486e-01,  2.3304e-01,  1.7631e-01,  2.0531e-01,
           1.3548e-01, -5.4571e-02, -5.8380e-01, -2.8579e-01,  3.9145e-02,
           1.6843e-01, -1.3098e-01, -5.3251e-01, -3.3119e-01,  2.1391e-01,
           2.0365e-02,  2.5871e-01,  3.0231e-01,  2.2347e-01,  8.2336e-02],
         [-2.6240e-01,  5.7727e-01,  7.5022e-01,  1.1286e+00,  1.4363e-01,
          -5.9789e-01,  2.2451e-01,  3.9173e-01,  1.0343e-04,  4.9248e-01,
           8.1926e-02,  6.5883e-01,  1.7928e-01, -3.2762e-01,  2.1309e-01,
           4.7268e-02,  2.3669e-01,  1.6041e-01,  4.4013e-01, -3.3977e-01,
           1.6948e-01,  7.8320e-02,  5.0579e-02, -1.7976e-01, -9.3011e-03,
           3.9965e-01, -3.7579e-01,  1.5283e-01, -5.3167e-01, -3.3253e-02,
           9.4004e-02, -5.0163e-03, -4.0249e-02, -5.0623e-01, -3.5583e-01,
           3.0141e-01, -7.2123e-01, -1.1309e+00, -6.3580e-01,  1.8497e-01,
           2.7050e-01,  3.9481e-02, -1.0322e+00, -1.3559e-01,  5.8780e-01,
          -1.1639e-01, -3.6303e-01, -1.7988e-01,  4.0366e-01,  6.1873e-01,
          -2.6765e-02,  1.6085e-03, -1.3900e-01, -6.4854e-01, -1.4893e-01,
          -3.6106e-01, -1.5600e-01,  5.2058e-01, -4.1703e-01, -3.2318e-01,
          -4.9499e-01, -1.2073e-01,  1.9493e-01, -3.5678e-01, -4.3260e-01,
          -2.5491e-01,  2.4930e-02,  3.0761e-01, -1.9622e-01, -3.7100e-01,
           3.5245e-01,  3.8906e-02,  3.0035e-01, -9.9291e-02,  4.3725e-01],
         [-1.5582e-01, -1.3243e-01, -4.7023e-01, -7.3085e-01, -5.8890e-01,
           2.1135e-01, -1.9282e-01, -3.8708e-02, -1.0555e-01, -1.1861e+00,
           7.4453e-01,  5.7281e-01, -5.8702e-02,  3.4184e-01,  1.8114e-01,
           3.5744e-01,  4.8024e-01,  1.6730e-01,  5.6030e-03,  3.2801e-01,
          -1.9979e-01,  4.6176e-02,  2.1207e-01, -1.6783e-01,  2.3093e-01,
           3.4853e-03,  1.9352e-02,  9.2615e-02,  7.0386e-03, -8.2842e-02,
           7.8819e-01,  3.7659e-01,  3.7559e-01,  3.1276e-03, -4.2343e-01,
           7.9246e-01,  2.1632e-01,  3.6203e-01, -3.9990e-02, -4.6926e-01,
          -1.8110e-01, -3.4253e-01, -1.8890e-01, -1.2983e-01,  5.0474e-01,
          -6.9326e-01, -1.6621e-01, -5.1735e-02,  1.3394e-01,  1.1639e+00,
           1.0607e-01,  4.8519e-02,  4.5270e-02,  7.0278e-02, -2.1408e-01,
          -1.6006e-01, -8.5684e-02, -2.8951e-01, -1.3423e-01, -3.6884e-01,
          -1.5183e-01, -1.5033e-01, -1.1547e-01, -2.4998e-01,  6.8535e-02,
           5.8670e-02,  1.9911e-02, -2.0078e-02, -6.1436e-02,  3.5844e-01,
           1.6318e-01,  1.7788e-01, -7.2086e-02,  1.6896e-01,  3.6979e-01],
```

```
[-8.0804e-01, -4.0291e-01, -6.8103e-01, -1.0088e+00, -1.0926e+00,
  5.5864e-02,  5.4670e-01, -2.5178e-02, -1.4243e-01, -1.3049e-01,
  2.2077e-01,  4.3431e-01, -1.0838e-01, -3.6578e-02,  5.0736e-01,
 -1.8432e-01, -1.1093e-01, -1.8187e-02, -5.2019e-01, -4.1071e-01,
  2.3439e-01,  1.2556e-01,  1.3583e-01,  1.8835e-01,  8.8800e-02,
  2.7382e-02,  3.7896e-01,  1.5156e-01, -9.5453e-02, -3.8959e-02,
  3.7379e-01,  1.1047e-01, -4.2293e-02, -5.9401e-02, -3.6883e-01,
  4.3738e-01,  1.9549e-01, -1.8878e-03,  7.1636e-02,  3.9559e-01,
  7.4064e-01,  3.1170e-01,  2.5923e-01,  3.8239e-01, -1.6845e-01,
  4.9004e-01,  5.4327e-01,  1.1420e-01, -2.0179e-01, -1.0406e+00,
 -1.4593e-02,  1.8880e-01,  1.1601e-01,  1.1814e-01, -1.4754e-01,
 -2.5084e-02, -6.1002e-02,  1.2061e-01,  3.6093e-02, -2.2030e-01,
 -2.9619e-01, -1.5440e-02,  2.6409e-01,  1.4566e-01, -5.6548e-02,
 -3.3512e-01, -2.2084e-01, -1.0385e-01, -1.6792e-01, -2.9185e-01,
  1.3797e-01,  2.3211e-03, -1.1017e-01,  3.2977e-02, -1.0826e-01],
[ 4.9039e-01,  5.6222e-01,  5.2426e-01,  3.1744e-01, -1.2399e-01,
  4.2211e-01, -6.7039e-01, -1.1291e-01, -2.0700e-01,  4.7521e-01,
 -1.4217e-01, -5.8533e-01,  2.9412e-01, -6.2917e-01,  6.1115e-01,
  1.2221e-01,  6.4098e-02, -2.0138e-01,  5.9490e-02,  5.1433e-01,
  5.1219e-01,  2.0514e-01, -9.0407e-02, -3.1231e-01, -1.9624e-01,
 -1.2510e-02, -7.0904e-02, -6.3400e-01, -3.8813e-01, -3.1199e-01,
 -3.5438e-01, -1.6614e-01, -4.2783e-01, -7.9429e-01, -4.1622e-01,
 -5.5415e-01, -1.1015e-01,  2.2357e-01, -1.0132e+00, -3.4561e-01,
 -6.4427e-02,  4.3672e-02, -1.5804e-01, -7.2156e-01, -1.2425e-01,
 -4.8349e-02, -3.7598e-01, -5.2861e-01, -3.3956e-01,  3.6546e-01,
  5.5193e-01,  2.2422e-02, -4.5293e-01, -2.8573e-01, -2.4928e-02,
  7.3757e-02,  2.2876e-02, -4.7816e-01, -2.1752e-01, -2.5718e-01,
  2.5665e-01,  2.5353e-01,  1.9235e-01,  2.5150e-01,  9.9595e-02,
  2.7098e-01, -2.9063e-02,  2.2074e-01,  6.5218e-01,  1.7204e-01,
 -1.4693e-01, -2.0180e-01, -2.6464e-02,  8.9305e-02,  4.7691e-02],
[-1.0260e-01,  1.7571e-01, -2.6048e-01,  4.6097e-01,  9.8586e-01,
 -4.0710e-01, -3.5118e-01, -3.7013e-01,  3.2359e-01,  4.8702e-01,
 -4.7832e-01, -4.5265e-01, -2.4007e-01, -1.1209e-01,  1.1973e-01,
  9.6908e-02, -5.8204e-02,  1.5238e-01, -3.1906e-01, -8.3427e-01,
 -4.7075e-02,  1.5582e-01,  1.5512e-01,  1.3534e-01, -2.1727e-01,
 -5.2756e-01,  5.8631e-04,  8.4100e-02, -5.1139e-02, -1.0927e-01,
 -7.7603e-01, -6.8601e-02,  2.4737e-01,  3.0443e-01,  5.9222e-01,
  3.2450e-01,  3.0727e-02,  2.5099e-01,  4.1710e-01,  7.6491e-01,
  4.7408e-01,  3.2064e-01,  2.0608e-01,  3.9027e-01,  4.2510e-02,
  2.7315e-01,  5.4911e-01,  4.3827e-01, -7.0547e-02, -5.8598e-01,
  1.0960e-01,  4.8787e-02, -2.4789e-01,  3.7811e-02,  1.5036e-01,
 -8.7938e-02,  1.1339e-02, -1.1447e-01,  1.8032e-01,  4.1995e-01,
  7.7076e-02,  5.6587e-02, -1.4733e-02, -6.6629e-02, -4.5176e-01,
 -5.7709e-02,  8.4980e-02, -4.0565e-01, -4.1415e-01, -2.8280e-01,
 -6.4005e-02,  1.4946e-01,  6.4026e-02, -6.9023e-02, -4.0875e-02],
```

```
[-3.7568e-03,  1.1289e-01,  4.3107e-01,  2.6121e-01,  4.7716e-01,
  1.9967e-01,  2.8572e-01,  1.9357e-01,  2.3879e-01,  7.8048e-01,
 -3.7982e-01,  1.1017e-01, -9.5435e-02,  1.6622e-02, -8.0347e-01,
 -8.6376e-01, -7.6341e-01, -7.2363e-03, -3.3486e-01,  1.1316e-01,
 -6.8792e-01, -9.4644e-01,  4.9213e-02,  2.0738e-01,  1.3850e-01,
 -4.5118e-01, -5.7904e-01, -6.6788e-01, -4.6156e-01,  1.2802e-02,
 -3.9450e-01, -6.2608e-01,  1.8982e-01,  3.7131e-01,  7.5801e-01,
 -1.1517e+00, -6.8058e-01, -1.3255e-03,  8.4628e-01,  7.5777e-02,
 -1.2330e+00, -7.7523e-01,  2.5317e-02,  5.7627e-02,  1.6572e-02,
 -3.3698e-01, -4.2907e-01,  4.0890e-01,  3.7078e-01, -1.8700e-01,
  2.1926e-01, -3.4333e-01, -5.2920e-01, -1.5201e-01, -6.4400e-03,
  1.4086e-01, -1.6241e-01, -3.0916e-01, -4.4870e-02,  1.7371e-01,
  1.5312e-01, -1.2070e-01, -3.9966e-02,  7.9197e-03,  1.5555e-01,
  2.2196e-01,  2.1680e-01,  7.1594e-02, -1.2678e-01, -6.8593e-02,
 -1.0980e-01,  2.1463e-01,  3.6732e-01,  2.5357e-01,  8.2964e-02],
[-1.0949e-01, -2.9411e-01, -5.9214e-01, -1.0385e-01, -3.0290e-02,
 -1.4612e-01,  2.5600e-01,  1.7431e-02, -6.0546e-01, -2.8803e-02,
  5.1238e-01,  1.7306e-01,  3.6159e-01, -1.9748e-01, -4.8328e-01,
  3.9840e-01,  5.2345e-01, -1.0422e-02,  1.8644e-01,  2.4226e-01,
  3.3794e-01,  1.0370e-01, -7.7270e-02, -3.0390e-01,  1.6920e-01,
  1.6220e-01,  6.1407e-01,  2.9634e-01,  3.0304e-01,  1.6476e-01,
  4.2516e-01,  5.7915e-01,  2.7422e-01, -1.1738e-01, -2.2568e-01,
 -1.0154e-01,  2.5949e-01, -4.9181e-01, -6.6178e-01,  2.4676e-01,
  1.4062e-01,  8.6793e-02, -4.9259e-01, -2.6305e-01,  1.5694e-01,
  3.2287e-01, -7.8718e-01, -1.0118e+00,  3.7049e-02, -4.1655e-01,
  2.1274e-01, -6.3326e-02, -1.7103e-01, -8.3457e-02, -1.2461e-01,
  1.3876e-01,  3.0713e-02,  2.6930e-01,  5.0944e-01,  1.4626e-01,
 -1.1706e-02, -1.3989e-01, -3.5523e-01, -1.8714e-01, -9.3068e-02,
 -3.2795e-01,  1.9054e-01, -3.9309e-02,  2.2958e-01,  2.5862e-01,
 -1.4949e-02, -5.7569e-01, -1.5098e-01, -2.4353e-01, -6.4590e-01],
[ 2.4289e-01,  8.7365e-02, -2.0343e-01, -1.4429e-01,  3.1094e-01,
 -2.2306e-01, -5.1294e-01, -3.6199e-01, -1.3721e-01, -5.5888e-02,
 -2.1053e-01, -7.2162e-01, -5.6297e-01, -3.9405e-02,  6.3238e-01,
  3.4937e-01, -5.7338e-02,  6.8830e-02, -1.2748e-01, -1.5701e-01,
 -4.7691e-01, -6.7848e-01, -3.9440e-01, -3.0556e-01, -2.5602e-01,
  5.3471e-02, -2.0976e-01, -1.4776e-01, -8.7485e-02,  4.2189e-02,
 -1.0438e-01, -3.6091e-02,  1.6954e-01, -2.1905e-01, -2.2306e-01,
  1.0928e-01,  1.0130e-01, -9.1677e-02, -1.4619e-02, -8.7368e-03,
 -3.2363e-02, -7.0556e-01, -3.1914e-01,  2.2791e-02,  9.8950e-03,
 -5.1071e-01, -2.4833e-01,  2.9408e-01, -7.5966e-02, -5.3462e-01,
 -2.0240e-01,  1.1426e-01,  7.1653e-02,  2.5304e-01,  9.1131e-02,
  2.2267e-01,  1.7950e-01, -3.7451e-02, -1.0856e-01,  2.2518e-01,
 -4.9928e-03, -1.4112e-02,  2.1013e-01,  1.6839e-01,  2.8787e-01,
 -9.4638e-02,  4.5003e-02,  1.6794e-01,  7.9342e-02, -7.5644e-02,
 -3.1814e-02,  1.7956e-01,  9.6186e-02, -1.5575e-02, -8.7755e-02],
```

```
        [ 1.0303e-01, -2.9063e-01,  4.5087e-02, -3.6343e-01,  5.3423e-02,
          1.5395e-01, -1.0454e-01,  1.2677e-01, -6.4325e-02, -1.8225e-01,
         -7.9891e-01, -7.2238e-01,  6.8554e-03,  4.0584e-02, -3.0220e-01,
         -2.8181e-01, -9.6163e-02, -2.3868e-01,  3.2634e-01, -6.8644e-02,
          3.5461e-01,  2.1483e-01, -1.2154e-01,  8.0624e-02, -8.1956e-01,
          2.9058e-02, -2.8281e-01,  1.6864e-01,  3.7752e-01,  1.9390e-01,
         -3.4294e-01, -1.3586e-01, -2.5107e-01,  5.6114e-02, -2.7623e-02,
         -7.0348e-01,  2.0959e-01, -2.1115e-01, -1.3504e-01, -3.9084e-01,
         -2.2721e-01,  6.9739e-01,  4.0482e-02, -1.3233e+00, -5.8399e-01,
          2.3869e-01,  3.7520e-01, -4.8224e-01, -1.6373e-01,  6.5237e-02,
         -3.4557e-01,  9.1830e-03,  3.2349e-01,  2.7761e-01,  7.9094e-02,
         -7.7327e-03,  2.0100e-01,  1.0542e-01, -3.3393e-02,  2.5489e-02,
          2.1033e-01,  1.0506e-01,  1.2496e-01,  2.7639e-01, -8.8882e-02,
         -7.8844e-02, -4.0633e-02,  2.0048e-01, -2.0784e-02, -3.3605e-01,
         -1.8332e-01, -2.3950e-01, -2.4197e-01, -2.9409e-01, -1.4901e-02]])
out.bias
 tensor([ 0.2748,  0.6083, -0.1970, -0.6693,  0.5291, -0.2911,  0.0876, -0.0701,
         0.1496, -0.2314])
```

```python
import numpy as np

model = CNN()
model.load_state_dict(torch.load('C:/Users/Acer/Downloads/CNN-FPGA-Implementation-main/miscellaneous/model/simplebasedcnn'))
model.eval()
# Example: accessing specific layers
# Replace with the actual names in your model (use print(model) to explore)
conv1_weight = model.conv1[0].weight.data.numpy()
conv1_bias = model.conv1[0].bias.data.numpy()
conv2_weight = model.conv2[0].weight.data.numpy()
conv2_bias = model.conv2[0].bias.data.numpy()
dense_weight = model.out.weight.data.numpy()
dense_bias = model.out.bias.data.numpy()

# Save each as flat text (1 value per line)
np.savetxt("C:/Users/Acer/Downloads/CNN-FPGA-Implementation-main/miscellaneous/model/conv1_weight.txt", conv1_weight.flatten(), fmt="%.8f")
np.savetxt("C:/Users/Acer/Downloads/CNN-FPGA-Implementation-main/miscellaneous/model/conv1_bias.txt", conv1_bias.flatten(), fmt="%.8f")
np.savetxt("C:/Users/Acer/Downloads/CNN-FPGA-Implementation-main/miscellaneous/model/conv2_weight.txt", conv2_weight.flatten(), fmt="%.8f")
np.savetxt("C:/Users/Acer/Downloads/CNN-FPGA-Implementation-main/miscellaneous/model/conv2_bias.txt", conv2_bias.flatten(), fmt="%.8f")
np.savetxt("C:/Users/Acer/Downloads/CNN-FPGA-Implementation-main/miscellaneous/model/dense_weight.txt", dense_weight.flatten(), fmt="%.8f")
np.savetxt("C:/Users/Acer/Downloads/CNN-FPGA-Implementation-main/miscellaneous/model/dense_bias.txt", dense_bias.flatten(), fmt="%.8f")

print("All weights/biases exported to .txt files.")
```

B. VGG-16 Model

```
# importing libraries
import tensorflow
import numpy as np
import matplotlib.pyplot as plt
import os
import random
from tqdm import tqdm # for progress bar

# Libraries for TensorFlow
from tensorflow import keras
from tensorflow.keras.utils import to_categorical, plot_model
from tensorflow.keras.preprocessing import image
from tensorflow.keras import models, layers


# Library for Transfer Learning
from tensorflow.keras.applications import VGG16
from keras.applications.vgg16 import preprocess_input

print("Importing libraries completed.")
```

Data Gathering

```
(x_train,y_train),(x_test,y_test)= keras.datasets.mnist.load_data()
```

```
Downloading data from https://storage.googleapis.com/tensorflow/tf-keras-datasets/mnist.npz
11490434/11490434 ━━━━━━━━━━━━━━━━━━━━ 0s 0us/step
```

Verifying dataset

```
print(x_train.shape)
print(y_train.shape)
print(x_test.shape)
print(y_test.shape)
```

```
(60000, 28, 28)
(60000,)
(10000, 28, 28)
(10000,)
```

Processing data to compact with VGG16

```
x_train=np.dstack([x_train] * 3)
x_test=np.dstack([x_test]*3)
x_train.shape,x_test.shape
```

```
((60000, 28, 84), (10000, 28, 84))
```

Reshape images as per the tensor format required by tensorflow

```
x_train = x_train.reshape(-1, 28,28,3)
x_test= x_test.reshape (-1,28,28,3)
x_train.shape,x_test.shape
```

```
((60000, 28, 28, 3), (10000, 28, 28, 3))
```

Resize the images 48*48 as required by VGG16

```
from keras.preprocessing.image import img_to_array, array_to_img

x_train = np.asarray([img_to_array(array_to_img(im, scale=False).resize((48,48))) for im in x_train])
x_test = np.asarray([img_to_array(array_to_img(im, scale=False).resize((48,48))) for im in x_test])
x_train.shape, x_test.shape
```

```
((60000, 48, 48, 3), (10000, 48, 48, 3))
```

Sorting images

```
class_names=['Zero', 'One', 'Two', 'Three', 'Four', 'Five', 'Six', 'Seven', 'Eight', 'Nine']
print(class_names)

val_class_names =['Zero', 'One', 'Two', 'Three', 'Four', 'Five', 'Six', 'Seven', 'Eight', 'Nine']
print(val_class_names)

test_class_names=['Zero', 'One', 'Two', 'Three', 'Four', 'Five', 'Six', 'Seven', 'Eight', 'Nine']
print(test_class_names)

# Function to know the name of the element

def Get_Element_Name(argument):
    switcher = {
        0: "Zero",
        1: "One",
        2: "Two",
        3: "Three",
        4: "Four",
        5: "Five",
        6: "Six",
        7: "Seven",
        8: "Eight",
        9: "Nine",
    }
    return switcher.get(argument, "Invalid")

print(Get_Element_Name(0))
print(Get_Element_Name(3))
print(Get_Element_Name(8))
```

Preparing Data

```
x=[] # to store array value of the images
x=x_train
y=[] # to store the labels of the images
y=y_train

test_images=[]
test_images=x_test
test_images_Original=[]
test_images_Original=x_test
test_image_label=[] # to store the labels of the images
test_image_label=y_test

val_images=[]
val_images=x_test
val_images_Original=[]
val_images_Original=x_test
val_image_label=[] # to store the labels of the images
val_image_label=y_test # to store the labels of the images

print("Preparing Dataset Completed.")
```

Verifying Data

```
print("Training Dataset")

x=np.array(x) # Converting to np arrary to pass to the model
print(x.shape)

y=to_categorical(y) # onehot encoding of the labels
# print(y)
print(y.shape)

# Test Dataset
print("Test Dataset")

test_images=np.array(test_images)
print(test_images.shape)

test_image_label=to_categorical(test_image_label) # onehot encoding of the labels)
print(test_image_label.shape)

# Validation Dataset
print("Validation Dataset")

val_images=np.array(val_images)
print(val_images.shape)

val_image_label=to_categorical(val_image_label) # onehot encoding of the labels)
print(val_image_label.shape)
```

Build a Model using Transfer Learning

```
print("Summary of default VGG16 model.\n")

# we are using VGG16 for transfer learnin here. So we have imported it
from tensorflow.keras.applications import VGG16

# initializing model with weights='imagenet'i.e. we are carring its original weights
model_vgg16=VGG16(weights='imagenet')

# display the summary to see the properties of the model
model_vgg16.summary()
```

Customize Model

```
print("Summary of Custom VGG16 model.\n")
print("1) We setup input layer.\n2) We removed top (last) layer. \n")

# let us prepare our input_layer to pass our image size. default is (224,224,3). We will change it to (48,48,3)
input_layer=layers.Input(shape=(48,48,3))

# initialize the transfer model VGG16 with appropriate properties per our need.
# we are passing paramers as following
# 1) weights='imagenet' - Using this we are carring weights as of original weights.
# 2) input_tensor to pass the VGG16 using input_tensor
# 3) we want to change the last layer so we are not including top layer
model_vgg16=VGG16(weights='imagenet',input_tensor=input_layer,include_top=False)

# See the summary of the model with our properties.
model_vgg16.summary()
```

Loading...

Observation:

1. The first layer is having image size = (224,224,3) now as we defined.
2. Also, see the folloiwng 2 top (last) layers which were there in original VGG16 are now not the part of our customized layer because we set include_top=False:

```
# access the current last layer of the model and add flatten and dense after it

print("Summary of Custom VGG16 model.\n")
print("1) We flatten the last layer and added 1 Dense layer and 1 output layer.\n")

last_layer=model_vgg16.output # we are taking last layer of the model

# Add flatten layer: we are extending Neural Network by adding flattn layer
flatten=layers.Flatten()(last_layer)

# Add dense layer
dense1=layers.Dense(100,activation='relu')(flatten)
dense1=layers.Dense(100,activation='relu')(flatten)
dense1=layers.Dense(100,activation='relu')(flatten)


# Add dense layer to the final output layer
output_layer=layers.Dense(10,activation='softmax')(flatten)

# Creating model with input and output layer
model=models.Model(inputs=input_layer,outputs=output_layer,name='CustomVGG16')

# Summarize the model
model.summary()
```

```
# Train only the last layer
print("We are making all the layers intrainable except the last layer. \n")
for layer in model.layers[:-1]:
    layer.trainable=False
model.summary()
```

Train Custom Model

```
from sklearn.model_selection import train_test_split
x_train, x_test, y_train, y_test = train_test_split(x,y,test_size=0.2,random_state=5)
#print(x_train)
#print(x_test)
#print(y_train)
#print(y_test)

print("Splitting data for train and test completed.")
```

```
Splitting data for train and test completed.
```

Compiling and Optimizing Model

```
model.compile(loss='categorical_crossentropy', optimizer='adam',metrics=['accuracy'])

print("Model compilation completed.")
model.summary()
```

```
history = model.fit(x_train,y_train,epochs=20,batch_size=128,verbose=True,validation_data=(x_test,y_test))

print("Fitting the model completed.")
```

Model Evaluation

```
# Function 1

def predict(img_name):
    img=image.load_img(img_name,target_size=(48,48))
    img=image.img_to_array(img)
    plt.imshow(img.astype('int32'))
    plt.show()
    img=preprocess_input(img)

    prediction=model.predict(img.reshape(1,48,48,3))
    output=np.argmax(prediction)

    print(class_names[output] + ": " + Get_Element_Name(class_names[output]))


# Function 2

# This function plots the image supplied in array
def plot_image(i, predictions_array, true_label, img): # taking index and 3 arrays viz. prediction array, true label array and image array

    predictions_array, true_label, img = predictions_array[i], true_label[i], img[i]

    plt.grid(False)
    plt.xticks([])
    plt.yticks([])

    plt.imshow(img.astype('int32'))

    predicted_label=np.argmax(predictions_array)
    true_label=np.argmax(true_label)

    if predicted_label == true_label: #setting up label color
        color='green' # correct then blue colour
    else:
        color='red' # wrong then red colour

    plt.xlabel("{} {:2.0f}% \n ({})".format(Get_Element_Name(predicted_label),
                                    100*np.max(predictions_array), Get_Element_Name(true_label),
                                    color=color, horizontalalignment='left'))
```

```
# Function 3

# This function plots bar chart supplied in the array data
def plot_value_array(i, predictions_array, true_label): # taking index along with predictions and true label array
    predictions_array, true_label = predictions_array[i], true_label[i]
    plt.grid(False)
    plt.xticks([])
    plt.yticks([])
    predicted_label=np.argmax(predictions_array)
    true_label=np.argmax(true_label)

    if predicted_label == 0:
        predicted_label=1
    if true_label == 0:
        true_label=1

    thisplot=plt.bar(range(10), predicted_label, color='seashell')
    plt.ylim([0,1])

    thisplot[predicted_label].set_color('red')
    thisplot[true_label].set_color('green')
```

Predictions

```
# Preparing prediction arrary
predictions=[]

for img in tqdm(val_images):
    img=img.reshape(1,48,48,3)
    predictions.append(model.predict(img))
```

Example Prediction

```
i=random.randrange(1, 10000) # Random input image
plt.figure(figsize=(6,3))
plt.subplot(1,2,1)
# we are passing "val_images_Original" just to show original image instead of "val_images"
# which is preprocessed as VGG16 process and used for prediction.
plot_image(i,predictions, val_image_label, val_images_Original)
plt.subplot(1,2,2)
plot_value_array(i, predictions, val_image_label)
plt.show()
```

Loading...

```
# Declaring variables
num_rows=5
num_cols=5
num_images=num_rows*num_cols

plt.figure(figsize=(2*2*num_cols,2*num_rows))

print("Classification of using Transfer Learning (VGG16)\n")
print("Predicted, Percentage, (Original)\n")

for i in range(num_images):
    plt.subplot(num_rows, 2*num_cols, 2*i+1)
    ii=random.randrange(1,10000)
    # we are passing "val_images_Original" just to show original image instead of "val_images"
    # which is preprocessed as VGG16 process and used for prediction.
    plot_image(ii,predictions, val_image_label, val_images_Original)

    plt.subplot(num_rows, 2*num_cols, 2*i+2)

    plot_value_array(i, predictions, val_image_label)
plt.subplots_adjust(hspace=0.5)
plt.show()
```

Loss and Accuracy

```
acc = history.history['accuracy']
val_acc = history.history['val_accuracy']
loss = history.history['loss']
val_loss = history.history['val_loss']
epochs = range(1, len(acc) + 1)

plt.title('Training and validation accuracy')
plt.plot(epochs, acc, 'red', label='Training acc')
plt.plot(epochs, val_acc, 'blue', label='Validation acc')
plt.legend()

plt.figure()
plt.title('Training and validation loss')
plt.plot(epochs, loss, 'red', label='Training loss')
plt.plot(epochs, val_loss, 'blue', label='Validation loss')

plt.legend()

plt.show()
```

C. Image Conversion

```python
import torchvision.transforms as transforms
from PIL import Image
import numpy as np
import os

# === Configuration ===
img_path = "C:/Users/Acer/Downloads/CNN-FPGA-Implementation-main/miscellaneous/demo_pack/image_data/raw_image.png"  # UPDATE this to your image path
output_path = "C:/Users/Acer/Downloads/CNN-FPGA-Implementation-main/miscellaneous/demo_pack/image_data/img_signed_in.dat"

# === Load and convert image to grayscale ===
img = Image.open(img_path).convert("L")  # 'L' mode = grayscale
img = img.resize((28, 28))  # Ensure it's 28x28 if needed

# === Convert image to numpy array ===
array = np.asarray(img, dtype=np.uint8)

# === Flatten to 1D and normalize to -1.0 to +1.0 ===
flat = array.flatten().astype(np.float32)
norm = (flat / 127.5) - 1.0  # maps 0-255 → -1.0 to +1.0

# === Convert to signed Q0.8 ===
q08_signed = np.clip(np.round(norm * 128), -128, 127).astype(np.int8)


# === Save to .dat file: one signed 8-bit int per line ===
with open(output_path, "w") as f:
    for val in q08_signed:
        f.write(f"{val}\n")

print(f"✅ Success: Signed Q0.8 data saved to {output_path}")

```

```python
import torchvision.transforms as transforms
from PIL import Image
import numpy as np
import os

# === Configuration ===
img_path = "C:/Users/Acer/Downloads/CNN-FPGA-Implementation-main/miscellaneous/demo_pack/image_data/raw_image.png"  # UPDATE this to your image path
output_path = "C:/Users/Acer/Downloads/CNN-FPGA-Implementation-main/miscellaneous/demo_pack/image_data/img_signed_32bit_in.dat"

# === Load and convert image to grayscale ===
img = Image.open(img_path).convert("L")  # 'L' mode = grayscale
img = img.resize((28, 28))  # Ensure it's 28x28 if needed

# === Convert image to numpy array ===
array = np.asarray(img, dtype=np.uint8)

# === Flatten and normalize to [-1.0, 1.0] ===
flat = array.flatten().astype(np.float32)
norm = (flat / 127.5) - 1.0

# === Convert to signed Q16.16 ===
scaled = np.round(norm * 65536).astype(np.int64)  # use int64 to avoid intermediate overflow
q16_16_signed = np.clip(scaled, -2147483648, 2147483647).astype(np.int32)

# === Save to .dat file: one 32-bit signed int per line ===
with open(output_path, "w") as f:
    for val in q16_16_signed:
        f.write(f"{int(val)}\n")  # write as normal int

print(f"✅ 32-bit Q16.16 .dat saved to: {output_path}")

```

D. Intel HEX File Generation

```python
import numpy as np

def float_to_q8_8(val):
    """Convert a float (-128 to 127.996) to Q8.8 fixed-point (signed 16-bit int)."""
    scaled = int(round(val * 256))  # 2^8 = 256
    if scaled < -32768 or scaled > 32767:
        raise OverflowError(f"Value {val} out of range after scaling.")
    return scaled & 0xFFFF  # ensure it's 16-bit

def generate_hex_file(txt_path, hex_path):
    with open(txt_path, 'r') as f:
        lines = f.readlines()

    hex_lines = []
    for line in lines:
        try:
            val = float(line.strip())
            fixed_val = float_to_q8_8(val)
            hex_lines.append(f"{fixed_val:04X}\n")  # 4-digit hex, uppercase
        except ValueError:
            print(f"Skipping invalid line: {line.strip()}")
        except OverflowError as e:
            print(f"Error: {e}")

    with open(hex_path, 'w') as f:
        f.writelines(hex_lines)

    print(f"✅ HEX file written: {hex_path} ({len(hex_lines)} values)")

generate_hex_file("C:/Users/Acer/Downloads/CNN-FPGA-Implementation-main/miscellaneous/model/wb16/conv1_weight.txt", "C:/Users/Acer/Downloads/CNN-FPGA-Implementation-main/src/IntelHEX/conv1_weight.hex")
generate_hex_file("C:/Users/Acer/Downloads/CNN-FPGA-Implementation-main/miscellaneous/model/wb16/conv1_bias.txt", "C:/Users/Acer/Downloads/CNN-FPGA-Implementation-main/src/IntelHEX/conv1_bias.hex")
generate_hex_file("C:/Users/Acer/Downloads/CNN-FPGA-Implementation-main/miscellaneous/model/wb16/conv2_weight.txt", "C:/Users/Acer/Downloads/CNN-FPGA-Implementation-main/src/IntelHEX/conv2_weight.hex")
generate_hex_file("C:/Users/Acer/Downloads/CNN-FPGA-Implementation-main/miscellaneous/model/wb16/conv2_bias.txt", "C:/Users/Acer/Downloads/CNN-FPGA-Implementation-main/src/IntelHEX/conv2_bias.hex")
generate_hex_file("C:/Users/Acer/Downloads/CNN-FPGA-Implementation-main/miscellaneous/model/wb16/dense_weight.txt", "C:/Users/Acer/Downloads/CNN-FPGA-Implementation-main/src/IntelHEX/dense_weight.hex")
generate_hex_file("C:/Users/Acer/Downloads/CNN-FPGA-Implementation-main/miscellaneous/model/wb16/dense_bias.txt", "C:/Users/Acer/Downloads/CNN-FPGA-Implementation-main/src/IntelHEX/dense_bias.hex")
```

```python
def dat_to_hex_signed_xxxx(dat_path, hex_path):
    with open(dat_path, 'r') as f:
        lines = f.readlines()

    hex_lines = []
    for line in lines:
        try:
            val = int(line.strip())
            if not -128 <= val <= 127:
                raise ValueError(f"Value {val} is not in signed 8-bit range (-128 to 127)")

            signed_word = val & 0xFF  # keep lower 8 bits
            full_word = signed_word  # lower byte, upper byte = 0
            hex_lines.append(f"{full_word:04X}\n")
        except ValueError:
            print(f"Skipping invalid line: {line.strip()}")

    with open(hex_path, 'w') as f:
        f.writelines(hex_lines)

    print(f"✅ HEX file saved in signed xxxx format: {hex_path} ({len(hex_lines)} entries)")

# === Example usage ===
dat_to_hex_signed_xxxx(
    "C:/Users/Acer/Downloads/CNN-FPGA-Implementation-main/miscellaneous/demo_pack/image_data/img_signed_in.dat",
    "C:/Users/Acer/Downloads/CNN-FPGA-Implementation-main/src/IntelHEX/img_signed_in.hex"
)
```

```python
import numpy as np

def float_to_q16_16(val):
    """Convert a float (-128 to 127.996) to Q16.16 fixed-point (signed 16-bit int)."""
    scaled = int(round(val * 65536))  # 2^16 = 256
    if scaled < -2147483648 or scaled > 2147483647:
        raise OverflowError(f"Value {val} out of range after scaling.")
    return scaled & 0xFFFFFFFF


def generate_hex_file(txt_path, hex_path):
    with open(txt_path, 'r') as f:
        lines = f.readlines()

    hex_lines = []
    for line in lines:
        try:
            val = float(line.strip())
            fixed_val = float_to_q16_16(val)
            hex_lines.append(f"{fixed_val:04X}\n")  # 4-digit hex, uppercase
        except ValueError:
            print(f"Skipping invalid line: {line.strip()}")
        except OverflowError as e:
            print(f"Error: {e}")

    with open(hex_path, 'w') as f:
        f.writelines(hex_lines)

    print(f"✅ HEX file written: {hex_path} ({len(hex_lines)} values)")

generate_hex_file("C:/Users/Acer/Downloads/CNN-FPGA-Implementation-main/miscellaneous/model/wb32/conv1_weight_32.txt", "C:/Users/Acer/Downloads/CNN-FPGA-Implementation-main/src/IntelHEX/wb32/conv1_weight_32.hex")
generate_hex_file("C:/Users/Acer/Downloads/CNN-FPGA-Implementation-main/miscellaneous/model/wb32/conv1_bias_32.txt", "C:/Users/Acer/Downloads/CNN-FPGA-Implementation-main/src/IntelHEX/wb32/conv1_bias_32.hex")
generate_hex_file("C:/Users/Acer/Downloads/CNN-FPGA-Implementation-main/miscellaneous/model/wb32/conv2_weight_32.txt", "C:/Users/Acer/Downloads/CNN-FPGA-Implementation-main/src/IntelHEX/wb32/conv2_weight_32.hex")
generate_hex_file("C:/Users/Acer/Downloads/CNN-FPGA-Implementation-main/miscellaneous/model/wb32/conv2_bias_32.txt", "C:/Users/Acer/Downloads/CNN-FPGA-Implementation-main/src/IntelHEX/wb32/conv2_bias_32.hex")
generate_hex_file("C:/Users/Acer/Downloads/CNN-FPGA-Implementation-main/miscellaneous/model/wb32/dense_weight_32.txt", "C:/Users/Acer/Downloads/CNN-FPGA-Implementation-main/src/IntelHEX/wb32/dense_weight_32.hex")
generate_hex_file("C:/Users/Acer/Downloads/CNN-FPGA-Implementation-main/miscellaneous/model/wb32/dense_bias_32.txt", "C:/Users/Acer/Downloads/CNN-FPGA-Implementation-main/src/IntelHEX/wb32/dense_bias_32.hex")
```

```python
import numpy as np

def dat_to_hex_signed_32bit(dat_path, hex_path):
    hex_lines = []
    with open(dat_path, 'r') as f:
        for line in f:
            try:
                val = int(line.strip())
                if val < -2147483648 or val > 2147483647:
                    raise OverflowError(f"Out of 32-bit range: {val}")
                val &= 0xFFFFFFFF  # force unsigned 32-bit representation
                hex_lines.append(f"{val:08X}\n")
            except Exception as e:
                print(f"⚠️ Skipping line: {line.strip()} — {e}")

    with open(hex_path, 'w') as f:
        f.writelines(hex_lines)

    print(f"✅ 32-bit HEX file written: {hex_path} ({len(hex_lines)} entries)")



# === Example usage ===
dat_to_hex_signed_32bit(
    "C:/Users/Acer/Downloads/CNN-FPGA-Implementation-main/miscellaneous/demo_pack/image_data/img_signed_32bit_in.dat",
    "C:/Users/Acer/Downloads/CNN-FPGA-Implementation-main/src/IntelHEX/wb32/img_signed_32bit_in.hex"
)
```

E. First Convolutional Layer Module

```
`timescale 1ns / 1ps

module conv_layer_1 #(
    parameter IN_CHANNELS = 1,
    parameter OUT_CHANNELS = 2,
    parameter IN_IMG_SIZE = 28,
    parameter OUT_IMG_SIZE = 24,
    parameter KERNEL_SIZE = 5,
    parameter DATA_WIDTH = 16,
    parameter SUM_WIDTH = DATA_WIDTH * 2 + 4
)(
    input wire clk,
    input wire reset_n,
    input wire start,
    input wire signed [DATA_WIDTH*IN_IMG_SIZE*IN_IMG_SIZE*IN_CHANNELS-1:0] image_in_linear,
    input wire signed [DATA_WIDTH*KERNEL_SIZE*KERNEL_SIZE*IN_CHANNELS*OUT_CHANNELS-1:0] weights_linear,
    input wire signed [DATA_WIDTH*OUT_CHANNELS-1:0] biases_linear,
    output reg finish,
    output reg signed [DATA_WIDTH*OUT_IMG_SIZE*OUT_IMG_SIZE*OUT_CHANNELS-1:0] map
);

    reg signed [DATA_WIDTH-1:0] image_ram    [0 : IN_CHANNELS*IN_IMG_SIZE*IN_IMG_SIZE-1];
    reg signed [DATA_WIDTH-1:0] weights_ram  [0 : OUT_CHANNELS*IN_CHANNELS*KERNEL_SIZE*KERNEL_SIZE-1];
    reg signed [DATA_WIDTH-1:0] biases_ram   [0 : OUT_CHANNELS-1];
    reg signed [DATA_WIDTH-1:0] feat_map     [0 : OUT_CHANNELS*OUT_IMG_SIZE*OUT_IMG_SIZE-1];

    reg signed [SUM_WIDTH-1:0] sum;
    reg signed [DATA_WIDTH*2-1:0] product;

    integer row, col, channel, filter, ker_row, ker_col;
    integer pixel_i, weight_i, row_offset, col_offset;

    reg process;
    reg start_0;
    wire run = start && !start_0;

    // Load input into RAMs
    integer loadram_i;
    always @(posedge clk or negedge reset_n) begin
        for (loadram_i = 0; loadram_i < IN_CHANNELS*IN_IMG_SIZE*IN_IMG_SIZE; loadram_i = loadram_i + 1)
            image_ram[loadram_i] = image_in_linear[(loadram_i+1)*DATA_WIDTH-1 -: DATA_WIDTH];

        for (loadram_i = 0; loadram_i < OUT_CHANNELS*IN_CHANNELS*KERNEL_SIZE*KERNEL_SIZE; loadram_i = loadram_i + 1)
            weights_ram[loadram_i] = weights_linear[(loadram_i+1)*DATA_WIDTH-1 -: DATA_WIDTH];

        for (loadram_i = 0; loadram_i < OUT_CHANNELS; loadram_i = loadram_i + 1)
            biases_ram[loadram_i] = biases_linear[(loadram_i+1)*DATA_WIDTH-1 -: DATA_WIDTH];
    end
```

```
36
37      // Load input into RAMs
38      integer loadram_i;
39      always @(posedge clk or negedge reset_n) begin
40          for (loadram_i = 0; loadram_i < IN_CHANNELS*IN_IMG_SIZE*IN_IMG_SIZE; loadram_i = loadram_i + 1)
41              image_ram[loadram_i] = image_in_linear[(loadram_i+1)*DATA_WIDTH-1 -: DATA_WIDTH];
42
43          for (loadram_i = 0; loadram_i < OUT_CHANNELS*IN_CHANNELS*KERNEL_SIZE*KERNEL_SIZE; loadram_i = loadram_i + 1)
44              weights_ram[loadram_i] = weights_linear[(loadram_i+1)*DATA_WIDTH-1 -: DATA_WIDTH];
45
46          for (loadram_i = 0; loadram_i < OUT_CHANNELS; loadram_i = loadram_i + 1)
47              biases_ram[loadram_i] = biases_linear[(loadram_i+1)*DATA_WIDTH-1 -: DATA_WIDTH];
48      end
49
50      always @(posedge clk or negedge reset_n) begin
51          if (!reset_n) begin
52              process <= 0;
53              finish <= 0;
54              row <= 0;
55              col <= 0;
56              filter <= 0;
57              start_0 <= 0;
58          end else begin
59              start_0 <= start;
60
61              if (run && !process) begin
62                  process <= 1;
63                  finish <= 0;
64                  row <= 0;
65                  col <= 0;
66                  filter <= 0;
67
68              end else if (process) begin
69                  sum = 0;
70                  // Convolutional algorithm
71                  for (channel = 0; channel < IN_CHANNELS; channel = channel + 1)
72                      for (ker_row = 0; ker_row < KERNEL_SIZE; ker_row = ker_row + 1)
73                          for (ker_col = 0; ker_col < KERNEL_SIZE; ker_col = ker_col + 1) begin
74                              row_offset = row + ker_row - KERNEL_SIZE / 2;
75                              col_offset = col + ker_col - KERNEL_SIZE / 2;
76
77                              if (row_offset >= 0 && row_offset < IN_IMG_SIZE && col_offset >= 0 && col_offset < IN_IMG_SIZE) begin
78                                  pixel_i = channel * IN_IMG_SIZE * IN_IMG_SIZE + row_offset * IN_IMG_SIZE + col_offset;
79                                  weight_i = filter * (IN_CHANNELS * KERNEL_SIZE * KERNEL_SIZE) + channel * (KERNEL_SIZE * KERNEL_SIZE) + ker_row * KERNEL_SIZE + ker_col;
80                                  product = image_ram[pixel_i] * weights_ram[weight_i];
81                                  sum = sum + product;
82                              end
83                          end
84
85                  // ReLU acitivate function
86                  sum = sum + biases_ram[filter];
87                  if (sum < 0) sum = 0;
88                  feat_map[filter * OUT_IMG_SIZE * OUT_IMG_SIZE + row * OUT_IMG_SIZE + col] <= sum[DATA_WIDTH-1:0];
89
90                  // Check if last filter
91                  if (filter < OUT_CHANNELS - 1) filter = filter + 1;
92                  // Check if finish
93                  else begin
94                      filter = 0;
95                      if (col < OUT_IMG_SIZE - 1) col = col + 1;
96                      else begin
97                          col = 0;
98                          if (row < OUT_IMG_SIZE - 1) row = row + 1;
99                          else begin
00                              row <= 0;
01                              process <= 0;
02                              finish <= 1;
03                          end
04                      end
05                  end
06              end else begin
07                  finish <= 0;
08              end
09          end
10      end
```

```
112     // Load calculated data to output
113     always @(*) begin
114         for (loadram_i = 0; loadram_i < OUT_CHANNELS*OUT_IMG_SIZE*OUT_IMG_SIZE; loadram_i = loadram_i + 1)
115             map[(loadram_i+1)*DATA_WIDTH-1 -: DATA_WIDTH] = feat_map[loadram_i];
116     end
117
118 endmodule
119
```

F. First Convolutional Layer Testbench Module

```
`timescale 1ns / 1ps

module tb_conv1_layer;
    parameter IN_CHANNELS = 1;
    parameter OUT_CHANNELS = 2;
    parameter IN_IMG_SIZE = 28;
    parameter OUT_IMG_SIZE = 24;
    parameter KERNEL_SIZE = 5;
    parameter DATA_WIDTH = 16;
    parameter SUM_WIDTH = DATA_WIDTH * 2 + 4;

    reg clk = 0;
    reg reset_n = 0;
    reg start = 0;

    // Linear inputs
    reg signed [DATA_WIDTH*IN_IMG_SIZE*IN_IMG_SIZE*IN_CHANNELS-1:0] image_in_linear;
    reg signed [DATA_WIDTH*KERNEL_SIZE*KERNEL_SIZE*IN_CHANNELS*OUT_CHANNELS-1:0] weights_linear;
    reg signed [DATA_WIDTH*OUT_CHANNELS-1:0] biases_linear;
    wire finish;
    wire signed [DATA_WIDTH*OUT_IMG_SIZE*OUT_IMG_SIZE*OUT_CHANNELS-1:0] map;

    // RAMs Instantiate
    reg signed [DATA_WIDTH-1:0] image_ram [0:IN_CHANNELS*IN_IMG_SIZE*IN_IMG_SIZE-1];
    reg signed [DATA_WIDTH-1:0] weights_ram [0:OUT_CHANNELS*IN_CHANNELS*KERNEL_SIZE*KERNEL_SIZE-1];
    reg signed [DATA_WIDTH-1:0] biases_ram [0:OUT_CHANNELS-1];

    integer i;

    // Clock generation
    always #5 clk = ~clk;

    // Instantiate the conv_layer module
    conv_layer_1 #(
    .IN_CHANNELS(IN_CHANNELS),
    .OUT_CHANNELS(OUT_CHANNELS),
    .IN_IMG_SIZE(IN_IMG_SIZE),
    .OUT_IMG_SIZE(OUT_IMG_SIZE),
    .KERNEL_SIZE(KERNEL_SIZE),
    .DATA_WIDTH(DATA_WIDTH),
    .SUM_WIDTH(SUM_WIDTH)

) conv1(
    .clk(clk),
    .reset_n(reset_n),
    .start(start),
    .image_in_linear(image_in_linear),
    .weights_linear(weights_linear),
    .biases_linear(biases_linear),
    .finish(finish),
    .map(map)
);
    initial begin
        // Reset
        reset_n = 0;
        #10 reset_n = 1;

        // Load image, weights, biases
        $readmemh("C:/Users/Acer/Downloads/CNN-FPGA-Implementation-main/src/IntelHEX/mnist_in.hex", image_ram);
        $readmemh("C:/Users/Acer/Downloads/CNN-FPGA-Implementation-main/src/IntelHEX/conv1_weight.hex", weights_ram);
        $readmemh("C:/Users/Acer/Downloads/CNN-FPGA-Implementation-main/src/IntelHEX/conv1_bias.hex", biases_ram);

        // Pack into buses
        for (i = 0; i < IN_IMG_SIZE * IN_IMG_SIZE * IN_CHANNELS; i = i + 1)
            image_in_linear[i*DATA_WIDTH +: DATA_WIDTH] = image_ram[i];

        for (i = 0; i < KERNEL_SIZE * KERNEL_SIZE * IN_CHANNELS * OUT_CHANNELS; i = i + 1)
            weights_linear[i*DATA_WIDTH +: DATA_WIDTH] = weights_ram[i];

        for (i = 0; i < OUT_CHANNELS; i = i + 1)
            biases_linear[i*DATA_WIDTH +: DATA_WIDTH] = biases_ram[i];

        // Start convolution
        #20 start = 1;
        #10 start = 0;

        // Wait for conv_done
        wait (finish);
        $display("First Convolutional Layer done.");

    end

endmodule
```

G. First Max Pooling and ReLU Layer Module

```
`timescale 1ns / 1ps

module maxpool_layer_1 #(
    parameter IMG_WIDTH = 24,
    parameter IMG_HEIGHT = 24,
    parameter CHANNELS = 2,
    parameter POOL_SIZE = 2,
    parameter DATA_WIDTH = 16
)(
    input wire clk,
    input wire reset_n,
    input wire start,
    input wire signed [IMG_WIDTH*IMG_HEIGHT*CHANNELS*DATA_WIDTH-1:0] data_in,
    output reg finish,
    output reg signed [(IMG_WIDTH/2)*(IMG_HEIGHT/2)*CHANNELS*DATA_WIDTH-1:0] data_out
);

    // === Internal 2D arrays ===
    reg [DATA_WIDTH-1:0] input_data    [0:CHANNELS-1][0:IMG_WIDTH*IMG_HEIGHT-1];
    reg [DATA_WIDTH-1:0] pooled_data   [0:CHANNELS-1][0:(IMG_WIDTH/2)*(IMG_HEIGHT/2)-1];

    integer loadram_i, channel, row, col, index, pool_index;
    reg [DATA_WIDTH-1:0] max;

    // === MaxPooling Logic ===
    always @(posedge clk) begin
        // 1. Load input to RAMs
        for (channel = 0; channel < CHANNELS; channel = channel + 1) begin
            for (loadram_i = 0; loadram_i < IMG_WIDTH * IMG_HEIGHT; loadram_i = loadram_i + 1)
                input_data[channel][loadram_i] = data_in[(channel*IMG_WIDTH*IMG_HEIGHT + loadram_i)*DATA_WIDTH +: DATA_WIDTH];
        end

        // 2. Max pooling each channel
        for (channel = 0; channel < CHANNELS; channel = channel + 1) begin
            pool_index = 0;
            for (row = 0; row < IMG_HEIGHT; row = row + 2) begin
                for (col = 0; col < IMG_WIDTH; col = col + 2) begin
                    index = row * IMG_WIDTH + col;
                    max = input_data[channel][index];
                    if (input_data[channel][index + 1] > max) max = input_data[channel][index + 1];
                    if (input_data[channel][index + IMG_WIDTH] > max) max = input_data[channel][index + IMG_WIDTH];
                    if (input_data[channel][index + IMG_WIDTH + 1] > max) max = input_data[channel][index + IMG_WIDTH + 1];
                    pooled_data[channel][pool_index] = max;
                    pool_index = pool_index + 1;
                end
            end
        end

        // 3. Extract from 2D RAM to 1D output
        for (channel = 0; channel < CHANNELS; channel = channel + 1) begin
            for (loadram_i = 0; loadram_i < (IMG_WIDTH/2)*(IMG_HEIGHT/2)-1; loadram_i = loadram_i + 1)
                data_out[(channel*(IMG_WIDTH/2)*(IMG_HEIGHT/2) + loadram_i)*DATA_WIDTH +: DATA_WIDTH] = pooled_data[channel][loadram_i];
        end
    end
    // === Finish flag ===
    always @(posedge clk or negedge reset_n) begin
        if (!reset_n) finish <= 0;
        else if (start) finish <= 1;
    end

endmodule

```

H. First Max Pooling and ReLU Layer Testbench Module

```
`timescale 1ns / 1ps

module tb_maxpool1_layer;
    parameter IMG_WIDTH = 24;
    parameter IMG_HEIGHT = 24;
    parameter CHANNELS = 2;
    parameter POOL_SIZE = 2;
    parameter DATA_WIDTH = 16;

    reg clk = 0;
    reg reset_n = 0;
    reg start = 0;
    reg signed [IMG_WIDTH*IMG_HEIGHT*CHANNELS*DATA_WIDTH-1:0] data_in;
    wire signed [(IMG_WIDTH/2)*(IMG_HEIGHT/2)*CHANNELS*DATA_WIDTH-1:0] data_out;
    wire finish;

    integer i;

    // Clock generation
    always #5 clk = ~clk;

    // DUT
    maxpool_layer_1 maxrelu1 (
        .clk(clk),
        .reset_n(reset_n),
        .start(start),
        .data_in(data_in),
        .finish(finish),
        .data_out(data_out)
    );

    initial begin
        // Reset
        reset_n = 0;
        #10 reset_n = 1;

        data_in[15:0] = 13;

        // Start
        #10 start = 1;
        #1000 start = 0;

        // Wait for finish
        wait (finish);

    end

endmodule
```

I. First Convolutional Block Module

```
`timescale 1ns / 1ps

module conv_block1 #(
    parameter IN_IMG_SIZE = 28,
    parameter OUT_IMG_SIZE = 24,
    parameter KERNEL_SIZE = 5,
    parameter IN_CHANNELS = 1,
    parameter OUT_CHANNELS = 2,
    parameter DATA_WIDTH = 16
)(
    input wire clk,
    input wire reset_n,
    input wire start,

    input wire signed [DATA_WIDTH*IN_IMG_SIZE*IN_IMG_SIZE*IN_CHANNELS-1:0] image_in_linear,
    input wire signed [DATA_WIDTH*KERNEL_SIZE*KERNEL_SIZE*IN_CHANNELS*OUT_CHANNELS-1:0] weights_linear,
    input wire signed [DATA_WIDTH*OUT_CHANNELS-1:0] biases_linear,

    output wire finish_block,
    output wire signed [DATA_WIDTH*(OUT_IMG_SIZE/2)*(OUT_IMG_SIZE/2)*OUT_CHANNELS-1:0] block_out_linear
);

    wire finish;
    wire signed [DATA_WIDTH*OUT_IMG_SIZE*OUT_IMG_SIZE*OUT_CHANNELS-1:0] conv_out_linear;

    // === First Convolution Layer ===
    conv_layer_1 #(
        .IN_IMG_SIZE(IN_IMG_SIZE),
        .OUT_IMG_SIZE(OUT_IMG_SIZE),
        .KERNEL_SIZE(KERNEL_SIZE),
        .IN_CHANNELS(IN_CHANNELS),
        .OUT_CHANNELS(OUT_CHANNELS),
        .DATA_WIDTH(DATA_WIDTH)
    ) conv1 (
        .clk(clk),
        .reset_n(reset_n),
        .start(start),
        .image_in_linear(image_in_linear),
        .weights_linear(weights_linear),
        .biases_linear(biases_linear),
        .finish(finish),
        .map(conv_out_linear)
    );

    // === First MaxPool Layer ===
    maxpool_layer_1 #(
      .IMG_WIDTH(OUT_IMG_SIZE),
      .IMG_HEIGHT(OUT_IMG_SIZE),
        .POOL_SIZE(OUT_CHANNELS),
        .CHANNELS(OUT_CHANNELS),
        .DATA_WIDTH(DATA_WIDTH)
    ) pool1 (
        .clk(clk),
        .reset_n(reset_n),
        .start(finish),  // trigger pooling when conv finishes
        .data_in(conv_out_linear),
        .finish(finish_block),
        .data_out(block_out_linear)
    );

endmodule
```

J. First Convolutional Block Testbench Module

```
`timescale 1ns / 1ps

module tb_convl_block;
    parameter IN_IMG_SIZE = 28;
    parameter OUT_IMG_SIZE = 24;
    parameter KERNEL_SIZE = 5;
    parameter IN_CHANNELS = 1;
    parameter OUT_CHANNELS = 2;
    parameter DATA_WIDTH = 16;

    reg clk = 0;
    reg reset_n = 0;
    reg start = 0;

    // Linear inputs
    reg signed [DATA_WIDTH*IN_IMG_SIZE*IN_IMG_SIZE*IN_CHANNELS-1:0] image_in_linear;
    reg signed [DATA_WIDTH*KERNEL_SIZE*KERNEL_SIZE*IN_CHANNELS*OUT_CHANNELS-1:0] weights_linear;
    reg signed [DATA_WIDTH*OUT_CHANNELS-1:0] biases_linear;
    wire finish_block;
    wire signed [DATA_WIDTH*OUT_IMG_SIZE*OUT_IMG_SIZE*OUT_CHANNELS-1:0] block_out_linear;

    // RAMs Instantiate
    reg signed [DATA_WIDTH-1:0] image_ram [0:IN_CHANNELS*IN_IMG_SIZE*IN_IMG_SIZE-1];
    reg signed [DATA_WIDTH-1:0] weights_ram [0:OUT_CHANNELS*IN_CHANNELS*KERNEL_SIZE*KERNEL_SIZE-1];
    reg signed [DATA_WIDTH-1:0] biases_ram [0:OUT_CHANNELS-1];

    integer i;

    // Clock generation
    always #5 clk = ~clk;

    // Instantiate the conv_layer module
    conv_block1 #(
    .IN_IMG_SIZE(IN_IMG_SIZE),
    .OUT_IMG_SIZE(OUT_IMG_SIZE),
    .KERNEL_SIZE(KERNEL_SIZE),
    .IN_CHANNELS(IN_CHANNELS),
    .OUT_CHANNELS(OUT_CHANNELS),
    .DATA_WIDTH(DATA_WIDTH)

) convl_block(
    .clk(clk),
    .reset_n(reset_n),
    .start(start),
    .image_in_linear(image_in_linear),
    .weights_linear(weights_linear),
    .biases_linear(biases_linear),
    .finish_block(finish_block),
    .block_out_linear(block_out_linear)
);
    initial begin
        // Reset
        reset_n = 0;
        #10 reset_n = 1;

        // Load image, weights, biases
        $readmemh("C:/Users/Acer/Downloads/CNN-FPGA-Implementation-main/src/IntelHEX/mnist_in.hex", image_ram);
        $readmemh("C:/Users/Acer/Downloads/CNN-FPGA-Implementation-main/src/IntelHEX/convl_weight.hex", weights_ram);
        $readmemh("C:/Users/Acer/Downloads/CNN-FPGA-Implementation-main/src/IntelHEX/convl_bias.hex", biases_ram);

        // Pack into buses
        for (i = 0; i < IN_IMG_SIZE * IN_IMG_SIZE * IN_CHANNELS; i = i + 1)
            image_in_linear[i*DATA_WIDTH +: DATA_WIDTH] = image_ram[i];

        for (i = 0; i < KERNEL_SIZE * KERNEL_SIZE * IN_CHANNELS * OUT_CHANNELS; i = i + 1)
            weights_linear[i*DATA_WIDTH +: DATA_WIDTH] = weights_ram[i];

        for (i = 0; i < OUT_CHANNELS; i = i + 1)
            biases_linear[i*DATA_WIDTH +: DATA_WIDTH] = biases_ram[i];

        // Start convolution
        #20 start = 1;
        #10 start = 0;

        // Wait for conv_done
        wait (finish_block);
        $display("First Convolutional Block done.");

    end
endmodule
```

K. Second Convolutional Layer Module

```
`timescale 1ns / 1ps

module conv_layer_2 #(
    parameter IN_CHANNELS = 2,
    parameter OUT_CHANNELS = 3,
    parameter IN_IMG_SIZE = 12,
    parameter OUT_IMG_SIZE = 10,
    parameter KERNEL_SIZE = 3,
    parameter DATA_WIDTH = 16,
    parameter SUM_WIDTH = DATA_WIDTH * 2 + 4
)(
    input wire clk,
    input wire reset_n,
    input wire start,
    input wire signed [DATA_WIDTH*IN_IMG_SIZE*IN_IMG_SIZE*IN_CHANNELS-1:0] image_in_linear,
    input wire signed [DATA_WIDTH*KERNEL_SIZE*KERNEL_SIZE*IN_CHANNELS*OUT_CHANNELS-1:0] weights_linear,
    input wire signed [DATA_WIDTH*OUT_CHANNELS-1:0] biases_linear,
    output reg finish,
    output reg signed [DATA_WIDTH*OUT_IMG_SIZE*OUT_IMG_SIZE*OUT_CHANNELS-1:0] map
);

    reg signed [DATA_WIDTH-1:0] image_ram    [0 : IN_CHANNELS*IN_IMG_SIZE*IN_IMG_SIZE-1];
    reg signed [DATA_WIDTH-1:0] weights_ram  [0 : OUT_CHANNELS*IN_CHANNELS*KERNEL_SIZE*KERNEL_SIZE-1];
    reg signed [DATA_WIDTH-1:0] biases_ram   [0 : OUT_CHANNELS-1];
    reg signed [DATA_WIDTH-1:0] feat_map     [0 : OUT_CHANNELS*OUT_IMG_SIZE*OUT_IMG_SIZE-1];

    reg signed [SUM_WIDTH-1:0] sum;
    reg signed [DATA_WIDTH*2-1:0] product;

    integer row, col, channel, filter, ker_row, ker_col;
    integer pixel_i, weight_i, row_offset, col_offset;

    reg process;
    reg start_0;
    wire run = start && !start_0;

    // Load input into RAMs
    integer loadram_i;
    always @(posedge clk or negedge reset_n) begin
        for (loadram_i = 0; loadram_i < IN_CHANNELS*IN_IMG_SIZE*IN_IMG_SIZE; loadram_i = loadram_i + 1)
            image_ram[loadram_i] = image_in_linear[(loadram_i+1)*DATA_WIDTH-1 -: DATA_WIDTH];

        for (loadram_i = 0; loadram_i < OUT_CHANNELS*IN_CHANNELS*KERNEL_SIZE*KERNEL_SIZE; loadram_i = loadram_i + 1)
            weights_ram[loadram_i] = weights_linear[(loadram_i+1)*DATA_WIDTH-1 -: DATA_WIDTH];

        for (loadram_i = 0; loadram_i < OUT_CHANNELS; loadram_i = loadram_i + 1)
            biases_ram[loadram_i] = biases_linear[(loadram_i+1)*DATA_WIDTH-1 -: DATA_WIDTH];
    end
```

```
    always @(posedge clk or negedge reset_n) begin
        if (!reset_n) begin
            process <= 0;
            finish <= 0;
            row <= 0;
            col <= 0;
            filter <= 0;
            start_0 <= 0;
        end else begin
            start_0 <= start;

            if (run && !process) begin
                process <= 1;
                finish <= 0;
                row <= 0;
                col <= 0;
                filter <= 0;

            end else if (process) begin
                sum = 0;
                // Convolutional algorithm
                for (channel = 0; channel < IN_CHANNELS; channel = channel + 1)
                    for (ker_row = 0; ker_row < KERNEL_SIZE; ker_row = ker_row + 1)
                        for (ker_col = 0; ker_col < KERNEL_SIZE; ker_col = ker_col + 1) begin
                            row_offset = row + ker_row - KERNEL_SIZE / 2;
                            col_offset = col + ker_col - KERNEL_SIZE / 2;

                            if (row_offset >= 0 && row_offset < IN_IMG_SIZE && col_offset >= 0 && col_offset < IN_IMG_SIZE) begin
                                pixel_i = channel * IN_IMG_SIZE * IN_IMG_SIZE + row_offset * IN_IMG_SIZE + col_offset;
                                weight_i = filter * (IN_CHANNELS * KERNEL_SIZE * KERNEL_SIZE) + channel * (KERNEL_SIZE * KERNEL_SIZE) + ker_row * KERNEL_SIZE + ker_col;
                                product = image_ram[pixel_i] * weights_ram[weight_i];
                                sum = sum + product;
                            end
                        end

                // Max Pooling
                sum = sum + biases_ram[filter];
                if (sum < 0) sum = 0;
                feat_map[filter * OUT_IMG_SIZE * OUT_IMG_SIZE + row * OUT_IMG_SIZE + col] <= sum[DATA_WIDTH-1:0];

                // Check if last filter
                if (filter < OUT_CHANNELS - 1) filter = filter + 1;
                // Check if finish
                else begin
                    filter = 0;
                    if (col < OUT_IMG_SIZE - 1) col = col + 1;
                    else begin
                        col = 0;
                        if (row < OUT_IMG_SIZE - 1) row = row + 1;
                        else begin
                            row <= 0;
                            process <= 0;
                            finish <= 1;
                        end
                    end
                end
            end else begin
                finish <= 0;
            end
        end
    end

    // Load calculated data to output
    always @(*) begin
        for (loadram_i = 0; loadram_i < OUT_CHANNELS*OUT_IMG_SIZE*OUT_IMG_SIZE; loadram_i = loadram_i + 1)
            map[(loadram_i+1)*DATA_WIDTH-1 -: DATA_WIDTH] = feat_map[loadram_i];
    end

endmodule
```

L. Second Convolutional Layer Testbench Module

```
`timescale 1ns / 1ps

module tb_conv2_layer;
    parameter IN_CHANNELS = 1;
    parameter OUT_CHANNELS = 2;
    parameter IN_IMG_SIZE = 28;
    parameter OUT_IMG_SIZE = 24;
    parameter KERNEL_SIZE = 5;
    parameter DATA_WIDTH = 16;
    parameter SUM_WIDTH = DATA_WIDTH * 2 + 4;

    reg clk = 0;
    reg reset_n = 0;
    reg start = 0;

    // Linear inputs
    reg signed [DATA_WIDTH*IN_IMG_SIZE*IN_IMG_SIZE*IN_CHANNELS-1:0] image_in_linear;
    reg signed [DATA_WIDTH*KERNEL_SIZE*KERNEL_SIZE*IN_CHANNELS*OUT_CHANNELS-1:0] weights_linear;
    reg signed [DATA_WIDTH*OUT_CHANNELS-1:0] biases_linear;
    wire finish;
    wire signed [DATA_WIDTH*OUT_IMG_SIZE*OUT_IMG_SIZE*OUT_CHANNELS-1:0] map;

    // RAMs Instantiate
    reg signed [DATA_WIDTH-1:0] image_ram [0:IN_CHANNELS*IN_IMG_SIZE*IN_IMG_SIZE-1];
    reg signed [DATA_WIDTH-1:0] weights_ram [0:OUT_CHANNELS*IN_CHANNELS*KERNEL_SIZE*KERNEL_SIZE-1];
    reg signed [DATA_WIDTH-1:0] biases_ram [0:OUT_CHANNELS-1];
    reg signed [DATA_WIDTH-1:0] feat_map [0:OUT_CHANNELS*OUT_IMG_SIZE*OUT_IMG_SIZE-1];

    integer i;

    // Clock generation
    always #5 clk = ~clk;

    // Instantiate the conv_layer module
    conv_layer_2 #(
    .IN_CHANNELS(IN_CHANNELS),
    .OUT_CHANNELS(OUT_CHANNELS),
    .IN_IMG_SIZE(IN_IMG_SIZE),
    .OUT_IMG_SIZE(OUT_IMG_SIZE),
    .KERNEL_SIZE(KERNEL_SIZE),
    .DATA_WIDTH(DATA_WIDTH),
    .SUM_WIDTH(SUM_WIDTH)

) conv1(
    .clk(clk),
    .reset_n(reset_n),
    .start(start),
    .image_in_linear(image_in_linear),
    .weights_linear(weights_linear),
    .biases_linear(biases_linear),
    .finish(finish),
    .map(map)
);
    initial begin
        // Reset
        reset_n = 0;
        #10 reset_n = 1;

        // Load image, weights, biases
        $readmemh("C:/Users/Acer/Downloads/CNN-FPGA-Implementation-main/src/IntelHEX/mnist_in.hex", image_ram);
        $readmemh("C:/Users/Acer/Downloads/CNN-FPGA-Implementation-main/src/IntelHEX/conv2_weight.hex", weights_ram);
        $readmemh("C:/Users/Acer/Downloads/CNN-FPGA-Implementation-main/src/IntelHEX/conv2_bias.hex", biases_ram);

        // Pack into buses
        for (i = 0; i < IN_IMG_SIZE * IN_IMG_SIZE * IN_CHANNELS; i = i + 1)
            image_in_linear[i*DATA_WIDTH +: DATA_WIDTH] = image_ram[i];

        for (i = 0; i < KERNEL_SIZE * KERNEL_SIZE * IN_CHANNELS * OUT_CHANNELS; i = i + 1)
            weights_linear[i*DATA_WIDTH +: DATA_WIDTH] = weights_ram[i];

        for (i = 0; i < OUT_CHANNELS; i = i + 1)
            biases_linear[i*DATA_WIDTH +: DATA_WIDTH] = biases_ram[i];

        // Start convolution
        #20 start = 1;
        #10 start = 0;

        // Wait for conv_done
        wait (finish);
        $display("First Convolutional Layer done.");

    end
endmodule
```

M. Second Max Pooling and ReLU Layer Module

```
`timescale 1ns / 1ps

module maxpool_layer_2 #(
    parameter IMG_WIDTH = 10,
    parameter IMG_HEIGHT = 10,
    parameter CHANNELS = 3,
    parameter POOL_SIZE = 2,
    parameter DATA_WIDTH = 16
)(
    input wire clk,
    input wire reset_n,
    input wire start,
    input wire signed [IMG_WIDTH*IMG_HEIGHT*CHANNELS*DATA_WIDTH-1:0] data_in,
    output reg finish,
    output reg signed [(IMG_WIDTH/2)*(IMG_HEIGHT/2)*CHANNELS*DATA_WIDTH-1:0] data_out
);

    // === Internal 2D arrays ===
    reg [DATA_WIDTH-1:0] input_data    [0:CHANNELS-1][0:IMG_WIDTH*IMG_HEIGHT-1];
    reg [DATA_WIDTH-1:0] pooled_data   [0:CHANNELS-1][0:(IMG_WIDTH/2)*(IMG_HEIGHT/2)-1];

    integer loadram_i, channel, row, col, index, pool_index;
    reg [DATA_WIDTH-1:0] max;

    // === MaxPooling Logic ===
    always @(*) begin
        // 1. Load input to RAMs
        for (channel = 0; channel < CHANNELS; channel = channel + 1) begin
            for (loadram_i = 0; loadram_i < IMG_WIDTH * IMG_HEIGHT; loadram_i = loadram_i + 1)
                input_data[channel][loadram_i] = data_in[(channel*IMG_WIDTH*IMG_HEIGHT + loadram_i)*DATA_WIDTH +: DATA_WIDTH];
        end

        // 2. Max pooling each channel
        for (channel = 0; channel < CHANNELS; channel = channel + 1) begin
            pool_index = 0;
            for (row = 0; row < IMG_HEIGHT; row = row + 2) begin
                for (col = 0; col < IMG_WIDTH; col = col + 2) begin
                    index = row * IMG_WIDTH + col;
                    max = input_data[channel][index];
                    if (input_data[channel][index + 1] > max) max = input_data[channel][index + 1];
                    if (input_data[channel][index + IMG_WIDTH] > max) max = input_data[channel][index + IMG_WIDTH];
                    if (input_data[channel][index + IMG_WIDTH + 1] > max) max = input_data[channel][index + IMG_WIDTH + 1];
                    pooled_data[channel][pool_index] = max;
                    pool_index = pool_index + 1;
                end
            end
        end

        // 3. Extract from 2D RAM to 1D output
        for (channel = 0; channel < CHANNELS; channel = channel + 1) begin
            for (loadram_i = 0; loadram_i < (IMG_WIDTH/2)*(IMG_HEIGHT/2)-1; loadram_i = loadram_i + 1)
                data_out[(channel*(IMG_WIDTH/2)*(IMG_HEIGHT/2) + loadram_i)*DATA_WIDTH +: DATA_WIDTH] = pooled_data[channel][loadram_i];
        end
    end

    // === Finish flag ===
    always @(posedge clk or negedge reset_n) begin
        if (!reset_n) finish <= 0;
        else if (start) finish <= 1;
    end

endmodule
```

N. Second Max Pooling and ReLU Layer Testbench Module

```
`timescale 1ns / 1ps

module tb_maxpool2_layer;
    parameter IMG_WIDTH = 10;
    parameter IMG_HEIGHT = 10;
    parameter CHANNELS = 3;
    parameter POOL_SIZE = 2;
    parameter DATA_WIDTH = 8;

    reg clk = 0;
    reg reset_n = 0;
    reg start = 0;
    reg signed [IMG_WIDTH*IMG_HEIGHT*CHANNELS*DATA_WIDTH-1:0] data_in;
    wire signed [(IMG_WIDTH/2)*(IMG_HEIGHT/2)*CHANNELS*DATA_WIDTH-1:0] data_out;
    wire finish;

    integer i;

    // Clock generation
    always #5 clk = ~clk;

    // DUT
    maxpool_layer_2 maxrelu1 (
        .clk(clk),
        .reset_n(reset_n),
        .start(start),
        .data_in(data_in),
        .finish(finish),
        .data_out(data_out)
    );

    initial begin
        // Reset
        reset_n = 0;
        #10 reset_n = 1;

        data_in[15:0] = 16'ha3;

        // Start
        #10 start = 1;
        #1000 start = 0;

        // Wait for finish
        wait (finish);

    end

endmodule
```

O. Second Convolutional Block Module

```
`timescale 1ns / 1ps

module conv_block2 #(
    parameter IN_IMG_SIZE = 12,
    parameter OUT_IMG_SIZE = 10,
    parameter KERNEL_SIZE = 3,
    parameter IN_CHANNELS = 2,
    parameter OUT_CHANNELS = 3,
    parameter DATA_WIDTH = 16
)(
    input wire clk,
    input wire reset_n,
    input wire start,

    input wire signed [DATA_WIDTH*IN_IMG_SIZE*IN_IMG_SIZE*IN_CHANNELS-1:0] image_in_linear,
    input wire signed [DATA_WIDTH*KERNEL_SIZE*KERNEL_SIZE*IN_CHANNELS*OUT_CHANNELS-1:0] weights_linear,
    input wire signed [DATA_WIDTH*OUT_CHANNELS-1:0] biases_linear,

    output wire finish_block,
    output wire signed [DATA_WIDTH*(OUT_IMG_SIZE/2)*(OUT_IMG_SIZE/2)*OUT_CHANNELS-1:0] block_out_linear
);

    wire finish;
    wire signed [DATA_WIDTH*OUT_IMG_SIZE*OUT_IMG_SIZE*OUT_CHANNELS-1:0] conv_out_linear;

    // === First Convolution Layer ===
    conv_layer_2 #(
        .IN_IMG_SIZE(IN_IMG_SIZE),
        .OUT_IMG_SIZE(OUT_IMG_SIZE),
        .KERNEL_SIZE(KERNEL_SIZE),
        .IN_CHANNELS(IN_CHANNELS),
        .OUT_CHANNELS(OUT_CHANNELS),
        .DATA_WIDTH(DATA_WIDTH)
    ) conv2 (
        .clk(clk),
        .reset_n(reset_n),
        .start(start),
        .image_in_linear(image_in_linear),
        .weights_linear(weights_linear),
        .biases_linear(biases_linear),
        .finish(finish),
        .map(conv_out_linear)
    );

    // === First MaxPool Layer ===
    maxpool_layer_2 #(
      .IMG_WIDTH(OUT_IMG_SIZE),
      .IMG_HEIGHT(OUT_IMG_SIZE),
        .POOL_SIZE(OUT_CHANNELS),
        .CHANNELS(OUT_CHANNELS),
        .DATA_WIDTH(DATA_WIDTH)
    ) pool2 (
        .clk(clk),
        .reset_n(reset_n),
        .start(finish),  // trigger pooling when conv finishes
        .data_in(conv_out_linear),
        .finish(finish_block),
        .data_out(block_out_linear)
    );

endmodule
```

P. Second Convolutional Block Testbench Module

```
`timescale 1ns / 1ps

module tb_conv2_block;
    parameter IN_IMG_SIZE = 12;
    parameter OUT_IMG_SIZE = 10;
    parameter KERNEL_SIZE = 3;
    parameter IN_CHANNELS = 2;
    parameter OUT_CHANNELS = 3;
    parameter DATA_WIDTH = 16;

    reg clk = 0;
    reg reset_n = 0;
    reg start = 0;

    // Linear inputs
    reg signed [DATA_WIDTH*IN_IMG_SIZE*IN_IMG_SIZE*IN_CHANNELS-1:0] image_in_linear;
    reg signed [DATA_WIDTH*KERNEL_SIZE*KERNEL_SIZE*IN_CHANNELS*OUT_CHANNELS-1:0] weights_linear;
    reg signed [DATA_WIDTH*OUT_CHANNELS-1:0] biases_linear;
    wire finish_block;
    wire signed [DATA_WIDTH*OUT_IMG_SIZE*OUT_IMG_SIZE*OUT_CHANNELS-1:0] block_out_linear;

    // RAMs Instantiate
    reg signed [DATA_WIDTH-1:0] image_ram [0:IN_CHANNELS*IN_IMG_SIZE*IN_IMG_SIZE-1];
    reg signed [DATA_WIDTH-1:0] weights_ram [0:OUT_CHANNELS*IN_CHANNELS*KERNEL_SIZE*KERNEL_SIZE-1];
    reg signed [DATA_WIDTH-1:0] biases_ram [0:OUT_CHANNELS-1];

    integer i;

    // Clock generation
    always #5 clk = ~clk;

    // Instantiate the conv_layer module
    conv_block2 #(
    .IN_IMG_SIZE(IN_IMG_SIZE),
    .OUT_IMG_SIZE(OUT_IMG_SIZE),
    .KERNEL_SIZE(KERNEL_SIZE),
    .IN_CHANNELS(IN_CHANNELS),
    .OUT_CHANNELS(OUT_CHANNELS),
    .DATA_WIDTH(DATA_WIDTH)

) conv2_block(
    .clk(clk),
    .reset_n(reset_n),
    .start(start),
    .image_in_linear(image_in_linear),
    .weights_linear(weights_linear),
    .biases_linear(biases_linear),
    .finish_block(finish_block),
    .block_out_linear(block_out_linear)
);
    initial begin
        // Reset
        reset_n = 0;
        #10 reset_n = 1;

        // Load image, weights, biases
        $readmemh("C:/Users/Acer/Downloads/CNN-FPGA-Implementation-main/src/IntelHEX/mnist_in.hex", image_ram);
        $readmemh("C:/Users/Acer/Downloads/CNN-FPGA-Implementation-main/src/IntelHEX/conv2_weight.hex", weights_ram);
        $readmemh("C:/Users/Acer/Downloads/CNN-FPGA-Implementation-main/src/IntelHEX/conv2_bias.hex", biases_ram);

        // Pack into buses
        for (i = 0; i < IN_IMG_SIZE * IN_IMG_SIZE * IN_CHANNELS; i = i + 1)
            image_in_linear[i*DATA_WIDTH +: DATA_WIDTH] = image_ram[i];

        for (i = 0; i < KERNEL_SIZE * KERNEL_SIZE * IN_CHANNELS * OUT_CHANNELS; i = i + 1)
            weights_linear[i*DATA_WIDTH +: DATA_WIDTH] = weights_ram[i];

        for (i = 0; i < OUT_CHANNELS; i = i + 1)
            biases_linear[i*DATA_WIDTH +: DATA_WIDTH] = biases_ram[i];

        // Start convolution
        #20 start = 1;
        #10 start = 0;

        // Wait for conv_done
        wait (finish_block);
        $display("Second Convolutional Block done.");

    end
endmodule
```

Q. Flatten Layer Module

```
1  `timescale 1ns / 1ps
2
3  module flatten_layer #(
4      parameter FILTER_HEIGHT = 5,
5      parameter FILTER_WIDTH = 5,
6      parameter CHANNELS = 3,
7      parameter DATA_WIDTH = 16
8  )(
9      input wire clk,
10     input wire reset_n,
11     input wire signed [FILTER_HEIGHT*FILTER_WIDTH*CHANNELS*DATA_WIDTH-1:0] data_in,
12     input wire start,
13     output reg signed [FILTER_HEIGHT*FILTER_WIDTH*CHANNELS*DATA_WIDTH-1:0] vector_out,
14     output reg finish
15 );
16   always @(posedge clk or negedge reset_n) begin
17       if(!reset_n) begin
18           finish <= 0;
19           vector_out <= 0;
20       end
21       else begin
22           if(start) begin
23               vector_out <= data_in;
24               finish <= 1;
25           end
26       end
27   end
28 endmodule
```

R. Flatten Layer Testbench Module

```
1   `timescale 1ns / 1ps
2
3   module tb_flatten_layer;
4       parameter FILTER_HEIGHT = 5;
5       parameter FILTER_WIDTH = 5;
6       parameter CHANNELS = 3;
7       parameter DATA_WIDTH = 16;
8
9       reg clk = 0;
10      reg reset_n = 0;
11      reg start = 0;
12
13      reg signed [FILTER_HEIGHT*FILTER_WIDTH*CHANNELS*DATA_WIDTH-1:0] data_in;
14
15      wire signed [FILTER_HEIGHT*FILTER_WIDTH*CHANNELS*DATA_WIDTH-1:0] vector_out;
16      wire finish;
17
18      integer i;
19
20      // Clock
21      always #5 clk = ~clk;
22
23      flatten_layer #(
24          .FILTER_HEIGHT(FILTER_HEIGHT),
25          .FILTER_WIDTH(FILTER_WIDTH),
26          .DATA_WIDTH(DATA_WIDTH),
27          .CHANNELS(CHANNELS)
28      ) flat (
29          .clk(clk),
30          .reset_n(reset_n),
31          .start(start),
32          .data_in(data_in),
33          .vector_out(vector_out),
34          .finish(finish)
35      );
36
37      initial begin
38          // Reset
39          reset_n = 0;
40          #10 reset_n = 1;
41
42          data_in[15:0] = 10;
43
44          // Trigger dense
45          #20 start = 1;
46          #20 start = 0;
47
48          // Wait for completion
49          wait (finish);
50
51          #20 $finish;
52      end
53  endmodule
```

S. ArgMax Layer Module

```
module argmax_layer #(
    parameter IN_SIZE = 10,
    parameter DATA_WIDTH = 16
)(
    input  wire clk,
    input  wire reset_n,
    input  wire start,
    input  wire signed [IN_SIZE*DATA_WIDTH-1:0] class_in,
    output reg finish_argmax,
    output reg [3:0] index_out
);

    integer i;
    reg [1:0] state;

    reg signed [DATA_WIDTH-1:0] val_array [0:IN_SIZE-1];
    reg signed [DATA_WIDTH-1:0] max_val;
    reg [3:0] max_idx;

    always @(posedge clk or negedge reset_n) begin
        if (!reset_n) begin
            finish_argmax <= 0;
            index_out <= 0;
            max_val <= 0;
            max_idx <= 0;
            state <= 0;
        end else begin
            case (state)
                2'd0: begin
                    finish_argmax <= 0;
                    if (start) begin
                        for (i = 0; i < IN_SIZE; i = i + 1)
                            val_array[i] <= class_in[i*DATA_WIDTH +: DATA_WIDTH];
                        state <= 2'd1;
                    end
                end

                2'd1: begin
                    max_val <= val_array[0];
                    max_idx <= 0;
                    i <= 1;  // prepare for loop
                    state <= 2'd2;
                end

                2'd2: begin
                    if (i < IN_SIZE) begin
                        if (val_array[i] > max_val) begin
                            max_val <= val_array[i];
                            max_idx <= i[3:0];
                        end
                        i <= i + 1;
                    end else begin
                        index_out <= max_idx;
                        finish_argmax <= 1;
                        state <= 2'd3;
                    end
                end

                2'd3: begin
                    finish_argmax <= 0;
                    state <= 2'd0;
                end
            endcase
        end
    end

endmodule
```

T. ArgMax Layer Testbench Module

```
1   `timescale 1ns / 1ps
2
3   module tb_argmax_layer;
4       parameter IN_SIZE = 10;
5       parameter DATA_WIDTH = 16;
6
7       reg clk = 0;
8       reg reset_n = 0;
9       reg start = 0;
10
11      reg signed [DATA_WIDTH*IN_SIZE-1:0] input_data_linear;
12      wire finish_argmax;
13      wire [3:0] index_out;
14
15     // Clock generation
16      always #5 clk = ~clk;
17
18      // DUT
19      argmax_layer #(
20          .DATA_WIDTH(DATA_WIDTH),
21          .IN_SIZE(IN_SIZE)
22      ) arg (
23           .clk(clk),
24           .reset_n(reset_n),
25           .start(start),
26          .class_in(input_data_linear),
27          .finish_argmax(finish_argmax),
28          .index_out(index_out)
29      );
30
31      initial begin
32           // Reset
33          reset_n = 0;
34          #10 reset_n = 1;
35
36          // Example test case: output should be 1
37          input_data_linear[159:0] = 160'b0;
38          input_data_linear[15:0] = 16'd20; // digit 0 probability
39          input_data_linear[31:16] = 16'd35; // digit 1 probability
40          input_data_linear[127:112] = 16'd90; // digit 7 probability
41
42          #20 start = 1;
43          #10 start = 0;
44           wait(finish_argmax);
45        #20 $finish;
46      end
47
48  endmodule
```

U. Fully Connected Layer Module

```
`timescale 1ns / 1ps

module fc_layer #(
    parameter IN_SIZE = 75,
    parameter OUT_SIZE = 10,
    parameter DATA_WIDTH = 16
)(
    input  wire clk,
    input  wire reset_n,
    input  wire start,
    input  wire signed [DATA_WIDTH*IN_SIZE-1:0]             map_in_linear,
    input  wire signed [DATA_WIDTH*IN_SIZE*OUT_SIZE-1:0]    weights_linear,
    input  wire signed [DATA_WIDTH*OUT_SIZE-1:0]            biases_linear,
    output reg finish_dense,
    output reg signed [DATA_WIDTH*OUT_SIZE-1:0]             predict_out_linear
);

    // === Internal arrays ===
    reg signed [DATA_WIDTH-1 : 0]   map_in_array   [0:IN_SIZE-1];
    reg signed [DATA_WIDTH-1 : 0]   weights_array  [0:OUT_SIZE*IN_SIZE-1];
    reg signed [DATA_WIDTH-1 : 0]   biases_array   [0:OUT_SIZE-1];
    reg signed [DATA_WIDTH*2+4 : 0] log_reg        [0:OUT_SIZE-1];

    // === Load data into RAMs ===
    integer index;
    always @(posedge clk or negedge reset_n) begin
        for (index = 0; index < IN_SIZE; index = index + 1)
            map_in_array[index] = map_in_linear[index*DATA_WIDTH +: DATA_WIDTH];
        for (index = 0; index < OUT_SIZE*IN_SIZE; index = index + 1)
            weights_array[index] = weights_linear[index*DATA_WIDTH +: DATA_WIDTH];
        for (index = 0; index < OUT_SIZE; index = index + 1)
            biases_array[index] = biases_linear[index*DATA_WIDTH +: DATA_WIDTH];
    end
    // === FSM ===
    reg [1:0] state = 2'b00;
    integer node_index;
    reg signed [DATA_WIDTH+DATA_WIDTH+4:0] accum;

    always @(posedge clk or negedge reset_n) begin
        if (!reset_n) begin
            state <= 0;
            finish_dense <= 0;
            node_index <= 0;
            predict_out_linear <= 0;
        end else begin
            case (state)
                2'b00: begin // IDLE
                    if (start) begin
                        node_index <= 0;
                        finish_dense <= 0;
                        state <= 2'b01;
                    end
                end

                2'b01: begin // COMPUTE
                    accum = 0;
                    for (index = 0; index < IN_SIZE; index = index + 1)
                        accum = accum + map_in_array[index] * weights_array[node_index*IN_SIZE + index];
                    accum = accum + biases_array[node_index];
                    log_reg[node_index] <= accum;

                    if (node_index == OUT_SIZE-1) begin
                        node_index <= 0;
                        state <= 2'b10;
                    end else begin
                        node_index <= node_index + 1;
                    end
                end

                2'b10: begin // WRITE OUT
                    for (index = 0; index < OUT_SIZE; index = index + 1)
                        predict_out_linear[index*DATA_WIDTH +: DATA_WIDTH] <= log_reg[index][DATA_WIDTH-1:0];
                    finish_dense <= 1;
                    state <= 2'b00;
                end
            endcase
        end
    end

endmodule
```

V. Fully Connected Layer Testbench Module

```
`timescale 1ns / 1ps

module tb_fc_layer;
    parameter IN_SIZE = 75;
    parameter OUT_SIZE = 10;
    parameter DATA_WIDTH = 16;

    reg clk = 0;
    reg reset_n = 0;
    reg start = 0;

    reg signed [DATA_WIDTH*IN_SIZE-1:0] map_in_linear;
    reg signed [DATA_WIDTH*IN_SIZE*OUT_SIZE-1:0]   weights_linear;
    reg signed [DATA_WIDTH*OUT_SIZE-1:0]           biases_linear;

    wire signed [DATA_WIDTH*OUT_SIZE-1:0]             predict_out_linear;
    wire finish_dense;

    // RAMs Instantiate
    reg signed [DATA_WIDTH-1:0] image_ram [0:IN_SIZE-1];
    reg signed [DATA_WIDTH-1:0] weights_ram [0:IN_SIZE*OUT_SIZE-1];
    reg signed [DATA_WIDTH-1:0] biases_ram [0:OUT_SIZE-1];


    integer i;

    // Clock
    always #5 clk = ~clk;

    fc_layer #(
        .IN_SIZE(IN_SIZE),
        .OUT_SIZE(OUT_SIZE),
        .DATA_WIDTH(DATA_WIDTH)
    ) fc (
         .clk(clk),
         .reset_n(reset_n),
         .start(start),
         .map_in_linear(map_in_linear),
         .weights_linear(weights_linear),
         .biases_linear(biases_linear),
         .finish_dense(finish_dense),
         .predict_out_linear(predict_out_linear)
         );

    initial begin
         // Reset
         reset_n = 0;
         #10 reset_n = 1;

         // Load image, weights, biases
         $readmemh("C:/Users/Acer/Downloads/CNN-FPGA-Implementation-main/src/IntelHEX/mnist_in.hex", image_ram);
         $readmemh("C:/Users/Acer/Downloads/CNN-FPGA-Implementation-main/src/IntelHEX/dense_weight.hex", weights_ram);
         $readmemh("C:/Users/Acer/Downloads/CNN-FPGA-Implementation-main/src/IntelHEX/dense_bias.hex", biases_ram);

         // Pack into buses
         for (i = 0; i < IN_SIZE; i = i + 1)
             map_in_linear[i*DATA_WIDTH +: DATA_WIDTH] = image_ram[i];

         for (i = 0; i < IN_SIZE*OUT_SIZE; i = i + 1)
             weights_linear[i*DATA_WIDTH +: DATA_WIDTH] = weights_ram[i];

         for (i = 0; i < OUT_SIZE; i = i + 1)
             biases_linear[i*DATA_WIDTH +: DATA_WIDTH] = biases_ram[i];


         // Trigger dense
         #20 start = 1;
         #20 start = 0;

         // Wait for completion
         wait (finish_dense);

         #20 $finish;
    end
endmodule
```

W. Top CNN Module

```
`timescale 1ns / 1ps

module cnn_top #(
    parameter IMG_WIDTH = 28,
    parameter IMG_HEIGHT = 28,
    parameter IMG_CHANNELS = 1,
    parameter DATA_WIDTH = 16,
    parameter CONV1_OUT_CHANNELS = 2,
    parameter CONV2_OUT_CHANNELS = 3,
    parameter CONV1_KERNEL = 5,
    parameter CONV2_KERNEL = 3
)(
    input wire clk,
    input wire reset_n,
    input wire start,
    output reg finish_out,
    output reg [3:0] class_out
);

    localparam CONV1_OUT_SIZE = 24;
    localparam MAXRELU1_OUT_SIZE = 12;
    localparam POOL_SIZE = 2;
    localparam CONV2_OUT_SIZE = 10;
    localparam MAXRELU2_OUT_SIZE = 5;
    localparam FLATTEN_OUT_SIZE = 75;
    localparam FC_OUT_SIZE = 10;

     // === Wires ===
     wire [DATA_WIDTH*CONV1_OUT_SIZE*CONV1_OUT_SIZE*CONV1_OUT_CHANNELS-1:0] conv1_out_linear;
     wire conv1_finish;

     wire [DATA_WIDTH*MAXRELU1_OUT_SIZE*MAXRELU1_OUT_SIZE*CONV1_OUT_CHANNELS-1:0] maxrelu1_out_linear;

     wire [DATA_WIDTH*CONV2_OUT_SIZE*CONV2_OUT_SIZE*CONV2_OUT_CHANNELS-1:0] conv2_out_linear;
     wire conv2_finish;

     wire [DATA_WIDTH*MAXRELU2_OUT_SIZE*MAXRELU2_OUT_SIZE*CONV2_OUT_CHANNELS-1:0] maxrelu2_out_linear;

     wire [DATA_WIDTH*FLATTEN_OUT_SIZE-1:0] flatten_out_linear;
     wire flatten_finish;

     wire signed [DATA_WIDTH*FC_OUT_SIZE-1:0] fc_out_linear;
     wire fc_finish;

     wire argmax_finish;
     wire [3:0] class_predict;
    // === RAMs ===
    reg signed [DATA_WIDTH-1:0] img_ram [0:IMG_WIDTH*IMG_HEIGHT*IMG_CHANNELS-1];
    reg signed [DATA_WIDTH-1:0] conv1_weight_ram [0:CONV1_KERNEL*CONV1_KERNEL*IMG_CHANNELS*CONV1_OUT_CHANNELS-1];
    reg signed [DATA_WIDTH-1:0] conv1_bias_ram [0:CONV1_OUT_CHANNELS-1];
    reg signed [DATA_WIDTH-1:0] conv2_weight_ram [0:CONV2_KERNEL*CONV2_KERNEL*CONV1_OUT_CHANNELS*CONV2_OUT_CHANNELS-1];
    reg signed [DATA_WIDTH-1:0] conv2_bias_ram [0:CONV2_OUT_CHANNELS-1];
    reg signed [DATA_WIDTH-1:0] dense_weight_ram [0:FLATTEN_OUT_SIZE*FC_OUT_SIZE-1];
    reg signed [DATA_WIDTH-1:0] dense_bias_ram [0:FC_OUT_SIZE-1];

    // === Load memory ===
    initial begin
        $readmemh("C:/Users/Acer/Downloads/CNN-FPGA-Implementation-main/src/IntelHEX/mnist_in.hex", img_ram);
        $readmemh("C:/Users/Acer/Downloads/CNN-FPGA-Implementation-main/src/IntelHEX/conv1_weight.hex", conv1_weight_ram);
        $readmemh("C:/Users/Acer/Downloads/CNN-FPGA-Implementation-main/src/IntelHEX/conv1_bias.hex", conv1_bias_ram);
        $readmemh("C:/Users/Acer/Downloads/CNN-FPGA-Implementation-main/src/IntelHEX/conv2_weight.hex", conv2_weight_ram);
        $readmemh("C:/Users/Acer/Downloads/CNN-FPGA-Implementation-main/src/IntelHEX/conv2_bias.hex", conv2_bias_ram);
        $readmemh("C:/Users/Acer/Downloads/CNN-FPGA-Implementation-main/src/IntelHEX/dense_weight.hex", dense_weight_ram);
        $readmemh("C:/Users/Acer/Downloads/CNN-FPGA-Implementation-main/src/IntelHEX/dense_bias.hex", dense_bias_ram);
    end

    // === Flatten lineares ===
    wire signed [DATA_WIDTH*IMG_WIDTH*IMG_HEIGHT*IMG_CHANNELS-1:0] img_linear;
    wire signed [DATA_WIDTH*CONV1_KERNEL*CONV1_KERNEL*IMG_CHANNELS*CONV1_OUT_CHANNELS-1:0] weight1_linear;
    wire signed [DATA_WIDTH*CONV1_OUT_CHANNELS-1:0] bias1_linear;
    wire signed [DATA_WIDTH*CONV2_KERNEL*CONV2_KERNEL*CONV1_OUT_CHANNELS*CONV2_OUT_CHANNELS-1:0] weight2_linear;
    wire signed [DATA_WIDTH*CONV2_OUT_CHANNELS-1:0] bias2_linear;
    wire signed [DATA_WIDTH*FLATTEN_OUT_SIZE*FC_OUT_SIZE-1:0] dense_weight_linear;
    wire signed [DATA_WIDTH*FC_OUT_SIZE-1:0] dense_bias_linear;
```

```
    genvar i;

generate
    // Image input
    for (i = 0; i < IMG_WIDTH * IMG_HEIGHT * IMG_CHANNELS; i = i + 1) begin : gen_img
        assign img_linear[i*DATA_WIDTH +: DATA_WIDTH] = img_ram[i];
    end

    // Conv1 weights
    for (i = 0; i < CONV1_KERNEL * CONV1_KERNEL * IMG_CHANNELS * CONV1_OUT_CHANNELS; i = i + 1) begin : gen_conv1_weight
        assign weight1_linear[i*DATA_WIDTH +: DATA_WIDTH] = conv1_weight_ram[i];
    end

    // Conv1 biases
    for (i = 0; i < CONV1_OUT_CHANNELS; i = i + 1) begin : gen_conv1_bias
        assign bias1_linear[i*DATA_WIDTH +: DATA_WIDTH]      = conv1_bias_ram[i];
    end

    // Conv2 weights
    for (i = 0; i < CONV2_KERNEL * CONV2_KERNEL * CONV1_OUT_CHANNELS * CONV2_OUT_CHANNELS; i = i + 1) begin : gen_conv2_weight
        assign weight2_linear[i*DATA_WIDTH +: DATA_WIDTH] = conv2_weight_ram[i];
    end

    // Conv2 biases
    for (i = 0; i < CONV2_OUT_CHANNELS; i = i + 1) begin : gen_conv2_bias
        assign bias2_linear[i*DATA_WIDTH +: DATA_WIDTH]     = conv2_bias_ram[i];
    end

    // dense weights
    for (i = 0; i < FLATTEN_OUT_SIZE*FC_OUT_SIZE; i = i + 1) begin : gen_dense_weight
        assign dense_weight_linear[i*DATA_WIDTH +: DATA_WIDTH] = dense_weight_ram[i];
    end

    // dense bias
    for (i = 0; i < FC_OUT_SIZE; i = i + 1) begin : gen_dense_bias
        assign dense_bias_linear[i*DATA_WIDTH +: DATA_WIDTH] = dense_bias_ram[i];
    end
endgenerate
    reg [3:0] state;

    // === FSM VARS ===

        localparam   S_IDLE      = 4'd0,
                     S_CONV1     = 4'd1,
                     S_CONV2     = 4'd2,
                     S_FLATTEN   = 4'd3,
                     S_FC      = 4'd4,
                     S_ARGMAX    = 4'd5,
                     S_FINISH    = 4'd6;

      // === Instantiate modules ===
      wire start_conv1 = (state == S_CONV1);
      wire start_conv2 = (state == S_CONV2);
      wire start_flatten  = (state == S_FLATTEN);
      wire start_dense = (state == S_FC);
      wire start_argmax = (state == S_ARGMAX);


      // === Module instances ===
    conv_block1 conv1 (
        .clk(clk), .reset_n(reset_n), .start(start_conv1),
        .image_in_linear(img_linear),
        .weights_linear(weight1_linear), .biases_linear(bias1_linear),
        .finish_block(conv1_finish),
        .block_out_linear(maxrelu1_out_linear)
    );

    conv_block2 conv2 (
        .clk(clk), .reset_n(reset_n), .start(start_conv2),
        .image_in_linear(maxrelu1_out_linear),
        .weights_linear(weight2_linear), .biases_linear(bias2_linear),
        .finish_block(conv2_finish),
        .block_out_linear(maxrelu2_out_linear)
    );

    flatten_layer flat (
        .clk(clk), .reset_n(reset_n), .data_in(maxrelu2_out_linear), .start(start_flatten), .vector_out(flatten_out_linear), .finish(flatten_finish)
    );

    fc_layer fc (
        .clk(clk), .reset_n(reset_n), .start(start_dense),
        .map_in_linear(flatten_out_linear),
        .weights_linear(dense_weight_linear),
        .biases_linear(dense_bias_linear),
        .finish_dense(fc_finish),
        .predict_out_linear(fc_out_linear)
    );

      argmax_layer arg(
    .clk(clk),
    .reset_n(reset_n),
    .start(start_argmax),
    .class_in(fc_out_linear),
    .finish_argmax(argmax_finish),
    .index_out(class_predict)
);
```

```
176         // === FSM process ===
177     always @(posedge clk or negedge reset_n) begin
178         if (!reset_n) begin
179             state <= S_IDLE;
180             class_out <= 0;
181             finish_out <= 0;
182         end else begin
183             case (state)
184                 S_IDLE:     if (start) state <= S_CONV1;
185                 S_CONV1:    if (conv1_finish) state <= S_CONV2;
186                 S_CONV2:    if (conv2_finish) state <= S_FLATTEN;
187                 S_FLATTEN:  if (flatten_finish) state <= S_FC;
188                 S_FC:    if (fc_finish) state <= S_ARGMAX;
189                 S_ARGMAX:  if(argmax_finish) state <= S_FINISH;
190                 S_FINISH:   begin
191                                 class_out <= class_predict;
192                                 finish_out <= 1;
193                                 state <= S_IDLE;
194                             end
195                 default:    state <= S_IDLE;
196             endcase
197         end
198     end
199
200 endmodule
```

X. Top CNN Testbench Module

```
`timescale 1ns / 1ps

module tb_cnn_top;
    parameter IMG_WIDTH = 28;
    parameter IMG_HEIGHT = 28;
    parameter IMG_CHANNELS = 1;
    parameter DATA_WIDTH = 16;
    parameter CONV1_OUT_CHANNELS = 2;
    parameter CONV2_OUT_CHANNELS = 3;
    parameter CONV1_KERNEL = 5;
    parameter CONV2_KERNEL = 3;

    reg clk = 0;
    reg reset_n = 0;
    reg start = 0;
    wire signed [3:0] class_out;
    wire finish_out;

    integer i;

    // Clock generation
    always #5 clk = ~clk;

    // DUT
    cnn_top top_cnn (
        .clk(clk),
        .reset_n(reset_n),
        .start(start),
        .finish_out(finish_out),
        .class_out(class_out)
    );

    initial begin
        // Reset
        reset_n = 0;
        #10 reset_n = 1;

        // Start
        #10 start = 1;
        #10 start = 0;

        // Wait for finish
        wait (finish_out);
        #10 $finish;
    end

endmodule
```