

Rochester Institute of Technology

RIT Digital Institutional Repository

Theses

5-2020

Design of Hardware CNN Accelerators for Audio and Image Classification

Rohini Jayachandre Gillela
rjg7712@rit.edu

Follow this and additional works at: <https://repository.rit.edu/theses>

Recommended Citation

Gillela, Rohini Jayachandre, "Design of Hardware CNN Accelerators for Audio and Image Classification" (2020). Thesis. Rochester Institute of Technology. Accessed from

This Thesis is brought to you for free and open access by the RIT Libraries. For more information, please contact repository@rit.edu.

DESIGN OF HARDWARE CNN ACCELERATORS
FOR
AUDIO AND IMAGE CLASSIFICATION

ROHINI JAYACHANDRE GILLELA

DESIGN OF HARDWARE CNN ACCELERATORS

FOR

AUDIO AND IMAGE CLASSIFICATION

by

Rohini Jayachandre Gillela

GRADUATE THESIS

Submitted in partial fulfillment
of the requirements for the degree of
MASTER OF SCIENCE
in Electrical Engineering

DEPARTMENT OF ELECTRICAL AND MICROELECTRONIC ENGINEERING
KATE GLEASON COLLEGE OF ENGINEERING
ROCHESTER INSTITUTE OF TECHNOLOGY
ROCHESTER, NEW YORK

MAY, 2020

**DESIGN OF HARDWARE CNN ACCELERATORS FOR AUDIO AND IMAGE
CLASSIFICATION**

ROHINI JAYACHANDRE GILLELA

Committee Approval:

We, the undersigned committee members, certify that Rohini Jayachandre Gillela has completed the requirements for the Master of Science degree in Electrical Engineering.

Mr. Mark A. Indovina, *Graduate Research Advisor* Date
Senior Lecturer, Department of Electrical and Microelectronic Engineering

Dr. Dorin Patru Date
Associate Professor, Department of Electrical and Microelectronic Engineering

Dr. Amlan Ganguly Date
Interim Department Head, Department of Computer Engineering

Dr. Sohail Dianat, *Department Head* Date
Professor, Department of Electrical and Microelectronic Engineering

I would like to dedicate this work to my professor Mr. Mark Indovina, my grandparents, my father Dr. Ravishankar Reddy Gillela, my mother Mrs. Madhavi Reddy, my brother Mr. Anirudh, and friends for their patience, love, and support during my thesis.

Declaration

I hereby declare that except where specific reference is made to the work of others, that all content of this Graduate Paper are original and have not been submitted in whole or in part for consideration for any other degree or qualification in this, or any other University. This Graduate Project is the result of my own work and includes nothing which is the outcome of work done in collaboration, except where specifically indicated in the text.

Rohini Jayachandre Gillela

May, 2020

Acknowledgements

I would like to express my heartfelt gratitude and thank profusely Prof. Mark Indovina for presenting me with an opportunity to work on my graduate thesis and for being a constant source of inspiration, motivation, support, and guidance. I would like to express my highest veneration to Dr. Dorin Patru and Dr. Amlan Ganguly for their support and invaluable feedback on my work. I finally thank my family and friends for their patience.

Abstract

Ever wondered how the world was before the internet was invented? You might soon wonder how the world would survive without self-driving cars and Advanced Driver Assistance Systems (ADAS). The extensive research taking place in this rapidly evolving field is making self-driving cars futuristic and more reliable. The goal of this research is to design and develop hardware Convolutional Neural Network (CNN) accelerators for self-driving cars, that can process audio and visual sensory information. The idea is to imitate a human brain that takes audio and visual data as input while driving. To achieve a single die that can process both audio and visual sensory information, it takes two different kinds of accelerators where one processes visual data from images captured by a camera and the other processes audio information from audio recordings. The Convolutional Neural Network AI algorithm is chosen to classify images and audio data. Image CNN (ICNN) is used to classify images and Audio CNN (ACNN) to classify any sound of significance while driving. The two networks are designed from scratch and implemented in software and hardware. The software implementation of the two AI networks utilizes the Numpy library of Python, while the hardware implementation is done in Verilog®. CNN is trained to classify between three classes of objects, Cars, Lights, and Lanes, while the other CNN is trained to classify sirens from an emergency vehicle, vehicle horn, and speech.

Contents

- Contents** **v**

- List of Figures** **ix**

- List of Tables** **xiii**

- 1 Introduction** **1**
 - 1.1 Research Goals 4
 - 1.2 Thesis Contributions 5
 - 1.3 Organization 6

- 2 Background Research** **8**
 - 2.1 Audio Classification Methods 10
 - 2.2 CNN Architectures 12
 - 2.3 CNN Accelerator Designs 16

- 3 Classification Methodology** **20**
 - 3.1 Convolutional Neural Networks (CNN's) 22
 - 3.2 Max Pooling 26
 - 3.3 Activation Functions 28

3.3.1	Sigmoid Activation Function	29
3.3.2	Tanh Activation Function	30
3.3.3	ReLU Activation Function	32
3.4	Fully connected Layer	33
3.5	Back Propagation	35
3.6	Numerical Example of Convolution Layer	37
4	Software Implementation	42
4.1	TensorFlow	42
4.1.0.1	What is a tensor?	43
4.1.0.2	Graphs and sessions	43
4.1.0.3	Tensorboard	44
4.2	Image Classification using TensorFlow	45
4.2.0.1	Load Dataset into TensorFlow	45
4.2.0.2	Convolutional Neural Network and Architecture	49
4.3	Training	52
4.3.1	Adam Optimizer	53
4.3.2	Loss and Accuracy	55
4.3.3	Tensorboard Output	55
4.4	Audio Classification using TensorFlow	63
4.4.1	Mel Frequency Cepstral Coefficients (MFCC)	63
4.4.2	Network Architecture	68
5	Network Architecture	84
5.0.1	Network Architecture for Image Classification	85
5.0.2	Network Architecture for Audio Classification	88

5.0.3	CNN Architecture Summary	89
5.1	Fixed Point Representation and Computation for Hardware Efficiency	92
6	Hardware Design & Implementation	96
6.1	System Requirements	96
6.2	System Design and Architecture	97
6.3	Network Configurations	101
6.4	Memory Configuration and Transactions	104
6.5	Design and Implementation of Convolutional Layers	113
6.6	Design and Implementation of Fully Connected Layers	116
6.7	Basic Hardware Blocks	117
7	Verification of IANET	123
7.1	SystemVerilog	123
8	Results	127
9	Conclusion	131
9.1	Future Work	132
	References	135
I	Source Code	140
I.1	TensorFlow Image CNN	140
I.2	TensorFlow Audio CNN	152
I.3	C Reference Model and Fixed Point Functions	162
I.4	Shared Object file Functions	186
I.5	NumPy Functions	189

I.6	CNN from Scratch - Python	215
I.7	ACCUM	225
I.8	ADD	228
I.9	MUL	230
I.10	RELU	232
I.11	MAXPOOL	235
I.12	ICNN RTL	238
I.13	ACNN RTL	334
I.14	SystemVerilog Test-bench Interface	430
I.15	Driver	433
I.16	Monitor Class	453
I.17	Environment	459
I.18	Top Module	460
I.19	Test	467

List of Figures

- 2.1 Neural Network Flow Chart 13
- 2.2 Top Level Structure of Layer Computing Unit [1] 17
- 2.3 Top Level Structure of Layer Computing Unit [1] 18
- 2.4 Top Level Structure of the Origami Chip [2] 19

- 3.1 Kernel Striding over R-Channel 24
- 3.2 Kernel Striding over G-Channel 25
- 3.3 Kernel Striding over B-Channel 26
- 3.4 Maximum Pooling Heat-map 28
- 3.5 Sigmoid Function Plot 30
- 3.6 Tanh Function Plot 31
- 3.7 ReLU Function Plot 33
- 3.8 Fully Connected Layer 35
- 3.9 Convolution 39

- 4.1 Loss of Image Convolutional Neural Network TensorFlow Model 56
- 4.2 Accuracy of Image Convolutional Neural Network TensorFlow Model 56
- 4.3 Image CNN Data flow Graph - Tensorboard 57
- 4.4 Image CNN Inference Model Graph 58

4.5	Image CNN Weights - Layer 1	59
4.6	Image CNN Weights - Layer 2	59
4.7	Image CNN Weights - Layer 3	60
4.8	Image CNN Weights - Layer 4	60
4.9	Image CNN Weights - Layer 5	61
4.10	Image CNN Weights - Layer 6	61
4.11	Image CNN Weights - Layer 7	62
4.12	Image CNN Weights - Layer 8	62
4.13	Distribution	64
4.14	FFT	64
4.15	Fbank	65
4.16	MFCCs	65
4.17	Envelope Signal	65
4.18	Distribution-Envelope	66
4.19	FFT Envelope	66
4.20	Fbank	67
4.21	MFCCs	67
4.22	Loss of the Audio CNN Model	75
4.23	Audio CNN Model Predictions Histogram	76
4.24	Audio CNN Model Loss	76
4.25	Audio CNN Model Accuracy	76
4.26	Audio CNN Model Graph - Tensorboard	77
4.27	Audio Inference CNN Model Graph	78
4.28	Audio CNN Weights - Layer 1	79
4.29	Audio CNN Weights - Layer 2	79

4.30	Audio CNN Weights - Layer 3	80
4.31	Audio CNN Weights - Layer 4	80
4.32	Audio CNN Weights - Layer 5	81
4.33	Audio CNN Weights - Layer 6	81
4.34	Audio CNN Weights - Layer 7	82
4.35	Audio CNN Weights - Layer 8	82
4.36	Audio CNN Model Softmax Prediction Histogram	83
5.1	CNN Architecture for Image Classification - Convolutional Layers	86
5.2	CNN Architecture for Image Classification - Fully Connected Layers	88
5.3	Convolutional Layers	90
5.4	Fully connected Layers	91
6.1	System-Level Architecture	98
6.2	High Architecture of the Accelerators	99
6.3	State Machine for Top Level Implementation	101
6.4	State Machine for Configuration of Layer Parameters	104
6.5	Main Memory Organization	107
6.6	State Machine to Read Weights of the Layer	111
6.7	State Machine to Read Feature Map of the Layer	112
6.8	Structure of the Convolutional Layer	114
6.9	State Machine for acc-mul in Convolution	116
6.10	Structure of the Fully-Connected Layer	117
6.11	Hardware design for Adder	118
6.12	Hardware design for Accumulator	119
6.13	Hardware design for Multiplier	120

6.14	Hardware design for ReLU	121
6.15	Hardware design for Max-pool	122
7.1	Pin Level Representation of IANET	125
7.2	SystemVerilog Test-bench Block Diagram	126
8.1	Accuracy vs Fixed Point representation ICNN and ACNN	128

List of Tables

- 2.1 CNN Architecture for Music Information Retrieval (MIR) 11
- 2.2 AlexNet - Architectural details 14
- 2.3 VGG16 Net - Architectural details 15

- 4.1 Labeled Audio Dataset Organization 70

- 5.1 ICNN Architecture Summary 92
- 5.2 ACNN Architecture Summary 93

- 6.1 Base Address of Weights 106
- 6.2 Base Address of Feature Maps 106
- 6.3 Accelerator Slave Ports and Addresses 109

- 8.1 28 nm Implementation Results 128
- 8.2 Accelerator Performance Metrics 129
- 8.3 Memory Requirements for Accelerators as per the Bit Width of Fixed Point Implementation 130

Listings

4.1	TensorFlow Sessions	44
4.2	Naming Graph-Node on Tensorboard	44
4.3	Plotting Histograms on Tensorboard	45
4.4	Loading Images into TensorFlow	46
4.5	Creating tfrecords in TensorFlow	47
4.6	TableGen Register Set Definitions	48
4.7	Convolutional Layer in TensorFlow	50
4.8	Max-pool layer in TensorFlow	50
4.9	Fully-Connected Layer in TensorFlow	51
4.10	Softmax in TensorFlow	52
4.11	Audio Dataset Organization	69
4.12	Import Audio Dataset	71
4.13	Shuffle Audio Dataset	72
4.14	Distribution of Dataset	73
I.1	TensorFlow Image CNN	140
I.2	TensorFlow Audio CNN	152
I.3	C Reference Model and Fixed Point Functions	162
I.4	Shared Object file Functions	186

I.5	NumPy Functions	189
I.6	CNN from Scratch - Python	215
I.7	ACCUM	225
I.8	ADD	228
I.9	MUL	230
I.10	RELU	232
I.11	MAXPOOL	235
I.12	ICNN	238
I.13	ACNN	334
I.14	SystemVerilog Test-bench Interface	430
I.15	SystemVerilog Test-bench Driver	433
I.16	SystemVerilog Test-bench Monitor	453
I.17	SystemVerilog Test-bench Environment	459
I.18	SystemVerilog Test-bench Top Module	460
I.19	SystemVerilog Test-bench Test Case	467

Glossary

Acronyms

2D	Two-Dimensional
3D	Three-Dimensional
ACC	Adaptive Cruise Control
ACNN	Audio Convolutional Neural Network
ADAS	Advanced Driver-Assistance Systems
AEB	Automatic Emergency Braking
AI	Artificial Intelligence
ALU	Arithmetic-Logic Unit
API	Application Programming Interface
ASIC	Application-specific Integrated Circuit
CNN	Convolutional Neural Network
CPU	Central Processing Unit

CSV	Comma-Separated Values File
DFT	Design For Test
DNN	Deep Neural Network
DPI	Direct Programming Interface
DUT	Design Under Test
FC	Fully Connected
FCW	Forward Collision Warning
FF	Feed Forward
FFT	Fast Fourier Transform
FIFO	First-In, First-Out
FM	Feature Map
FPGA	Field-Programmable Gate Array
fps	frames per second
GLAD	GoogLeNet for Autonomous Driving
GPU	Graphics Processing Unit
IANET	Image-Audio Network
IC	Integrated Circuit
ICNN	Image Convolutional Neural Network

IHC	Intelligent High Beam Control
IHD	Intensity, Height, and Depth
LC	Lane Centering
LDW	Lane Departure Warning
LIDAR	Light Detection and Ranging
LKA	Lane Keeping Assist
LRM	Local Response Normalization
LSTM	Long Short-Term Memory
MB-LBP	Multi-Block Local Binary Pattern
MFCC	Mel Frequency Cepstral Coefficients
MIR	Music Information Retrieval
MM	Main Memory
NLP	Natural Language Processing
OCR	Optical Character Recognition
Q	Binary fixed point number format
ReLU	Rectified Linear Unit
ResNet	Residual Network
RGB	Red, Green, Blue

RNN	Recurrent Neural Network
RTL	Register Transfer Level
SoC	System on a Chip
TJA	Traffic Jam Assist
TPU	Tensor Processing Unit
TSR	Traffic Sign Recognition
UVM	Universal Verification Methodology
VGG	Visual Geometry Group
WT	Weights
YOLO	You Only Look Once
ZFNet	Zeiler and Fergus Network

Chapter 1

Introduction

The idea of building an autonomous vehicle has been around ever since the 1920s. In the 1920s, the cars were not self-driving cars per se; they were remote-controlled and heavily depended on radio technology. Using morse code, the operator manipulated the steering wheel, brakes, and horn of the vehicle. The control signals to the left or right turn, U-turn, and overtake were sent from a car followed closely from behind to the car in front through a spherical antenna. These cars were known as Phantom auto's. In the 1950s, light detector circuits, and other sensor circuits were incorporated to advance the functioning of autonomous cars by keeping them from crossing lanes and crashing into the sideways, keep track of the location and velocity of other vehicles in surroundings. Experiments were conducted on highways, with lights embedded by the sideways, for the car to detect and move forward. In the late 1970s to 1980s, the first fully self-driving car was built. These cars had 4D vision techniques; they achieved high speeds on roads; they had all the necessary processors and sensors.

The idea of building autonomous cars continued to evolve over the years. The concept of neural networks was first enforced to accelerate autonomous driving cars in the 1990s. In the 2010s, several companies like Nvidia, Tesla, Google, Toyota, Audi, etc., have been investing

in research on autonomous cars to bring them into the everyday life of ordinary people. Most of this research is focused on the processors like CPU's, GPU's, FPGA's, Custom IC's, TPU's and ASIC's and the applications running on them. There are a variety of processors that are specifically designed for autonomous driving like MobileyeQ4 and other AI-driven chips.

Driving is one of the most complex tasks that a human brain comprising over a hundred billion neurons performs. A human brain takes several sensory inputs like visual data, sound internal and external to the car, any imbalance sensed due to wheel alignment, see if the vehicle is steered more than usual to go in one direction (this indicates fault related to wheel alignment)—various other factors to make one decision while driving.

Visual information is considered to be one of the essential forms of data. However, it should be kept in mind that visual information is not entirely sufficient to make a decision while driving. A human can understand the traffic scenario behind the car and can predict the traffic in front. A horn from a car behind can indicate to a human driver that he is too slow for the traffic on the road or that the driver behind needs to overtake. How will a self-driving car understand such a scenario? Self-driving cars should be able to drive between other manually driven cars, which necessitates self-driving cars to understand situations deeper than just taking the visual input and acting upon. The thesis [3] further explains in detail the challenges posed to hearing impaired or deaf drivers.

Most autonomous driving applications are using FPGA's as they are low cost and re-programmable. Designing hardware accelerators for autonomous driving applications can be highly beneficial as the hardware design can be customized to meet the demands of the application bringing the low power and high-speed implementation to reality.

Some of the most important things to keep in mind while driving are the speed of the vehicle in front and the scenario of vehicles behind to anticipate possible future scenarios and handle them, the signboards on roads, traffic lights, separating lines drawn on the road, keep track of

blind spots while crossing lanes, listen to sirens from emergency vehicles, horn from neighboring vehicles in traffic and sound from a minor rear-end collision. Most of these things can be done by deploying vision computing engines such as the MobileyeQ4 does. To substitute a human brain in driving, one should create a design that can process both the audio and visual information in real-time, leaving no gap in design and making the system robust to unforeseen events.

The current designs are utilizing vision computing engines in autonomous vehicles to achieve most of the present Advanced Driver Assistant Systems (ADAS) features. Eliminating the process of evaluating the sound from surroundings can be disadvantageous as sound provides us with information about things that are out of sight. The sound external to the car provides vital sensory information that can be used to analyze the situation better and ensure passenger safety. To substitute a human in driving a vehicle, it will require an SoC that is designed to take various inputs from surroundings and process the information to produce a relevant response at every instance.

The design of accelerators in this project will focus on receiving audio and visual inputs, and process audio information with a sound processing engine while processing visual data with a vision computing engine. The main focus of this thesis lies in the integration of sound and image processing. The idea takes its origin from the observation of a human processing information. The main inputs to the human brain while driving are the visual and acoustic information that play a vital role in decision making.

There are several intelligent systems that are replacing human decision making in the field of driving, for example, the most recent ADAS include features like Lane Departure Warning (LDW), Lane Keeping Assist (LKA), Lane Centering (LC), Forward Collision Warning (FCW), Automatic Emergency Braking (AEB), Adaptive Cruise Control (ACC), Traffic Jam Assist (TJA), Traffic Sign Recognition (TSR), Intelligent High Beam Control (IHC), Blind Spot Detection System, Collision Avoidance System, Crosswind Stabilization, Pedestrian Recognition,

Tire Pressure Monitoring, Bird's eye view, auto parking systems, fuel indicators, etc.

Now that self-driving cars are introduced, there is information that is captured from several cameras and processed. All of these systems are exploiting the visual information to the best possible. However, it should be realized that visual data does not entirely suffice the decision making of an autonomous vehicle. The acoustic data, along with visual data, can be a great combination to solve various potential issues. Some issues that the sound can solve are siren from an emergency vehicle that is out of sight, vehicle honking, children playing out of sight etc; there are certain situations where sound plays an important role.

This project focuses on integrating audio and image or visual processing units with the help of efficient hardware design and architecture of the Convolutional Neural Networks (CNNs). It makes use of two CNN accelerators, one for audio and the other one for vision or image processing units. This project also presents the power and area-efficient hardware implementation of the two CNN accelerators.

1.1 Research Goals

Deep Neural Networks (DNN) are widely known for their ability to match the extracted features with the ones that it encountered during training, which is also known as their “ability to learn feature’s”. The autonomous driving industry has peaked its interests in the field of deep neural networks as it has various ADAS that use DNN’s. Mobileye’s EyeQ System on Chip (SoC) is a family of vision-based System on Chip (SoC) solutions developed for self-driving cars. It’s very idea of mono camera originates from the observation of the workings of a human vision system while driving. Mobileye is also developing SoCs, which include multi-sensory information. It is fusing vision with radar to add redundancy and improve performance under some low visibility road conditions. Similarly, there are several developments in the design of SoCs and proprietary

computation cores or accelerators for vision processing units and other sensing technologies like radar.

With the current state of the art, there is still an existing gap between a manually driven and a self-driving car that this thesis addresses. The hardware design in this paper focuses on integrating the audio processing system with a vision or image processing system closing any gap in the decision-making process between a self-driving car and a human driven car. A human, processes both audio and visual information while driving, which makes it a socially viable precision driving system. For self-driving cars to be socially acceptable and more precise and accurate (ultra-precise and unerring), they should be able to process and react to both audio and visual information. The primary research goals of this research are enlisted below:

1. To address a research gap by integrating image and audio processing units in hardware.
2. Implement the two hardware accelerators according to the derived design requirements in lower bit-width fixed-point versions
3. Design area and power efficient hardware for CNN's.
4. Reduce number of parameters that result in reduced network size.
5. Verify the hardware in SystemVerilog.

1.2 Thesis Contributions

The thesis contributions to research and development in the field of digital systems design and verification are as follows:

1. Unique architectural design of CNN accelerators in hardware.

2. Research of several network architectures in the efforts of reducing the size of networks, keeping the accuracy at it's best.
3. Integrating hardware for image and audio processing units.
4. Floating point 32 to fixed point conversion, introducing techniques such as clipping, inference with different bit-widths.
5. Development of SystemVerilog test-bench for the verification of integrated hardware CNN accelerators classifying image and audio data.

1.3 Organization

The structure of the thesis is as follows:

- Chapter 2: This Chapter elaborates on various existing architectures of CNNs both for Audio and Image processing. Also gives some background on existing models that integrate different sensory information giving a basis for this research.
- Chapter 3: This chapter discusses the methodology that is followed to implement Convolutional Neural Networks from the very basic concepts like convolution, ReLu, MaxPool, Fully Connected Network, Back Propagation, etc.
- Chapter 4: Software Implementation that includes the process of implementing the networks in Tensorflow to implementing them from scratch in Python and C is discussed in this chapter.
- Chapter 5: This chapter discusses in detail the process of reducing the network size, the reduced bit-width implementations for successful computation in fixed-point representation.

-
- Chapter 6: This chapter explains the hardware design implementation of CNN's in detail. The techniques deployed to perform memory reads for weights and input feature maps.
 - Chapter 7: The process of development of SystemVerilog test-bench for IANET accelerator is discussed in this chapter.
 - Chapter 8: The results of the software and hardware implementation are discussed here.
 - Chapter 9: Conclusion and future work are discussed in this chapter.
 - Appendix

Chapter 2

Background Research

The existing research, related models of CNN, and the research gap are discussed in detail. There are a vast number of on-going and existing projects for Advanced Driver Assistance Systems (ADAS). Most of them are vision-based, as this study suggests.

Several image-based algorithms are used to extract information from the images captured using a camera. For example, the authors of the paper [4] present a sign recognition system that uses an algorithm to correct the rotation angle at the recognition stage and an integral image-based algorithm to segment the speed limit number. This work focused on capturing the information displayed on the rectangular sign boards efficiently using the Multi-Block Local Binary Pattern (MB-LBP) feature extraction method.

Similarly, the research in [5] uses GoogLeNet's structure to create a new algorithm called GoogLeNet for Autonomous Driving (GLAD) model. The work in [5] is based on different approaches depending on the way the driving environment information is processed, such as the Mediated Perception approach, Behavior Reflex approach, and Direct Perception approach. In mediated Perception approach, features of importance are extracted, deploying several techniques, and the environment is assumed to be unknown. The algorithm obtaining information

assumes the information is being passed on to an Artificially Intelligent (AI) engine. In the Behavioral Reflex approach, the neural network is trained to monitor and anticipate various driving scenarios to make a driving decision. In the Direct Perception approach, Convolutional Neural Network (CNN) learns to extract pre-selected features from the input image and assumes full knowledge of the road architecture. In the process of developing GLAD, the authors modified the affordance parameters such that the autonomous car is fully aware of all other vehicles in its surrounding environment along with the change in the controller process to provide accurate decisions based on new information provided.

As it is known, pedestrian recognition contributes to the most vital information during the decision-making process of autonomous driving systems. The authors of [6] present a novel 2D representation of a 3D point cloud inclusive of the intensity information returned from the LIDAR. CNN, one of the powerful DNN algorithms, is used to handle this data to identify pedestrians in complex visual driving scenes accurately. A 3D space is created that combines the Intensity, Height, and Depth (IHD representation) that is plotted as an image and passed into a CNN for training. The AlexNet architecture is exploited in the process of extracting underlying patterns from the Intensity information.

The CNN in [7] is used for dynamic object detection in grid maps that help identify any forthcoming obstacles on the plane. The article [8] presents a unique approach that combines different multiple geometric views of dynamic traffic scenarios to be able to detect the object in motion. The authors of [9] combine the vision and LIDAR data to build precise terrain maps for autonomous driving vehicles. Likewise there are several articles like [10–18] that make an invaluable contribution to today's ameliorating field of ADAS and autonomous driving systems. However, as discussed in detail, most of them are vision-based systems making use of powerful AI algorithms such as DNNs. It is evident from the above research that CNN is one of the most powerful AI algorithms utilized for image classification. There are several architectures for CNN

that are highly accurate for image classification.

References [2], [19], [20], present examples of the wide range of applications using CNN to solve DNN problems. The CNN is known for its weight sharing and feature extraction qualities that are being widely taken advantage of in the field of vision or image processing. In the past, CNN's were used and deemed successful in the field of audio processing, where the audio is converted into visual format called spectrograms. The Section 2.1 reviews some of the audio classification CNN architectures. The Section 2.2 reviews some of the widely used image classification architectures.

2.1 Audio Classification Methods

The Convolutional Neural Networks are widely used in image classification while Recurrent Neural Network [21] (RNN) or Long Short Term Memory [22] (LSTM) are the two popularly used algorithms for audio processing.

The research in [23] identified the challenges posed to the Music Information Retrieval (MIR) system. Due to several highly elusive characteristics of audio data, recouping the vital features becomes essential for the performance of a sound classification system. The research proves that the CNN algorithm can capture crucial information and classify the audio data. The audio data is similar to that of images after some form of transitions like FFT (Fast Fourier Transform) and MFCC (Mel Frequency Cepstral Coefficients). This CNN has a total of five layers, where the first layer is a 190×13 input feature map. This input feature map hosts the 13 MFCCs from 190 adjacent frames of one excerpt [23]. In the second layer, there are three kernels of size 46×1 that convolve with a window of 10×13 of the input feature map and stride by a length of 4 each time. Similarly, in the third and fourth layers, there are 15 and 65 feature maps, respectively. In the fourth layer, a full connection takes place that results in the

Table 2.1: CNN Architecture for Music Information Retrieval (MIR)

Layer No.	No. of Kernels	Kernel Size	Type of Layer
1	N/A	N/A	Input
2	3	46×1	Conv
3	15	10×1	Conv
4	65	1×1	Conv
5	1	N/A	FC

output classification. The structure of this CNN model is summarized in Table 2.1. This network followed a training strategy similar to the one implemented in this project, where 60% of the data is for training, remaining 40% for testing and evaluation. This model has classified the music files into ten genres and has achieved 84% accuracy.

However, this study was limited to classifying the data that it was trained on and not accurate on unseen data. The network can be more robust provided the network is extended to have more parameters that retrieve extensive characteristics of musical information.

Similarly, in [24], various CNN architectures like AlexNet[25], VGG[26], Inception[27], and ResNet[28] were used to experiment on 5.24 million hour videos sampled to 30,871 labeled data. Each of these network architectures was modified based on the input size of the feature map to the network. According to the results, the Inception and ResNet networks showed best performance on acoustic event detection. As these networks provide high model capacity and common structures that occur repeatedly in specific regions of the input feature maps.

There has been some research work on integrating audio and visual data processing in the past. Most of the models that combined audio and image data were used for acoustic scene classification. For example the authors of [29] perform acoustic scene classification using parallel combination of LSTM and CNN AI algorithms. The use of RNN in this paper was eliminated due to its vanishing and exploding gradients problem. In [29], the neurons of hidden layers of two algorithms, the LSTM and CNN are combined into a Fully Connected (FC) layer unlike in

this project.

Similarly, the authors of [30], have integrated audio and image analysis results, and used video OCR technology for text analysis from frames to interpret scenes from a news video. They used Natural Language Processing (NLP) techniques to classify or categorize the news stories based on the text obtained from the video OCR process. The models in [29] and [30] were implemented for multi-media as the target product and not for automotive industry.

Note that the typical application of RNN's is classification for connected processes, such as handwriting or speech recognition. It is therefore necessary for an RNN to maintain state to properly detect fluid handwriting or speech. For this work, we will detect abrupt sounds such as a sirens or car horns and there is no need to keep state for classification. Therefore, this research work focus on CNN's for audio classification.

2.2 CNN Architectures

Some of the widely used neural networks are CNN's, RNN, LSTM, Feed Forward networks (FF) and many more. The CNN is used widely to classify images while it is not restricted or limited to doing so. The RNN and LSTM are used widely in natural language processing systems. The Fig. 2.1 gives a brief idea of the neural networks, where it suggests that CNN and RNN are two DNN algorithms while many more exist. It also suggests that there are various architectures for CNN like AlexNet, VGG Net, LeNet, GoogLeNet, and ZFNet, etc., while LSTM is an improvement over RNN.

The AlexNet[25] architecture of deep convolutional neural network was trained on the ImageNet dataset to classify 1.2 million high resolution images into thousand different classes. The AlexNet architecture is designed for inputs of size $227 \times 227 \times 3$. The architecture is summarized in Table 2.2. The AlexNet uses ReLU non-linearity function to prevent vanishing gradient

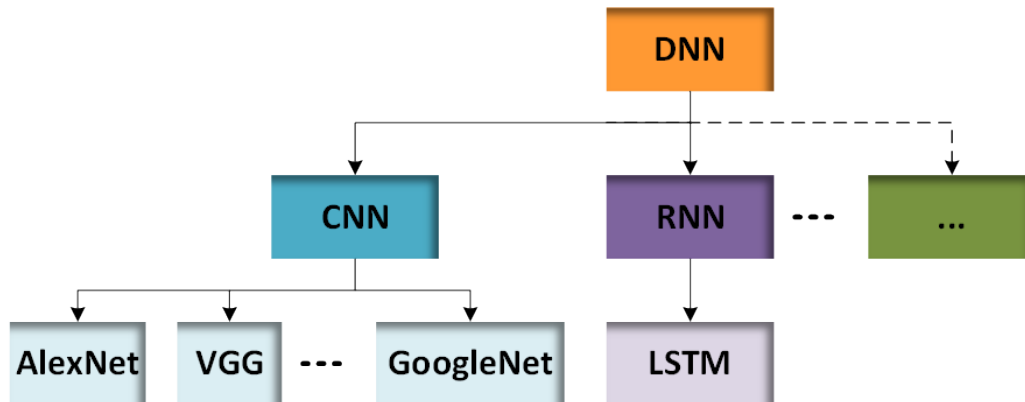


Figure 2.1: Neural Network Flow Chart

problem and also introduce the Local Response Normalization (LRN) technique. The ReLU operation can sometimes unnecessarily intensify the active neuron, and that is why the LRN is used to dampen or amplify the neighboring neurons based on their values. The AlexNet also uses data augmentation techniques to avoid overfitting and improve learning features effectively.

In VGG16[26] or VGG19 there are 16 or 19 layers respectively. The architecture of VGG16 is summarized in Table 2.3. The VGGNet was designed to reduce the number of network parameters while increasing its depth and for that reason a kernel size of 3×3 is used through out the network. The VGG16 is considered a huge network for large-scale image recognition with the total network parameters summing to approximately 138 million parameters as shown in the Table 2.3. This network classifies images into 1000 different classes.

The RNN and LSTM are implemented for sequence detection, speech detection or Natural Language Processing more often. These algorithms are useful and more efficient with any data that involves sequence of events and requires related predictions because Recurrent Neural Networks (RNN) stores the past event and predicts the future event based on the past. However for audio data that includes environmental sounds or urban sounds, the sequence of data is not of significance and hence don't require RNN or LSTM for classification.

Table 2.2: AlexNet - Architectural details

Layer No.	Input	Output	Layer	Stride	Pad	Kernel Size	No. of Params
1	$227 \times 227 \times 3$	$55 \times 55 \times 96$	conv1	4	0	$11 \times 11 \times 3 \times 96$	34944
	$55 \times 55 \times 96$	$27 \times 27 \times 96$	maxpool1	2	0	$3 \times 3 \times 96 \times 96$	0
2	$27 \times 27 \times 96$	$27 \times 27 \times 256$	conv2	1	2	$5 \times 5 \times 96 \times 256$	614656
	$27 \times 27 \times 256$	$13 \times 13 \times 256$	maxpool2	2	0	$3 \times 3 \times 256 \times 256$	0
3	$13 \times 13 \times 256$	$13 \times 13 \times 384$	conv3	1	1	$3 \times 3 \times 256 \times 384$	885120
4	$13 \times 13 \times 384$	$13 \times 13 \times 384$	conv4	1	1	$3 \times 3 \times 384 \times 384$	1327488
5	$13 \times 13 \times 384$	$13 \times 13 \times 256$	conv5	1	1	$3 \times 3 \times 384 \times 256$	884992
	$13 \times 13 \times 256$	$6 \times 6 \times 256$	maxpool5	2	0	$3 \times 3 \times 256 \times 256$	0
6			fc6			$1 \times 1 \times 9216 \times 4096$	37752832
7			fc7			4096×4096	16781312
8			fc8			4096×1000	4097000
						Total	62378344

Table 2.3: VGG16 Net - Architectural details

Layer No.	Input	Output	Layer	Stride	Kernel Size	No. of Params
1	$224 \times 224 \times 3$	$224 \times 224 \times 64$	conv3-64	1	$3 \times 3 \times 3 \times 64$	1792
2	$224 \times 224 \times 64$	$224 \times 224 \times 64$	conv3064	1	$3 \times 3 \times 3 \times 64$	36928
	$224 \times 224 \times 64$	$112 \times 112 \times 64$	maxpool	2	$2 \times 2 \times 64 \times 64$	0
3	$112 \times 112 \times 64$	$112 \times 112 \times 128$	conv3-128	1	$3 \times 3 \times 64 \times 128$	73856
4	$112 \times 112 \times 128$	$112 \times 112 \times 128$	conv3-128	1	$3 \times 3 \times 128 \times 128$	147584
	$56 \times 56 \times 128$	$56 \times 56 \times 128$	maxpool	2	$2 \times 2 \times 128 \times 128$	65664
5	$56 \times 56 \times 256$	$56 \times 56 \times 256$	conv3-256	1	$3 \times 3 \times 128 \times 128$	295168
6	$56 \times 56 \times 256$	$56 \times 56 \times 256$	conv3-256	1	$3 \times 3 \times 128 \times 256$	590080
7	$56 \times 56 \times 256$	$56 \times 56 \times 256$	conv3-256	1	$3 \times 3 \times 256 \times 256$	590080
	$56 \times 56 \times 256$	$56 \times 56 \times 256$	maxpool	2	$2 \times 2 \times 256 \times 256$	0
8	$28 \times 28 \times 512$	$28 \times 28 \times 512$	conv3-512	1	$3 \times 3 \times 256 \times 512$	1180160
9	$28 \times 28 \times 512$	$28 \times 28 \times 512$	conv3-512	1	$3 \times 3 \times 512 \times 512$	2359808
10	$28 \times 28 \times 512$	$28 \times 28 \times 512$	conv3-512	1	$3 \times 3 \times 512 \times 512$	2359808
	$28 \times 28 \times 512$	$14 \times 14 \times 512$	maxpool	2	$2 \times 2 \times 512 \times 512$	0
11	$14 \times 14 \times 512$	$14 \times 14 \times 512$	conv3-512	1	$3 \times 3 \times 512 \times 512$	2359808
12	$14 \times 14 \times 512$	$14 \times 14 \times 512$	conv3-512	1	$3 \times 3 \times 512 \times 512$	2359808
13	$14 \times 14 \times 512$	$14 \times 14 \times 512$	conv3-512	1	$3 \times 3 \times 512 \times 512$	2359808
	$14 \times 14 \times 512$	$7 \times 7 \times 512$	maxpool	2	$2 \times 2 \times 512 \times 512$	0
14	$1 \times 1 \times 25088$	$1 \times 1 \times 4096$	fc		$1 \times 1 \times 25088 \times 4096$	102764544
15	$1 \times 1 \times 4096$	$1 \times 1 \times 4096$	fc		$1 \times 1 \times 25088 \times 4096$	16781312
16	$1 \times 1 \times 4096$	$1 \times 1 \times 1000$	fc		$1 \times 1 \times 4096 \times 1000$	4097000
					Total	138423208

2.3 CNN Accelerator Designs

The authors of [1] propose a tree shaped structure for the convolution operation where the feature map (FM) or image data is read in parallel, which reduces the latency involved in reading input data. The latency is reduced at the expense of high hardware utilization, as can be seen from the architecture. The network model is designed for 28×28 grayscale images with a six layer structure. The focus of this research lies in parallelizing the process as much as possible to improve the speed of the network. There are mainly two different kinds of parallel structured convolutional operations namely multi - convolutional window parallel structure and multi - convolutional kernel parallel structure. The difference between the two parallel structures is that in multi - convolutional window structure, all the neurons in every window of the present feature map share the same kernel while in multi convolutional kernel structure the current window of the feature map is convolved with different kernels.

Fig. 2.2 shows the general structure of each layer computing unit proposed in [1]. All the memory transactions that involve reading weights and feature maps are parallelized. The weights and feature maps are read parallelly using two independent channels from the Ping-Pang memory as shown in the Fig. 2.2. This structure makes use of a ping-pong cache to store intermediate results of convolution or max-pool. The layers alternate between using the ReLU operation as shown by the connections in the figure. The tree-shaped computing structure is proposed to reduce the computational complexity of convolution operation based on two - dimensional and three - dimensional computing structures. The tree-shaped structure is a multi-stream driven pipelined method which is divided into two sections: transmission and computing.

The tree shaped structure of hardware is given in Fig. 2.3. The interconnect logic switch is used to stride across the feature map and get the current window for convolution. The The feature map and kernel values are buffered in FIFO to match the operation speed between the memory

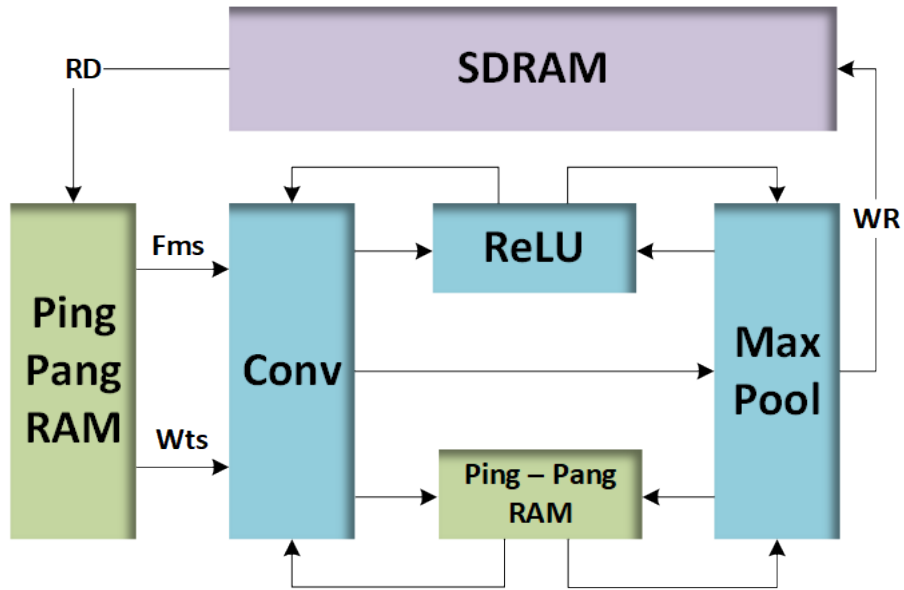


Figure 2.2: Top Level Structure of Layer Computing Unit [1]

and the computation unit. The network layers are designed to have same kernel size and varying feature map sizes in different layers. The buffer size remains the same through out. The process of convolution in this parallel structure requires 25 multipliers and 24 adders. As the Fig. 2.3 implies a distributed way of multiplication that simultaneously multiplies a single row/col of the feature map with respective row or col of the kernel. After the parallel multiplication it sends the values for accumulation to the additive tree shown in the figure. The accumulated value is stored in the registers at every level. The final output is written to the output buffer.

The CNN hardware design in [31] focuses on high speed computation by achieving a frame rate of 60fps. It also keeps the flexibility of kernel sizes as one of the main points, as that makes it suitable for different CNN architectures. The system architecture is designed to be flexible that fits all kinds of CNN architectures with varying kernel sizes. The system architecture of this design consists of data buffers, address generators to read the required window of feature map and kernel, a configurable processing engine, and a temporary storage unit.

The accelerator design has two operating modes, one is convolutional mode and the other is

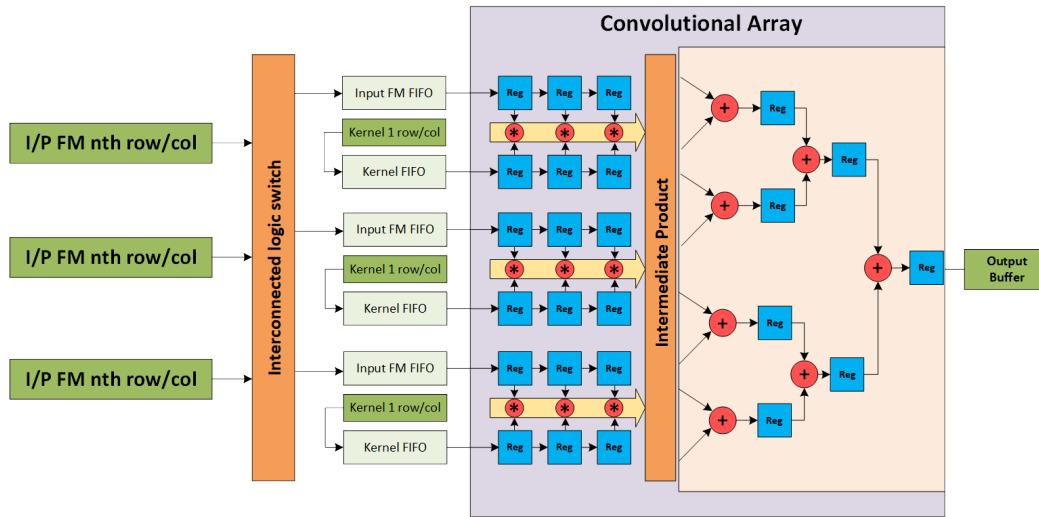


Figure 2.3: Top Level Structure of Layer Computing Unit [1]

fully connected mode. The convolutional mode performs convolution, pooling, and ReLU while the fully connected mode performs fully connected operation and ReLU. The architecture of the processing engine that performs convolution over volumes is similar to having multiple 2-D convolvers. Having multiple 2-D convolvers improves the parallel processing where multiple channel feature maps are convolved at the same time saving processing time. The maximum kernel size that can fit as it is in this architecture is 5×5 and a kernel size with greater dimensions can be decomposed further during convolution. In this kind of implementation, there is a high hardware resource requirement. The speed of processing is achieved at the cost of excessive hardware and power utilization.

The Origami structure presented in [2] is one of the most closely related works. It presents a CNN that takes inputs of size 240×320 , and has a constant kernel size of 7×7 throughout the network. The Origami chip uses two different clock speeds. The Fig. 2.4 shows two clock regions where, modules such as the Filter bank and Image bank run at 250 MHz, while the arithmetic unit runs 500 MHz to achieve a better throughput.

The input pixel stream with multiple channel data is read and stored into the SRAM image

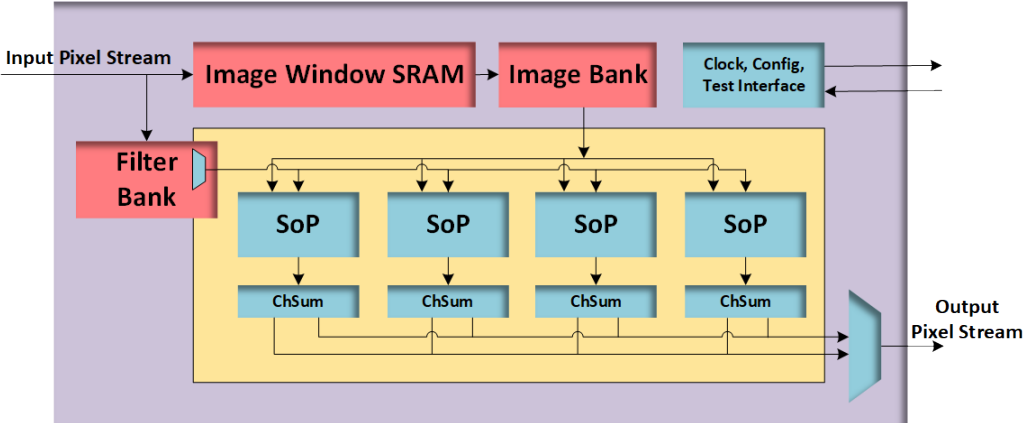


Figure 2.4: Top Level Structure of the Origami Chip [2]

window unit. The spatial window kept in SRAM unit is then loaded into the image bank, where the window size matching the kernel size is pushed into registers and the the window is run across the row. The data stored in registers is put through the SoP (Sum of Products) units where the dot product is performed between the kernel and the image window. The process repeats for all the channels of the input feature map and the channel data is accumulated and stored in ChSum units as shown in Fig. 2.4. The output is then written to output buffer.

Chapter 3

Classification Methodology

The Convolutional Neural Networks as mentioned in the previous chapter, are used to classify two forms of sensory information audio and image data. These networks have a predetermined number of output classes. The output classes are the number of classes that the network needs to classify the inputs into. In the present project, there are two networks, Image Convolutional Neural Network (ICNN) and Audio Convolutional Neural Network (ACNN). As the name suggests, ICNN classifies visual data into four output classes and ACNN classifies audio data into three output classes. The ICNN classifies images into Cars, Lights, Lanes and Pedestrians. The ACNN classifies spectrograms or audio inputs into car horns, sirens, and children playing.

The CNN is a supervised learning AI algorithm where the inputs and outputs are provided to the network during training. The network learns the recurring patterns in the input data and matches them with every input to classify them into their respective classes. These kinds of networks typically need huge labeled data for training. For this purpose, the data was gathered from the existing online datasets like the ImageNet dataset, Caltech dataset, Urban Sound dataset, etc. The CNN in this project makes use of dataset from ImageNet for cars [32], Caltech for lanes [33] and pedestrians [34], LISA Traffic Light Dataset for traffic lights [35], and Urban Sound

Classification dataset [36] for car horn, children playing and siren datasets.

The dataset for audio classification comprised of audio files from Urban Sound dataset and some of the YouTube tracks. All the stereo channel audio files were converted to mono channel audio file using an online tool [37]. All the audio files were later broken down into equal time slices using Python for uniformity of input data to the network.

The dataset comprises of at least 1200 images from each class of which 60% of the data is used to train the network, 20% of the data is used to evaluate the network and the other 20% is used to test the network. Similar data distribution is followed in audio classification network. In this project the CNN's are designed to have eight layers. The ICNN has four convolutional layers followed by four fully connected layers. The ACNN has five convolutional layers and three fully connected layers. Every layer in the networks is followed by a Rectified Linear Unit (ReLU) nonlinearity layer followed by an optional Max-pool layer. The functions implemented in every network layer depend on the requirement and architecture of the network. The ICNN is designed to classify input data into four classes implying there are exactly four neurons in the last layer. The ACNN is designed to classify the input data into three classes implying exactly three neurons in the output layer. Every element in a fully connected network is called a neuron.

A CNN or DNN typically comprises a series of layers that perform several distinct operations to classify various objects by extracting features. The CNN exploits convolution and full connection operations for feature extraction, Rectified Linear activation unit (ReLU) and maximum or average pooling to avoid any regularization and overfitting in the network, computation of cost, and Softmax function to backpropagate the loss during the training process.

This area of research uses the term's kernels and filters, weights and parameters interchangeably. Kernels or filters are matrices that extract specific features by convolving or establishing a full connection between input and output neurons. The bias values are also matrices, usually single-row matrices that add up to the FM of a convolution layer or a fully connected layer to

give an extra drive. The kernels or filters and biases together are called network weights or parameters.

In a deep neural network, primarily, there are three kinds of layers, they are the input layer, the hidden layer, and the output layer. The layer that processes the processed real-world input data is called input layer. The final layer in the network that outputs the result of classification is called the output layer. The layers hidden between the input and output layers are called hidden layers.

The functions deployed in the algorithm are discussed in detail in further sections of this chapter.

3.1 Convolutional Neural Networks (CNN's)

In a CNN, convolutional layers use learned kernels to input feature maps that extract features or create feature maps to summarize the repeated patterns. The learned kernels (or a random kernel during the forward propagation) extend over the height, width and depth of the input feature map. The convolutional layer is a core building block of CNNs because of their weight sharing feature. In the inputs like the images of size $64 \times 64 \times 3$, it is highly impractical to have a full connection established between all the input neurons and the output neurons. It will be an unnecessary computational overload leading to the inefficient implementation of neural networks. In convolutional layers, very few parameters are used compared to a fully connected layer because of their weight sharing nature. A set of local input parameters are connected to a specific output neuron as a function of few parameters that are common between other connections. However, not all the output neurons are connected to all the input neurons in convolutional layer.

Each layer of a convolutional neural network requires an input feature map, an input kernel and an input bias to output a feature map. The input bias matrix is optional based on the network

architecture. The input and output dimensions of the kernel or bias matrix should match with the input feature map of the layer for successful convolution operation. The output of any layer in a CNN is called a Feature Map (FM). A mathematical equation 3.1 determines the output size of the resulting convolutional layer's feature map, where m is the size of the output feature map, W is the width of the input matrix, K is the width of the kernel, P is the padding size, and S is the stride length.

$$m = \frac{W - K + 2P}{S} + 1 \quad (3.1)$$

The architecture of the CNN predefined the kernel and bias shapes corresponding to every convolutional and a fully connected layer, that remain constant throughout the training process. The kernel size can vary or stay constant from between different layers of the same CNN.

CNN uses a random-normalization technique to initialize the network parameters for its first iteration of training. The randomly initialized kernel values then convolve with the equivalent kernel-sized window of neurons in the input layer that result in one corresponding value of the output feature map. The kernel strides through the input matrix in steps determined by stride length, as shown in Fig. 3.1 to convolve with every window. Fig. 3.1 shows a convolution operation between a 7×7 input matrix and a 3×3 kernel with a stride length of one and padding of zero. The equation 3.2 represents convolution operation, where fm is output FM, f is the input image, and the letter h denotes kernel.

$$fm[m,n] = (f * h)[m,n] = \sum_j \sum_k h[j,k] f[m-j, n-k] \quad (3.2)$$

From the equation it is inferred that a dot product between the kernel and the current window of the feature map is performed. All the elements of the resultant matrix from the dot product operation are summed to form a single element of the resultant feature map. If there is a bias

matrix then the bias matrix for the current layer should be of size $m \times m$ where $m = \frac{W-K+2P}{S} + 1$. The bias matrix is added to the feature map after the convolution operation is complete.

The figures 3.1, 3.2, and 3.3 show the channel wise strides that take place during convolution of an input RGB image. After the convolution over all the input channels and the current filter is complete the channel convoluted data is added. This operation is called convolution over volumes. In other words, from Fig.3.1 the R-channel matrix of size 5×5 will be available after convolution with a kernel of size 3×3 and similarly from Fig. 3.2 and Fig. 3.3 channel G and channel B matrices of size 5×5 will be available respectively. These three 5×5 matrices are added together to perform convolution over volumes which is mathematically represented as $1 \times 7 \times 7 \times 3$ input feature map convolving with a kernel of size $1 \times 3 \times 3 \times 3$.

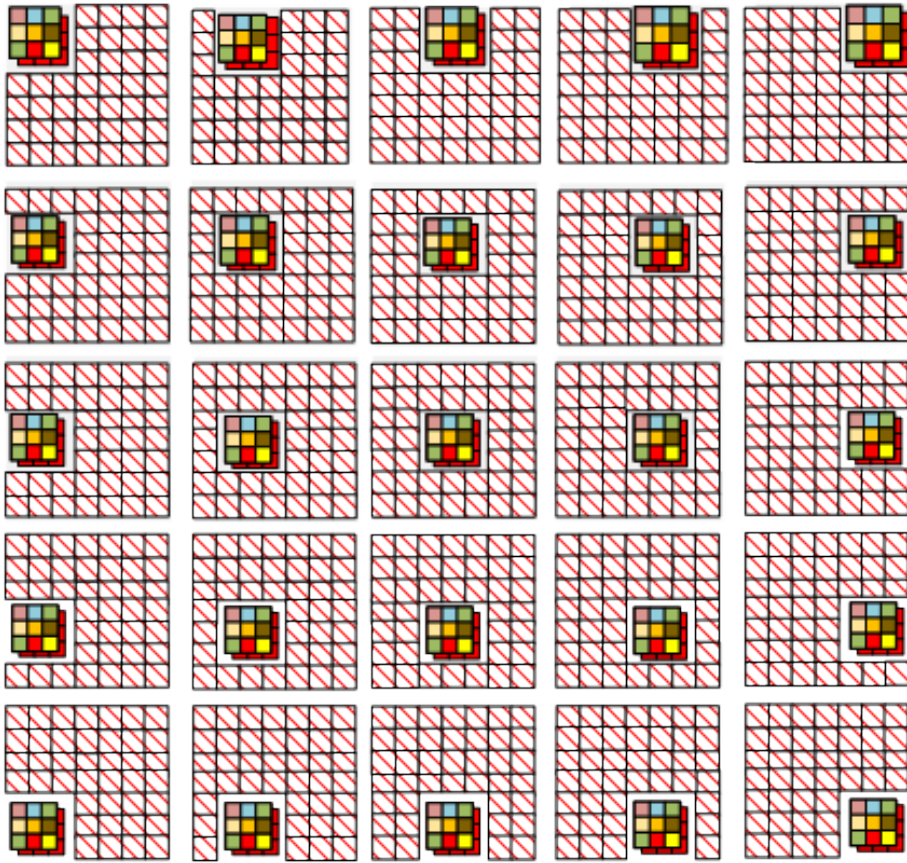


Figure 3.1: Kernel Striding over R-Channel

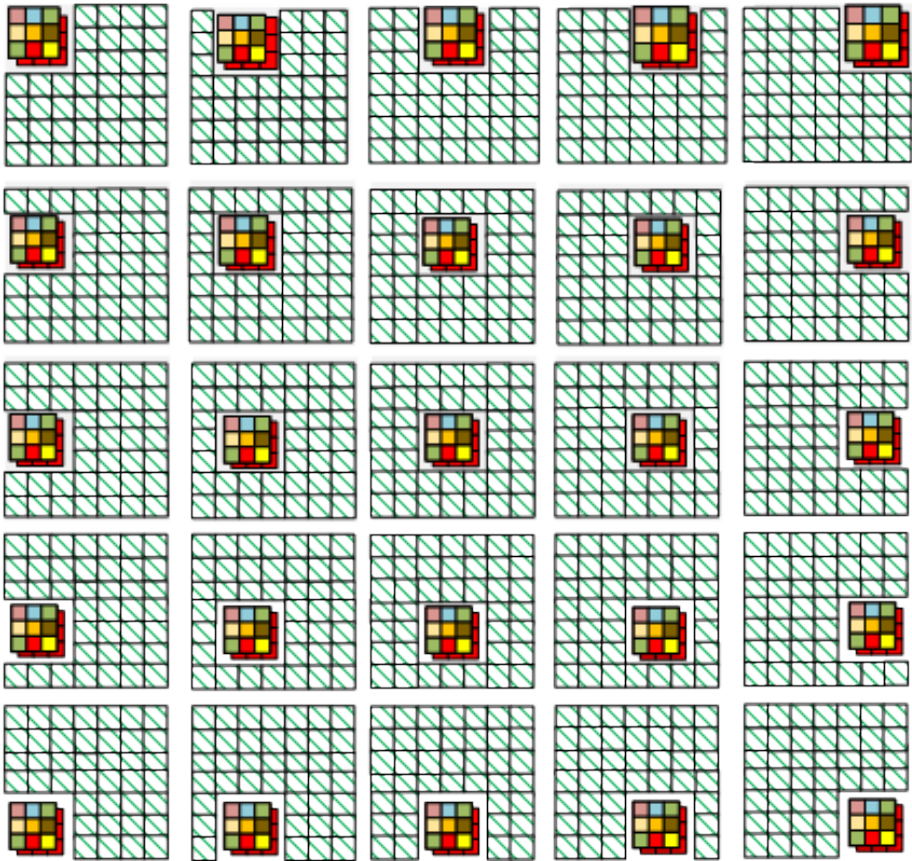


Figure 3.2: Kernel Striding over G-Channel

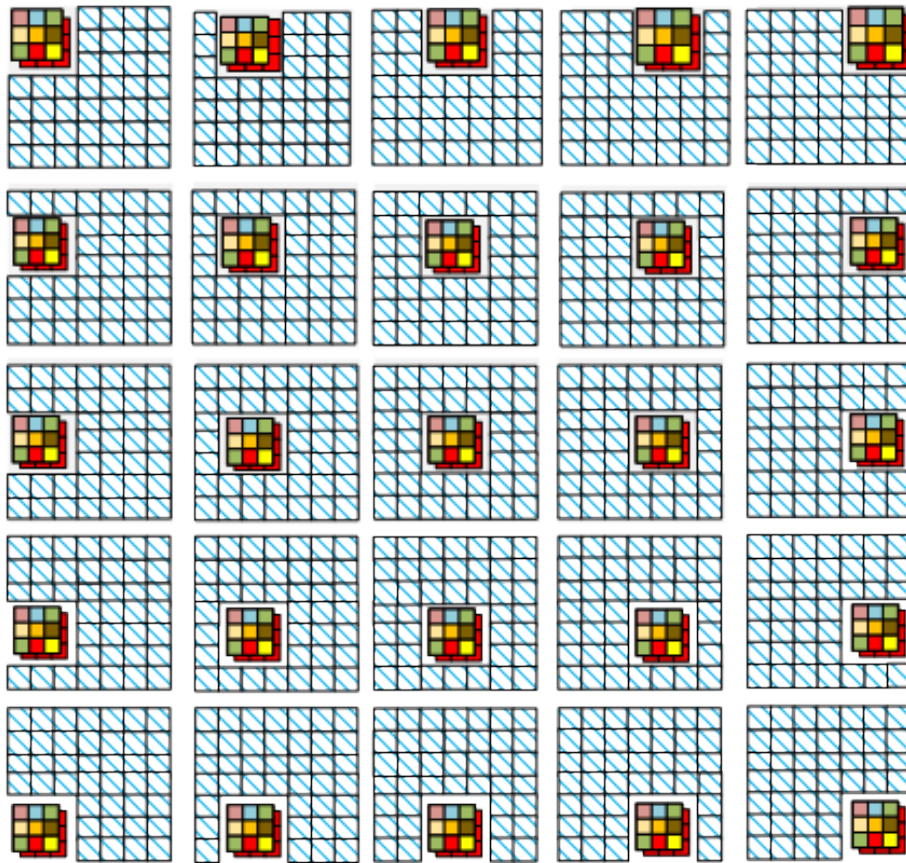


Figure 3.3: Kernel Striding over B-Channel

3.2 Max Pooling

It is a predominantly existing technique to insert a pooling layer between every two successive Convolutional layers or between a convolution layer and a fully connected layer based on the network architecture and related requirements. The purpose of pooling is to progressively curtail the spatial size of the output feature map of current network layer. This results in reduced network parameters and significantly cuts down the computational freight. The process of pooling is also referred to as down sampling the feature map. The process of down sampling the signal reduces the number of resulting data points while restoring the most important data points as inferred

from any signal processing problem. This process helps desensitize the output feature maps by down sampling, the feature maps are highly sensitive to the location of patterns. This process of making the resulting feature maps robust is referred to as “local translation invariance”. Pooling summarizes the presence of prominent features in patches. Pooling is performed on every input feature map separately. Pooling is performed on the output of convolutional layer, specifically after applying the non linearity function.

There are two different kinds of pooling layers. They are Average Pooling and Maximum Pooling layers. Average pooling summarizes the average presence of features while Maximum pooling summarizes the most prominent or active features of the feature map. Most CNN architectures ideally consider a kernel size of 2×2 and a stride length of 2. This way of striding over the feature map will result in half the size of input feature map.

In Average pooling the average value of the patch is calculated and the patch is replaced by the average value in the resulting feature map. The mathematical representation of Average pooling operation is represented by the equation 3.3.

$$y = \frac{fm[i, j] + fm[i + 1, j + 1] + fm[i + m, j + m] + fm[i + m + 1, j + m + 1]}{4} \quad (3.3)$$

In maximum pooling the maximum value among the values in the current window of the input feature map is determined to replace the correlated value in the output feature map. The heat map for a 2×2 window of max pooling is depicted in Fig. 3.4. The maximum number is depicted using a darker color, which will be the output of the operation. The window is strided over the width and height of feature maps just like the convolution strides shown in Fig. 3.1. The window moves back to column one after it reaches the end of first row with a stride of 2 for pooling. The resulting output is half the size of the input matrix if a kernel size is 2×2 and stride is 2. The equation 3.4 states the Max-Pool operation.

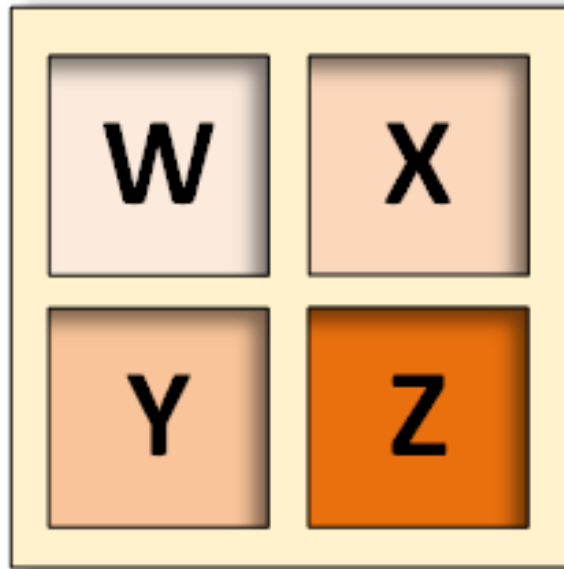


Figure 3.4: Maximum Pooling Heat-map

$$y = \max(fm[i, j], fm[i + 1, j + 1], fm[i + m, j + m], fm[i + m + 1, j + m + 1]) \quad (3.4)$$

Max pooling has been widely used in computer vision tasks like image classification. The underlying reason is because, max-pooling operation extracts the sharp features such as edges and other prominent features of the input image, whereas the average pooling smooths the output by averaging the values. This might result in taking into account the features that don't play as much role in distinguishing one image from another.

3.3 Activation Functions

The activation function is a transfer function that transforms the values of the feature map to range between 0 to 1 or -1 to 1. A deep neural network or a CNN in the absence of an activation function is essentially a linear regression model. The activation function determines as to which

nodes of the network will remain active. The activation functions are of two types, they are linear activation function and non-linear activation function. The linear activation function is an equation of one degree polynomial. This function does not limit the values within a range. The values typically range between $-\infty$ to $+\infty$. The convolutional layer in itself is a linear function. If the activation function used is also a linear function then the neural network will essentially become a linear regression model and will not learn the features efficiently during back-propagation when the weights and gradients are updated. The derivative of a linear function is always a constant value. Hence, non-linear activation functions are used in most DNNs.

There are several activation functions used in neural networks such as linear or identity activation function, non-linear activation functions such as Sigmoid or Logistic function, Tanh or hyperbolic tangent activation function, ReLU or Rectified Linear Unit activation function, Leaky ReLU activation function, Parametric ReLU activation function, Softmax activation function, and Swish activation function. The three most widely used activation functions are Sigmoid, Tanh, and ReLU activation functions.

3.3.1 Sigmoid Activation Function

The Sigmoid function transforms the data points to range between 0 to 1 and results in an S - curve as shown in Fig. 3.5. This function is still used as an activation function in the neural networks as it is non-linear, continuously differentiable, monotonic and has a fixed range for output values. It is not a step function which gives binary values. It is a continuous function that maps every given input to a particular value within its output range. The Sigmoid activation function is represented by the equation 3.5.

$$Y(x) = \frac{1}{1 + e^{-x}} \quad (3.5)$$

The derivative of the function is given by the equation 3.6.

$$Y'(x) = Y(x).(1 - Y(x)) \quad (3.6)$$

As it is inferred from the plot in Fig. 3.5, at the beginning or towards the end, the values of function $y(x)$ don't vary as much for different values of x , which results in vanishing gradients problem. This problem incapacitates the network from learning efficiently. The output of the function is not zero centered which results in sparsely located transformations of inputs to outputs in the 0 to 1 range, making the optimization inefficient.

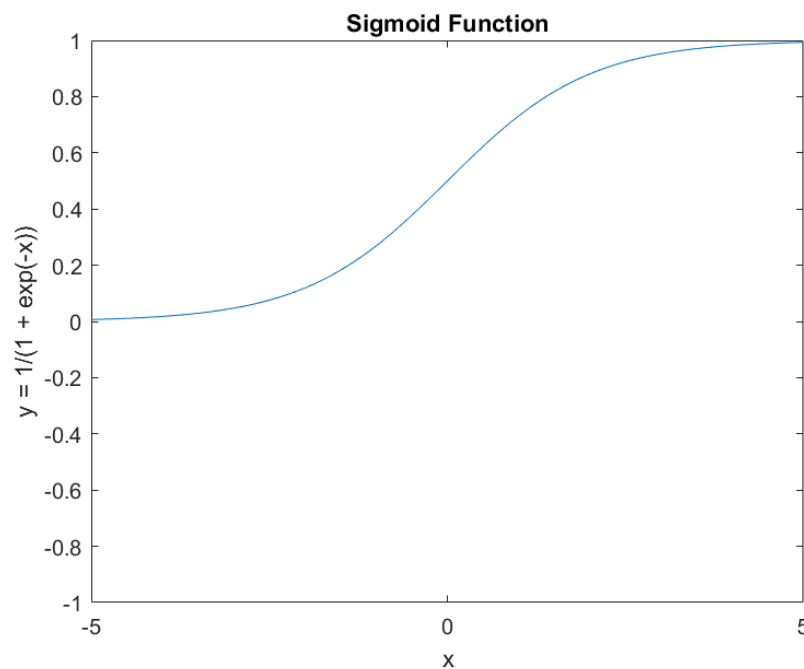


Figure 3.5: Sigmoid Function Plot

3.3.2 Tanh Activation Function

The tanh function transforms the data points to range between -1 to 1 and results in the curve as shown in Fig. 3.6. The output of this function is zero centered unlike the Sigmoid activation

function. It shares the similar properties with Sigmoid function where the output values are continuous and the function is continuously differentiable. The equations 3.7 and 3.8 represent the tanh activation function and its derivative respectively.

$$Y(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}} \quad (3.7)$$

The derivative of $\tanh(x)$ is $\tanh'(x) = 1 - (\tanh(x))^2$, hence $Y'(x)$ is represented as shown in equation 3.8.

$$Y'(x) = 1 - \left(\frac{e^x - e^{-x}}{e^x + e^{-x}}\right)^2 \quad (3.8)$$

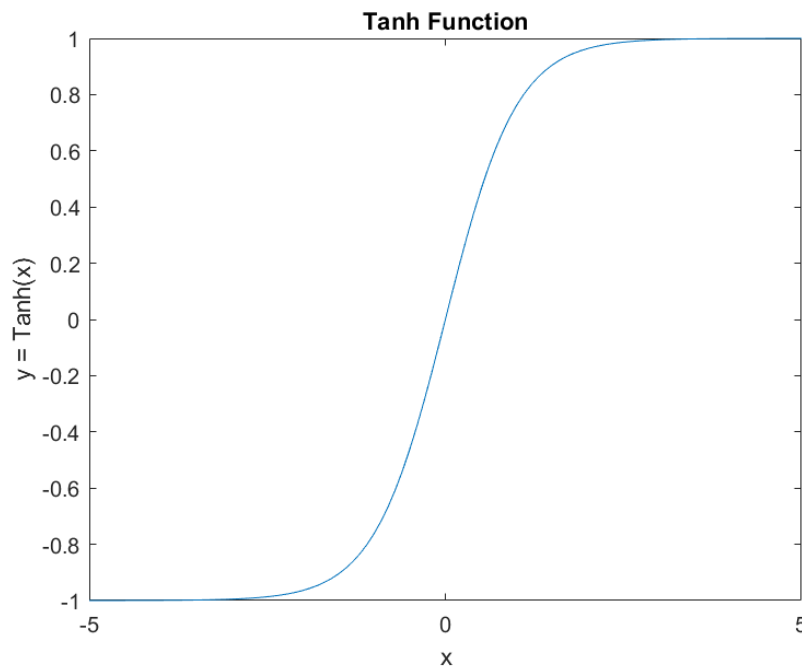


Figure 3.6: Tanh Function Plot

As it can be inferred from the plot shown in Fig. 3.6, the output values don't vary with varying input values resulting in a gradient vanishing problem that makes the learning process of

the network harder.

3.3.3 ReLU Activation Function

In neural networks the loss is propagated backwards from the output layer to the input layer. The gradient diminishes dramatically as it is propagated backwards, and by the time it reaches the input layer the gradient values become so small that they don't make a difference when updated. This is why the learning becomes difficult with vanishing gradients.

This activation function rectifies the problem of vanishing gradients by stopping to propagate diminishing gradients backwards. This function has a gradient slope of 1 which helps keep the gradients propagating backwards same and not drive them to a smaller value. Thus this activation function accelerates the learning process.

In this function, any negative number is replaced by a zero and a positive number remains as is. The equation 3.9 represents the ReLu activation function. The derivative of this function is 1 when the value is greater than zero and the derivative is 0 otherwise.

$$Y(x) = \max(0, x) \quad (3.9)$$

The feature maps resulting from every convolutional or a fully connected layer go through the ReLu activation function to introduce nonlinearity in the algorithm. The Fig. 3.7 depicts the plot of ReLU activation function.

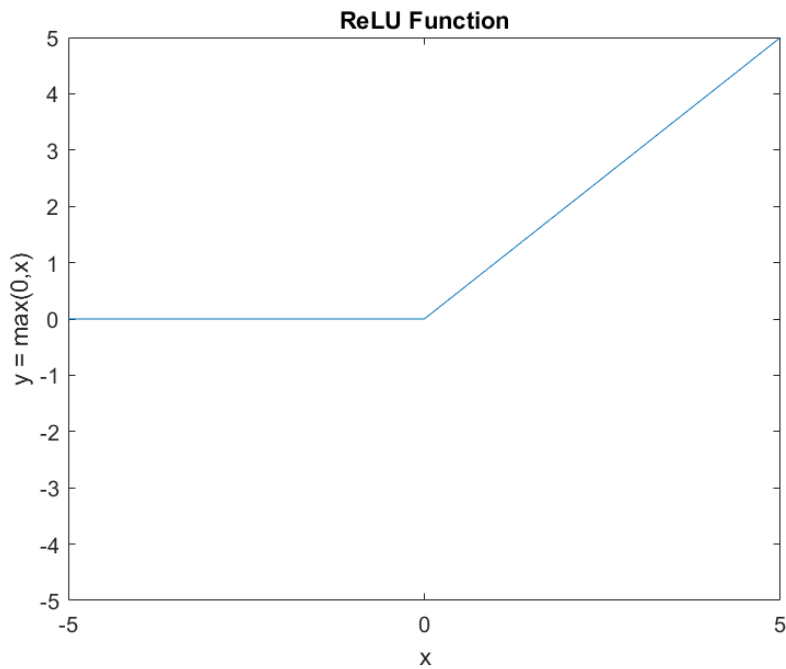


Figure 3.7: ReLU Function Plot

3.4 Fully connected Layer

Fully connected layers are an essential component of Deep Neural Networks like CNNs. In any typical structure of a CNN, fully connected layers are found at the end of the network structure. After a series of convolutional layers, the result of convolutional layers are fed into fully connected layers or a fully connected network that drive the final classification decision of the network. Due to this process the fully connected layers end up being hidden layers and the output layer of the network. In a hidden FC layer, there is an input matrix of neurons from its preceding convolutional layer that is flattened into a single-row matrix, an input kernel of matching dimensions, and an optional input bias matrix. If the input feature map is resulting from a convolutional layer, the transition layer unravels or flattens the FM into a single-row matrix before sending it to the FC layer otherwise feeds it to the fully connected layer as is. The kernel is a two-dimensional

matrix, with its length equal to the number of output neurons and its width equal to the number of input neurons. A matrix multiplication takes place between the flattened input neurons and the input kernel to establish a full connection. This process ensures every input neuron is connected with every output neuron. If there is a bias matrix, the addition of corresponding values of two single-row matrices takes place. This process computes the output neurons. Fig. 3.8 shows two fully connected layers where every neuron in the input layer I is connected with every neuron in layer J with the help of parameters. The equation 3.10 gives a mathematical representation of the fully connected layer.

$$J_i = \sum_{j=0}^k I_i * W_I[i, j] \quad (3.10)$$

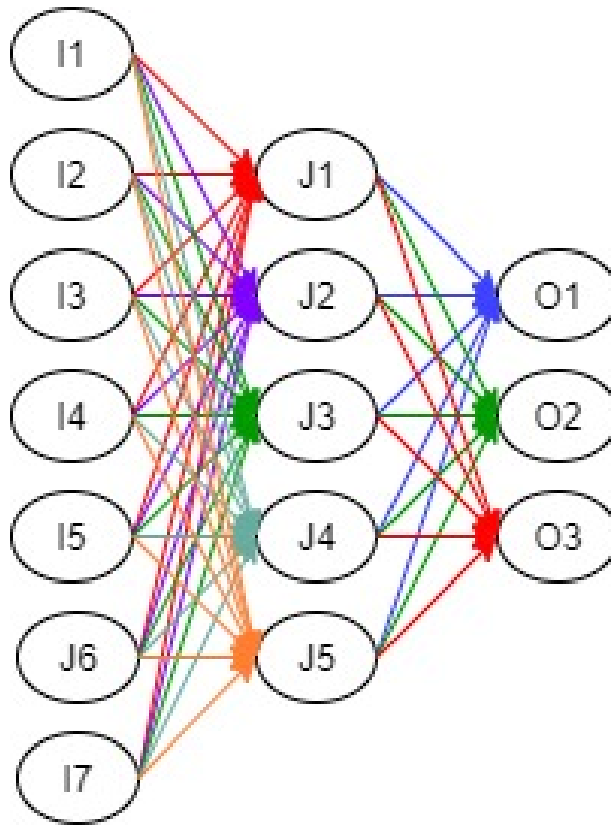


Figure 3.8: Fully Connected Layer

3.5 Back Propagation

In the output layer of CNN, after applying the Softmax function, cross-entropy loss of the network is computed, which is the distance between the outputs computed by the network and the actual intended outputs of the network. The derivative of cross-entropy loss is backpropagated in the network to drive the network's randomly initialized weights in the desired direction. The equation 3.11 represents the Softmax function.

$$F(x) = \frac{\text{Exp}(X_i)}{\sum_{k=0}^N \text{Exp}(X_k)} \quad i = 0, 1, 2, \dots, k \quad (3.11)$$

The Softmax function computes the ratio of the exponential of each input value and sum of all the exponential of every input value. This process normalizes the output neuron values to range between 0 and 1. The summation of all the values after applying Softmax function equals to one.

In the output layer of the CNN a Softmax function is applied as stated above. The cross entropy loss is given by equation 3.12, where y is the intended output of the network and p is the Softmax of network computed output. The equation 3.13 is used to compute Softmax for every element.

$$H(y, p) = - \sum_i y_i \log(p_i) \quad (3.12)$$

$$p_i = \frac{e^{X_i}}{\sum_{k=1}^N e^{X_k}} \quad (3.13)$$

The derivative of p_i or the Softmax function is derived by computing the partial derivative of Softmax function, as shown in equation 3.14.

$$\frac{\partial p_i}{\partial X_k} = \frac{\partial \frac{e^{X_i}}{\sum_{k=1}^N e^{X_k}}}{\partial X_k} = \begin{cases} p_i(1 - p_k) & \text{if}(i = k) \\ -p_i \cdot p_k & \text{if}(i \neq k) \end{cases} \quad (3.14)$$

The equation 3.15 gives a derivative of cross entropy loss shown in equation 3.12.

$$\frac{\partial H}{\partial o_i} = - \sum y_k \frac{\partial \log(p_k)}{\partial o_i} = - \sum y_k \frac{\partial \log(p_k)}{\partial p_k} \times \frac{\partial p_k}{\partial o_i} \quad (3.15)$$

On solving the equation 3.15 further by substituting the partial derivative of Softmax function with respect to the corresponding output neuron p_k and the derivative of log of Softmax function,

the following simple equation 3.16 is derived.

$$\frac{\partial H}{\partial o_i} = p_i - y_i \quad (3.16)$$

To back propagate the loss also means to minimize the gradient. There are several theories on gradient descent optimization as summarized in [38].

The derivative of the loss is propagated from the output layer to the input layer through the hidden layers. The derivative of cross entropy function is derived with respect to every network parameter. Once all the derivatives are derived all the network parameters are updated to minimize the loss and improve network accuracy. The same applies to all hidden layers that include fully connected, activation, pooling and convolution layers. The below equation shows the partial derivative of cross entropy function with respect to network parameters between the hidden layer j and output layer o as shown in Fig. 3.8.

$$\frac{\partial H}{\partial w_{jo}} = -\sum y_k \frac{\partial \log(p_k)}{\partial w_{jo}} = -\sum y_k \frac{\partial \log(p_k)}{\partial p_k} \times \frac{\partial p_k}{\partial o_i} \times \frac{\partial o_i}{\partial w_{jo}} \quad (3.17)$$

The equation 3.17 can be reduced further by substituting previously derived equations for $\frac{\partial H}{\partial o_i}$. The partial derivative of neuron in output layer i.e., $\frac{\partial o_i}{\partial w_{jo}}$ results in J_i as o_i is a sum of product of neurons from preceding hidden layer and current layer parameters denoted by w_{jo} . The network parameters are updated after every iteration based on gradient optimization for high accuracy.

3.6 Numerical Example of Convolution Layer

In convolution operation there is an n-dimensional input matrix and a number of filters that the input has to convolve with. Typically in a CNN at the input layer, an image is read in. The RGB (Red Green Blue) image is of say $3 \times 64 \times 64$ matrix. This indicates that the matrix has a height

and width of 64 and a depth of 3. The depth is otherwise known as number of channels. The filter of size $8 \times 3 \times 3 \times 3$ indicates that there are 8 filters with 3 channels and a height and width of 3. The number of channels in filters should always equal the number of channels in input for a successful convolution operation.

In this section, a CNN is computed with a numeric example for forward and backward propagation. For example a 7×7 matrix with a depth of 3 is convolved with a filter of height, width and depth of 3, 3, and 3 respectively.

Step 1: Firstly the input is read in and the dimensions of input matrix and the filter are determined. If the input matrix and the filter have equal channels, convolution is performed as shown in Fig. 3.9. As depicted in Fig. 3.9 the channel one of filter of size 3×3 is convolved with channel 1 of input matrix of size 7×7 , resulting in an output matrix of size 5×5 . There will be three 5×5 matrices resulting from convolution from three channels, that are summed up. The channel wise operations are explained using a numerical example.

The first channel of the filter is convolved with the first channel of input matrix. The 3×3 filter is convolved with slices of input matrix and translates over the matrix with a stride of one. Input[0:2, 0:2] will be the first slice, [0:2, 1:3] will be the second slice with a stride length of 1. The second slice would be Input[0:2, 2:4] with a stride length of 2. For an image of size $n \times n$ convolving with a filter of size $f \times f$, stride of s and a pooling factor of p the resultant matrix will be of size $[(n + 2p - f)/s + 1] \times [(n + 2p - f)/s + 1]$.

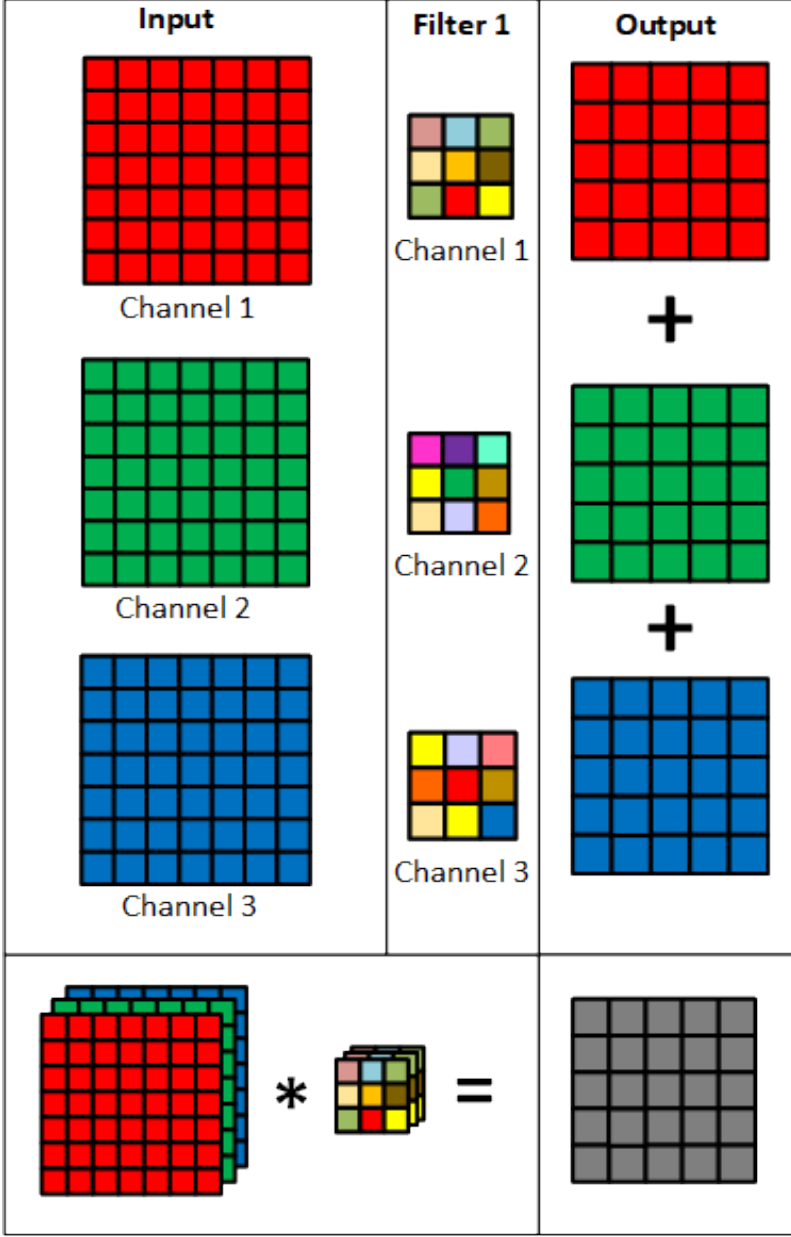


Figure 3.9: Convolution

The Fig. 3.3 shows how convolution is performed across every channel. The filter is strided over every slice of the image as shown in the Fig. 3.1. A dot product is performed between the filter and the slice of images and they are summed to get a corresponding output value.

The dot product between the first slice from first channel of the input matrix and the first channel of the filter is shown below. The sum of the dot product corresponds to first element of the output matrix, that is output[0,0].

$$\begin{array}{ccc} 1 & 2 & 3 \\ 1 & 4 & 5 \\ 2 & 3 & 4 \end{array} * \begin{array}{ccc} 1 & 2 & 3 \\ 2 & 3 & 4 \\ 2 & 4 & 5 \end{array} = \begin{array}{ccc} 1 & 4 & 9 \\ 2 & 12 & 20 \\ 4 & 12 & 20 \end{array} = 84$$

channel 1:

$$\begin{array}{ccc} 1 & 2 & 3 & 4 & 6 & 7 & 4 \\ 1 & 4 & 5 & 3 & 2 & 2 & 1 & & 84 & 107 & 128 & 121 & 93 \\ 2 & 3 & 4 & 6 & 8 & 4 & 3 & 1 & 2 & 3 & 99 & 115 & 118 & 95 & 68 \\ 2 & 3 & 6 & 4 & 3 & 2 & 1 & * & 2 & 3 & 4 & = & 82 & 100 & 114 & 91 & 82 \\ 1 & 2 & 3 & 3 & 4 & 2 & 5 & 2 & 4 & 5 & 81 & 96 & 95 & 72 & 73 \\ 1 & 2 & 5 & 4 & 3 & 2 & 3 & & & & 87 & 115 & 108 & 73 & 77 \\ 2 & 4 & 5 & 7 & 3 & 1 & 4 \end{array}$$

channel 2:

$$\begin{array}{ccc} 1 & 2 & 4 & 4 & 6 & 2 & 4 \\ 1 & 0 & 5 & 3 & 2 & 3 & 1 & & 57 & 57 & 80 & 71 & 62 \\ 2 & 6 & 4 & 6 & 8 & 6 & 3 & 2 & 1 & 3 & 57 & 78 & 86 & 84 & 70 \\ 2 & 2 & 6 & 4 & 3 & 7 & 1 & * & 3 & 1 & 4 & = & 71 & 75 & 81 & 94 & 68 \\ 2 & 6 & 3 & 1 & 4 & 4 & 5 & 1 & 2 & 1 & 55 & 54 & 58 & 64 & 62 \\ 1 & 2 & 2 & 1 & 3 & 2 & 3 & & & & 47 & 45 & 48 & 40 & 59 \\ 2 & 4 & 5 & 1 & 3 & 1 & 4 \end{array}$$

channel 3:

```

1 2 3 4 2 0 4
1 4 5 3 2 2 1           43 55 52 41 25
2 3 4 6 0 4 3   1 2 1   48 62 52 34 35
2 3 6 1 3 2 1 * 1 2 2 = 40 45 43 40 38
1 2 1 3 4 2 2   3 2 1   30 40 42 42 33
1 2 2 4 2 2 3           36 53 55 49 31
2 5 5 7 1 1 4

```

Final Feature map:

```

84 107 128 121 93   57 57 80 71 62  43 55 52 41 25
99 115 118 95  68   57 78 86 84 70  48 62 52 34 35
82 100 114 91  82 + 71 75 81 94 68 + 40 45 43 40 38 =
81 96  95  72 73   55 54 58 64 62  30 40 42 42 33
87 115 108 73 77   47 45 48 40 59  36 53 55 49 31

      184 219 260 233 180
      204 255 256 213 173
      193 220 238 225 188
      166 190 195 178 168
      170 213 211 162 167

```


Chapter 4

Software Implementation

4.1 TensorFlow

There are several Python-based AI tools like Keras, Theano, PyTorch, TensorFlow, etc, that help us design any neural network from the top level. These tools make the process of implementing a neural network easier, as TensorFlow doesn't require the user to learn the low-level details of the deep neural network. For example, low-level details in CNN mean, each pixel in a matrix is being convolved with an element in the kernel corresponding to a particular layer. The Python-based AI tool, Keras is built on top of TensorFlow and Theano.

To have better control over the network means to be able to choose what happens to each variable in the network; to choose if a variable in a network is trainable or not. TensorFlow provides an online debug tool to visualize the changes that take place through the hidden layers of a neural network during training. TensorFlow provides great programming flexibility through their API and great visibility through a powerful online debug tool called Tensorboard, (that will be discussed later in this chapter).

TensorFlow introduces and necessitates the understanding of some concepts like tensors,

graphs, and sessions.

4.1.0.1 What is a tensor?

A scalar is a mathematical entity that represents magnitude only, for example, weight, distance, quantity, etc [39]. A vector is a mathematical entity that gives information on both the magnitude and direction of any physical entity, example: velocity, acceleration, locations, etc. More often, a scalar quantity is multiplied by direction to obtain a vector. For example: to give directions to a location, go 3miles north, 1 mile west, etc, a sum of products of scalars with direction is resulting in a vector. Multiplying a unit vector by a scalar can change the magnitude of the vector but leaves the direction unchanged. To change both the direction and magnitude of the vector, multiplication with scalar is no longer sufficient. And finally a tensor can have one magnitude and multiple directions. For example, a scalar which is magnitude only is a tensor of rank 0 that contains one component only. A vector which has a magnitude associated with one direction is a tensor of rank 1 with three components. A dyad is a tensor of rank 2 that comprises of one magnitude and three directions that constitute to nine components in a tensor. Other mathematical rules apply to tensors that are similar to rules that apply to scalars and vectors.

Examples of some of the rules:

- All scalars are not tensors, although all tensors of rank 0 are scalars.
- All vectors are not tensors, although all tensors of rank 1 are vectors.
- Tensors can be multiplied by other tensors to form new tensors.

4.1.0.2 Graphs and sessions

TensorFlow represents all the computations performed in terms of a graph. To do this, it requires the user to define a graph and create a session. TensorFlow provides various high-level API's like estimator and the Keras neural-network library that generate the graph automatically. The

estimator stores the auto-generated graph in a model directory. TensorFlow's graphs can be useful to visualize the internal processing in a high-level API like the estimator. At low-level step, the session can be created using the below lines of the code:

```
1 sess = tf.Session()
2 sess.run(tf.global_variables_initializer())
```

Listing 4.1: TensorFlow Sessions

`Session.run()` can be used to run each node in the TensorFlow's graph and evaluate the values of tensors. Each node on the graph can be labeled using the `name_scope()` API:

```
1 with tf.name_scope("Output"):
2 net = tf.matmul(net, W["w8"])
3 net = tf.add(net, B["b8"])
```

Listing 4.2: Naming Graph-Node on Tensorboard

The above lines of code can be described as, the node "Output" in the graph contains computations like `matmul()` and `add()`.

4.1.0.3 Tensorboard

Tensorboard is an online web based tool that helps visualize the graphs written in TensorFlow. It has various other features such as plotting histograms of trainable variables, plotting graph for loss and accuracy for training and evaluation of the model. It allows the user to plot images that are being passed into the model, let's the user visualize the distributions of network parameters.

In the code below, `tf.summary.histogram()` plots the histogram on Tensorboard.

```
1 with tf.name_scope("Output"):  
2 net = tf.matmul(net, W["w8"])  
3 print("FC4 Output shape: ", net.shape)  
4 net = tf.add(net, B["b8"])  
5 tf.summary.histogram('W8', W["w8"])  
6 tf.summary.histogram('B8', B["b8"])  
7 # The below line of code adds an image to the Tensorboard.  
8 tf.summary.image('IMAGE', net)
```

Listing 4.3: Plotting Histograms on Tensorboard

4.2 Image Classification using TensorFlow

The Image Convolutional Neural Network (ICNN) is designed and trained to classify images like cars, traffic lights, and lanes.

4.2.0.1 Load Dataset into TensorFlow

This section deals in detail with the software implementation of CNN using TensorFlow and Python. This section also gives detailed account of steps to import the dataset into the model, perform convolution, max pooling, ReLu, fully connected layers in TensorFlow, the training API's used in the model and further discusses the differences in data representation in the TensorFlow and the Numpy model.

The images obtained from datasets are organized in the project directory as follows:

Dataset/Cars; Dataset/Lanes; Dataset/Pedestrians; Dataset/Lights;

A Python library called Open-cv is used to read the image from its location as shown in the code below:

For the data to be processed in TensorFlow it is required that the data be written to .tfrecord files or to csv files so that it is readable by the neural network. An array "addrs" that grabs the addresses of all the images using `glob.glob("path_to_directory")` is declared. Another global

```
1
2 def load_image(addr):
3     # read an image and resize to (64, 64)
4     # cv2 load images as BGR, convert it to RGB
5     img = cv2.imread(addr)
6     if img is None:
7         return None
8     img = cv2.resize(img, (64, 64), interpolation=cv2.INTER_CUBIC
9                       )
10    # returns a image of 64x64x3
11    img = cv2.cvtColor(img, cv2.COLOR_BGR2RGB)
12    return img
```

Listing 4.4: Loading Images into TensorFlow

variable to create a label according to the classifier is created. For example, in a classifier that has four classes like Cars, Lanes, Pedestrians, and Lights are labeled as 0, 1, 2 and 3 respectively. These global variables are used to create features of an image that is read by passing them through the "createDataRecord" function. This function has three arguments, they are, "out_file_name", address, and labels. A handle to the writer is initialized within the function. The for loop grabs the length of "addrs" array to get the total number of images that are to be read and written to the out file of tfrecord type. A load_image function is called to read the images at the given address using cv2 library from open-cv Python library. The image data is converted to a string and written to the out file. The writer is closed. Of all the images grabbed by array "addrs", 60% of the images are written to train.tfrecords, 20% of the images are written to val.tfrecords, and another 20% to test.tfrecords. The snippet of code to create data record is as follows:

The CreateDataRecord function is used to create the tfrecords that are later read into the CNN. The code in the snippet below uses shuffle() command that shuffles the data before creating the record. This is done to shuffle the training data for network to avoid regularization. When the network receives all images of one class followed by another (in other words arranged data) the network continues to learn features of car first and gradually when it starts to receive the data

```
1 def createDataRecord(out_filename , addr , labels):
2 # open the TFRecords file
3 writer = tf.python_io.TFRecordWriter(out_filename)
4 for i in range(len(addr)):
5     # Load the image 64x64x3
6     img = load_image(addr[i])
7     label = labels[i]
8     address = bytes(addr[i], 'utf-8')
9     if img is None:
10         continue
11     # Create a feature
12     feature = {
13         'image_raw': _bytes_feature(img.tostring()),
14         'label': _int64_feature(label)
15     }
16     # Create an example protocol buffer
17     example = tf.train.Example(features=tf.train.Features(
18         feature=feature))
19     # Serialize to string and write on the file
20     writer.write(example.SerializeToString())
21 writer.close()
sys.stdout.flush()
```

Listing 4.5: Creating tfrecords in TensorFlow

of the Lane class, it starts to learn patterns of lanes. In the process of learning features of lanes it forgets the features of cars that it has been trained for earlier. The network is regularized to learn certain features only. This problem necessitates the process of shuffling the training and evaluation data.

```
1 car_lane_train_path = 'train_dataset/*/*'
2 # read addresses and labels from the 'train' folder
3 addrs = glob.glob(car_lane_train_path)
4 labels = [0 if 'cars' in addr else 1 if 'lanes' in addr
5           else 1 if 'pedestrians' in addr else 3 for addr in addrs]
6 # 0 = Car, 1 = lanes, 2 = pedestrians, 3 = lights
7
8 # to shuffle data
9 c = list(zip(addrs, labels))
10 shuffle(c)
11 addrs, labels = zip(*c)
12 # Divide the data into 60% train, 20% validation, and 20% test
13
14 # Getting the training data
15 train_addrs = addrs[0:int(0.6 * len(addrs))]
16 train_labels = labels[0:int(0.6 * len(labels))]
17 # Getting the evaluation data
18 val_addrs = addrs[int(0.6 * len(addrs)):int(0.8 * len(addrs))]
19 val_labels = labels[int(0.6 * len(addrs)):int(0.8 * len(addrs))
20                  ]
21 # Getting the test data
22 test_addrs = addrs[int(0.8 * len(addrs)):]
23 test_labels = labels[int(0.8 * len(labels)):]
24
25 # creating TF records with corresponding datasets.
26 createDataRecord('train.tfrecords', train_addrs, train_labels)
27 createDataRecord('val.tfrecords', val_addrs, val_labels)
28 createDataRecord('test.tfrecords', test_addrs, test_labels)
```

Listing 4.6: TableGen Register Set Definitions

4.2.0.2 Convolutional Neural Network and Architecture

The CNN model in this project uses a combination of Alex Net and VGG Net architecture. Some of the prominent features of CNN are, convolutional layers followed by Max pooling layers and ReLu. Most of the CNN architectures like Alex Net and VGG Net have a series of successive fully connected layers followed by a few convolutional layers.

The image read into the network is resized to $64 \times 64 \times 3$ resolution. This allows us to have four convolutional layers and four fully connected layers. Based on the input image size and architecture all the kernel variables and bias variables are declared as trainable variables and initialized using random normalization. The $64 \times 64 \times 3$ image is resized and passed into a TensorFlow's API along with kernel, that performs convolution and outputs a feature map based on the number of Kernels.

The figure below, depicts the architecture of the CNN network built to classify images into three classes. In layer 1, also known as input layer, a 64×64 image is convolved by a $8 \times 3 \times 3 \times 3$ kernel. (The tensor representations are slightly different from numpy array representations, for example eight kernels of three channels each of size 3×3 in numpy is represented as $8 \times 3 \times 3 \times 3$, while in TensorFlow it is represented as $3 \times 3 \times 3 \times 8$). after convolution the bias value corresponding to each kernel is added and max pooling is performed. The ReLu non-linearity function is applied to the output feature maps of max-pooling layer. After applying ReLu function, the output of first layer i.e., of size $8 \times 31 \times 31$ is input to second layer. In second layer, a kernel of size $16 \times 8 \times 3 \times 3$ is passed in along with feature maps of size $8 \times 31 \times 31$ in to the `conv2d()` function of TensorFlow, that outputs feature maps of size $16 \times 29 \times 29$. Max pooling followed by ReLu operation is performed on these feature maps. This results in an output feature map of size $16 \times 14 \times 14$. The process repeats in layer three with a kernel size of $32 \times 16 \times 3 \times 3$. This results in a feature map of size $32 \times 6 \times 6$. The output from layer 3 is input to layer 4. In layer 4, the output from convolution is of size $64 \times 4 \times 4$ which results in having

about 1024 total elements. Hence, it is reshaped to 1024×1 to pass as an input to fully connected layer.

A convolutional layer is created in TensorFlow with the help TensorFlow API `tf.nn.conv2d()` as shown in the listing below.

```
1 def convolution_img(x, w, b, strides=1, name="conv"):  
2     with tf.name_scope(name):  
3         x = tf.nn.conv2d(x, w, strides=[1, strides, strides, 1],  
4             padding="VALID")  
5         x = tf.nn.bias_add(x, b) return tf.nn.relu(x)  
6     net = convolution_img(net, W["w1"], B["b1"], strides=1, name=  
7         "Conv_1")
```

Listing 4.7: Convolutional Layer in TensorFlow

The max-pool layer is created in TensorFlow using the API `tf.nn.max_pool()` as shown in the listing.

```
1 def max_pooling_layer(x, k=3, name="MaxPool"):  
2     with tf.name_scope(name):  
3         return tf.nn.max_pool(x, ksize=[1, k, k, 1], strides=[1, k, k,  
4             1], padding="VALID")
```

Listing 4.8: Max-pool layer in TensorFlow

To implement fully connected layer the kernel size should be determined. The kernel size in fully connected layer depends on the required number of outputs from that layer. The audio and image CNN networks have different architectures from one another. In the layer where it is transitioning from convolutional layer to fully connected layer, the input to the fully connected layer is flattened.

In ICNN, at layer 5 the fully connected network has an input tensor of size 1024×1 and requires only 512 neurons as output. This implies a kernel of size 1024×512 will be required. In layer 6 of ICNN, the fully connected layer has 512 input neurons that are output from layer 5

and 256 output neurons. In layer 7 of ICNN, there are 256 input neurons and 128 output neurons. Finally in layer 8 of ICNN there are 128 input neurons and 4 output neurons. The ICNN has four output neurons or output classes as it is classifying objects into four classes. This leads to weights of size 512×256 for layer 6, 256×128 for layer 7 and 128×4 for layer 8 of ICNN.

In ACNN, at layer 6 the fully connected network has an input tensor of size 1408×1 and requires only 512 neurons as output. This implies a kernel of size 1408×512 will be required. In layer 7 of ACNN, the fully connected layer has 512 input neurons that are output from layer 6 and 256 output neurons. In layer 7 of ACNN, there are 512 input neurons and 256 output neurons. Finally in layer 8 of ACNN there are 256 input neurons and 3 output neurons. The ACNN has three output neurons or output classes as it is classifying objects into three classes. This leads to weights of size 512×256 for layer 7 and 256×3 for layer 8 of ACNN.

A fully connected layer is achieved using the code below:

```
1 with tf.name_scope("FC_1"):  
2     net = tf.reshape(net, [-1, W["w5"].get_shape().as_list()[0]])  
3     net = tf.matmul(net, W["w5"])  
4     net = tf.add(net, B["b5"])  
5     net = tf.nn.relu(net)
```

Listing 4.9: Fully-Connected Layer in TensorFlow

The output from final layer is processed further by performing a Softmax activation function. This makes the prediction, as the class is determined by the highest value among the output neurons of the CNN. If the first neuron is greater than the other neurons in the output layer then the image passed into the model is predicted as the class that corresponds to first neuron. The Softmax function will be discussed later in this chapter.

4.3 Training

Now that the CNN model is built using TensorFlow, it requires training to classify images. TensorFlow provides high level API called Estimator class to easily train the model. This section will explain softmax function, working of TensorFlow's estimator, distribution and variation of weights during training.

The predictions of the model are drawn by performing softmax operation on the output neurons in final layer of the model. Softmax function is nothing but calculating the probability of events. It normalizes all the values such that the sum of all the output values are equal to one and range between (0,1). It is also known as normalized exponential function. Softmax function is given by the equation 4.1.

$$\sigma(z)_j = e^{z_j} / \sum_{k=1}^k e^{z_k} \text{ for } j = 1..k \quad (4.1)$$

In TensorFlow, softmax operation is performed using the below line of code:

```
1 y_pred = tf.nn.softmax(logits=logits)
```

Listing 4.10: Softmax in TensorFlow

The TensorFlow's estimator either trains, evaluates or performs inference based on the input argument. TensorFlow offers pre-made estimators or custom - made estimators. The estimator in TensorFlow's API is referred to as `tf.estimator.Estimator` class. The estimator not only trains the network but also logs training data into checkpoint file, so that it can recover training in the event of system failure. It also logs data-flow graph of the model in model directory created by estimator.

4.3.1 Adam Optimizer

Adam optimizer in TensorFlow utilizes Adam algorithm to train the network. The name Adam has come from Adaptive moment estimation. The Adam algorithm is a combination of Adaptive gradient algorithm and Root Mean Square Propagation, commonly known as RMSProp algorithm.

In the Gradient descent algorithm, the loss function $J(\theta)$ parameterized by a model's parameters is minimized by updating the parameters in opposite direction of the gradient of the function $\nabla_{\theta}J(\theta)$. The learning rate determines the step size to move towards a local minima. There are three gradient descent variants, Batch gradient descent, Stochastic gradient descent, and Mini-batch gradient descent. In Batch gradient descent, the gradient of cost function over entire dataset with respect to its parameters ϑ is computed. Since the gradient is computed for entire dataset it can induce significant amount of latency in updating the weights for each epoch. In stochastic gradient descent, gradient of cost function with respect to its parameters is computed and updated for each image at a time, this process can be faster and more efficient than the previous algorithm. In mini - batch gradient descent, a small number of images from a dataset are passed into the model to compute gradient of cost function and update weights for the same. This can be advantageous in a way, but it can be challenging to derive a correct learning rate for this model. As too small of a learning rate can lead to an extremely gradual convergence, while too large of a learning rate can cause the loss function to fluctuate and result in inconsistent network output. There are several gradient descent optimization algorithms like Momentum, Adagrad, Adadelata, RMSprop, Adam, etc.

In the momentum algorithm a perception of a rolling ball over the hill is considered. The ball rolling down the hill gains acceleration in the direction of slope while it gains resistance when the direction changes. Similarly in momentum algorithm the momentum term increases for dimensions whose gradients point in the same direction and reduces updates for dimensions

whose gradients change directions [38]. The algorithm is guided by following equations 4.2 and 4.3.

$$v_t = \gamma v_{t-1} + \theta \nabla_{\theta} J(\theta) \quad (4.2)$$

$$\theta = \theta - v_t \quad (4.3)$$

The momentum term γ is usually set to 0.9.

The RMSprop algorithm is similar to Adagrad and Adadelta algorithms. These algorithms adapt the learning rate to parameters, performing larger updates for infrequent and smaller updates for frequent parameters [38]. Adadelta is an improvement on Adagrad, where it limits the number of previously accumulated squared gradients to a fixed window size 'w'. RMSprop algorithm is guided by the following equations 4.4 and 4.5.

$$E[g^2]_t = 0.9E[g^2]_{t-1} + 0.1g_t^2 \quad (4.4)$$

$$\theta_{t+1} = \theta_t - \eta g_t / \sqrt{E[g^2]_t + \epsilon} \quad (4.5)$$

In Adam algorithm, adaptive learning rates are computed for each parameter. The algorithm stores the exponentially decaying average of past gradients v_t and m_t identical to RMSprop and momentum algorithms respectively. The gradients v_t and m_t are represented by equations 4.6 and 4.7 respectively.

$$v_t = \beta_2 m_{t-1} + (1 - \beta_2) g_t^2 \quad (4.6)$$

$$m_t = \beta_1 m_{t-1} + (1 - \beta_1) g_t \quad (4.7)$$

Bias corrected first and second moment estimates are given by the equations 4.8 and 4.9 respectively.

$$v_t = v_t / (1 - \beta_2^t) \quad (4.8)$$

$$m_t = m_t / (1 - \beta_1^t) \quad (4.9)$$

Adam's update rule is mathematically represented by the equation 4.10.

$$\theta_{t+1} = \theta_t - \eta m_t / \sqrt{v_t} + \epsilon \quad (4.10)$$

4.3.2 Loss and Accuracy

The model was run for 1274 global steps and was trained on a data set containing 790 images that comprises 230 images from each class. The loss was observed to improve consistently as the plot in Fig. 4.1 depicts. The model classifies images into three different classes with an accuracy that ranges between 99.28% to 100%. The Fig. 4.2 depicts a plot of improving accuracy of the network with increasing number of training steps. It can be observed from the plots that the accuracy is increasing with reducing loss.

4.3.3 Tensorboard Output

A complete flow of the model created in TensorFlow is depicted in the graph as shown in Fig. 4.3, (derived from Tensorboard) while the Fig. 4.4 depicts the flow chart for image inference

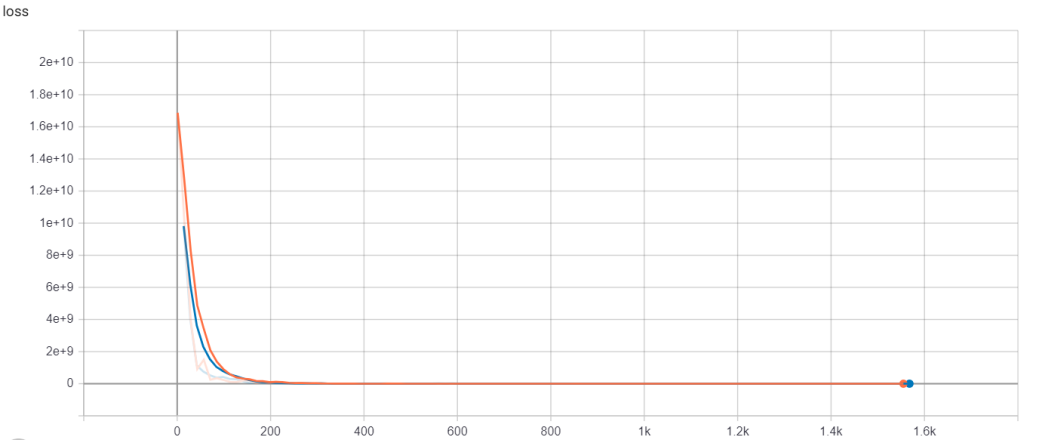


Figure 4.1: Loss of Image Convolutional Neural Network TensorFlow Model

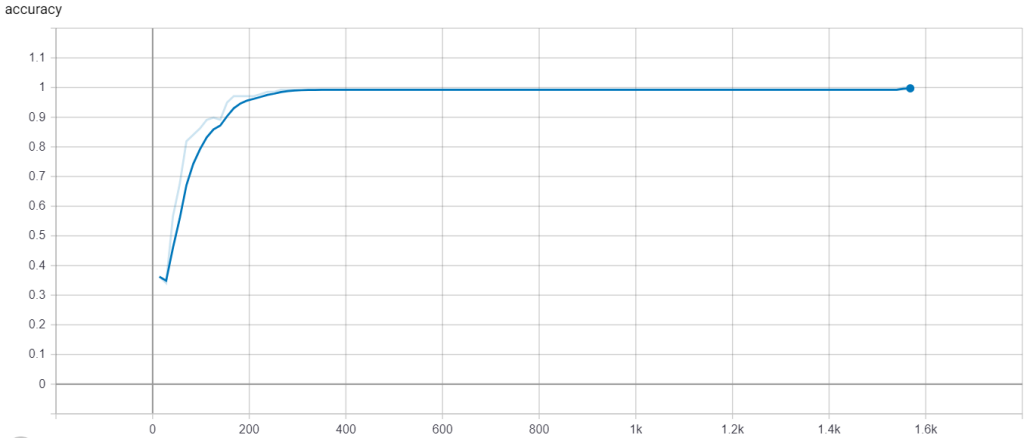


Figure 4.2: Accuracy of Image Convolutional Neural Network TensorFlow Model

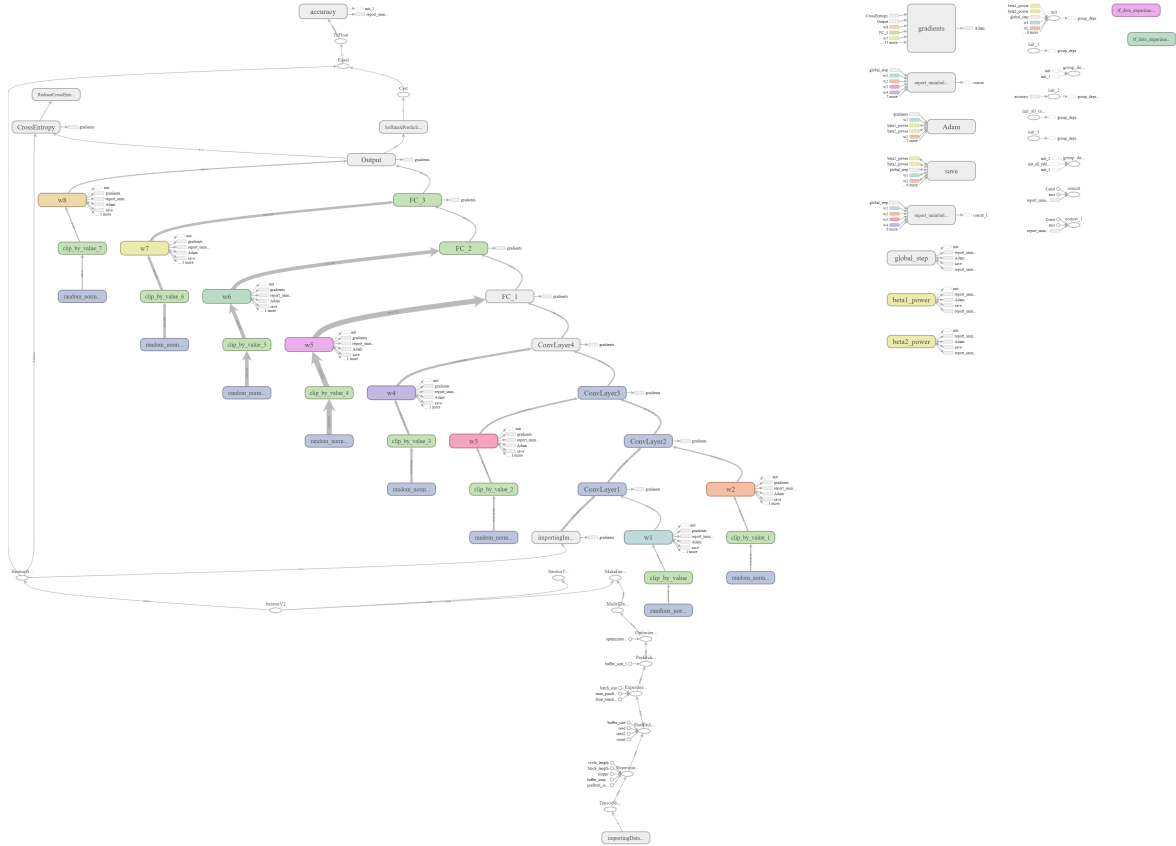


Figure 4.3: Image CNN Data flow Graph - Tensorboard

CNN model..

A summary of weights across all the training steps for all the layers of Image CNN model are plotted as Histograms on Tensorboard. It can be observed from figures (Fig. 4.5, 4.6, 4.7, 4.8, 4.9, 4.10, 4.11, 4.12) that randomly initialized network weights are driven in every layer with increasing global training steps to drive the loss to zero percent and accuracy to hundred percent.

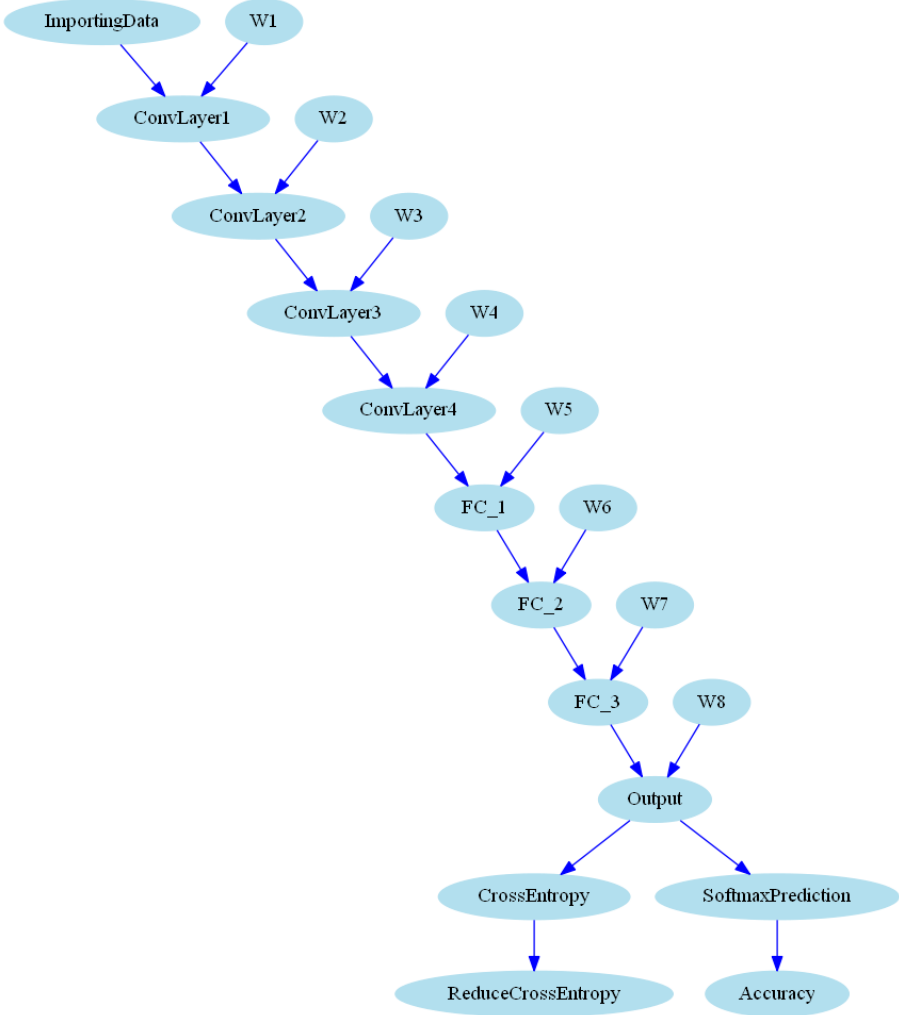


Figure 4.4: Image CNN Inference Model Graph

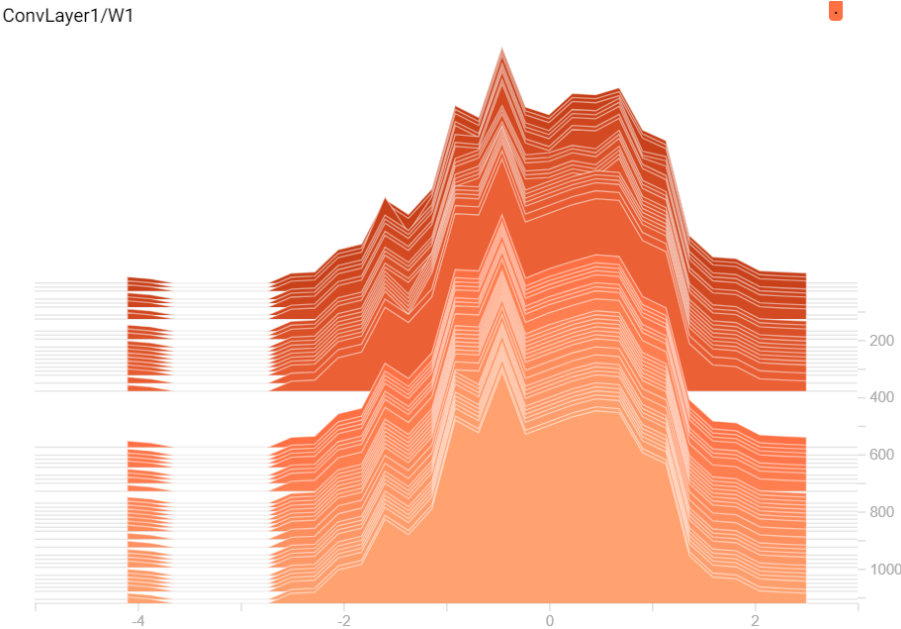


Figure 4.5: Image CNN Weights - Layer 1

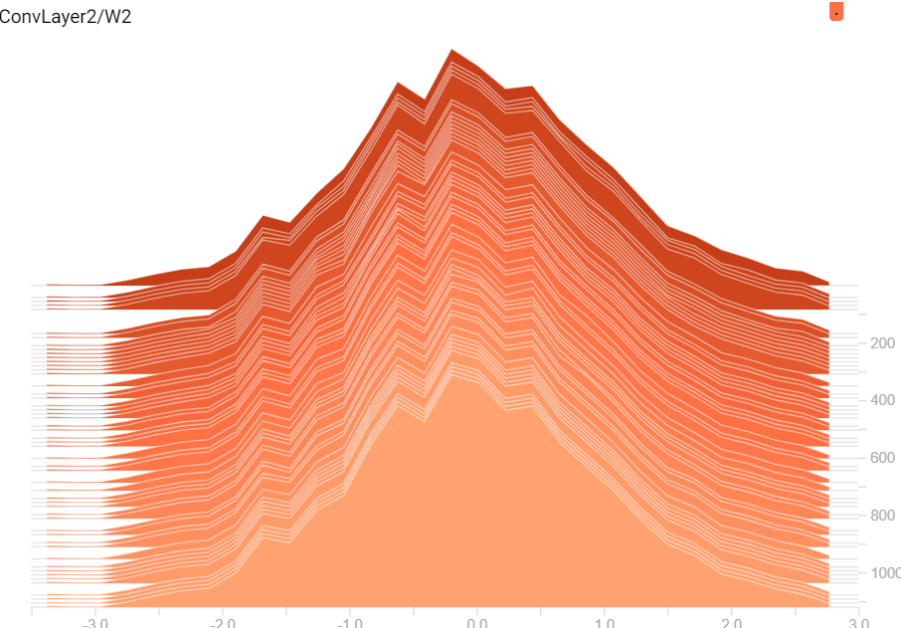


Figure 4.6: Image CNN Weights - Layer 2

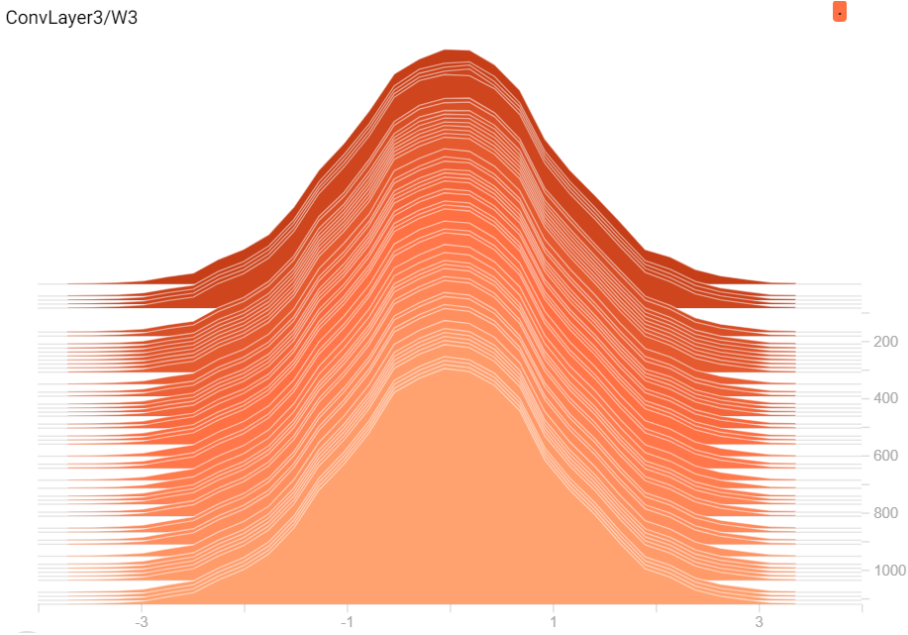


Figure 4.7: Image CNN Weights - Layer 3

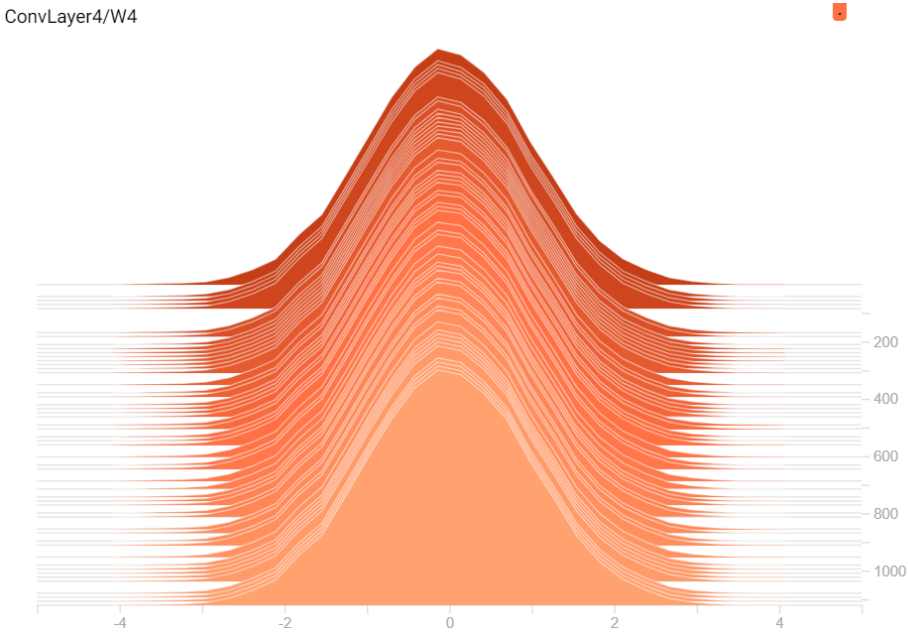


Figure 4.8: Image CNN Weights - Layer 4

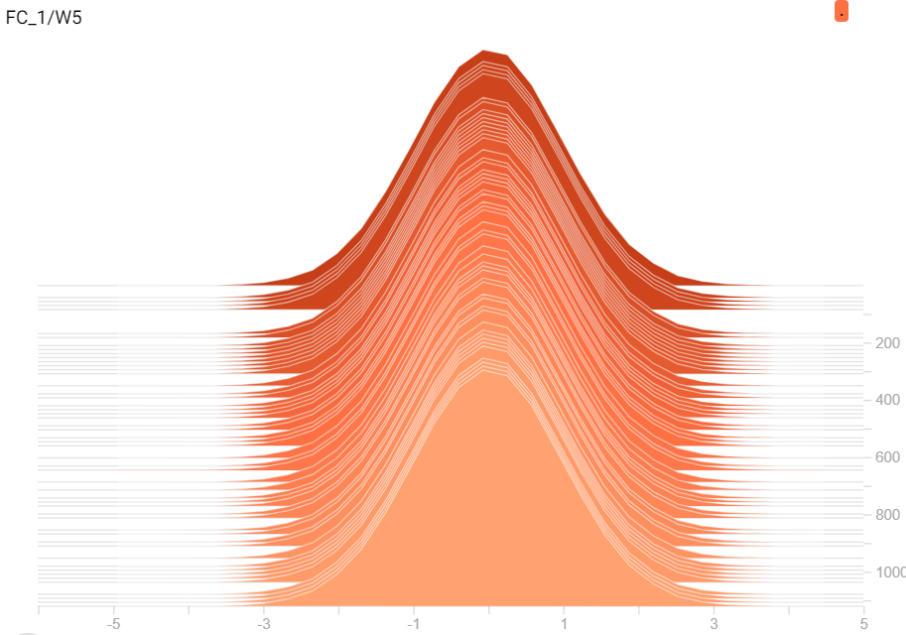


Figure 4.9: Image CNN Weights - Layer 5

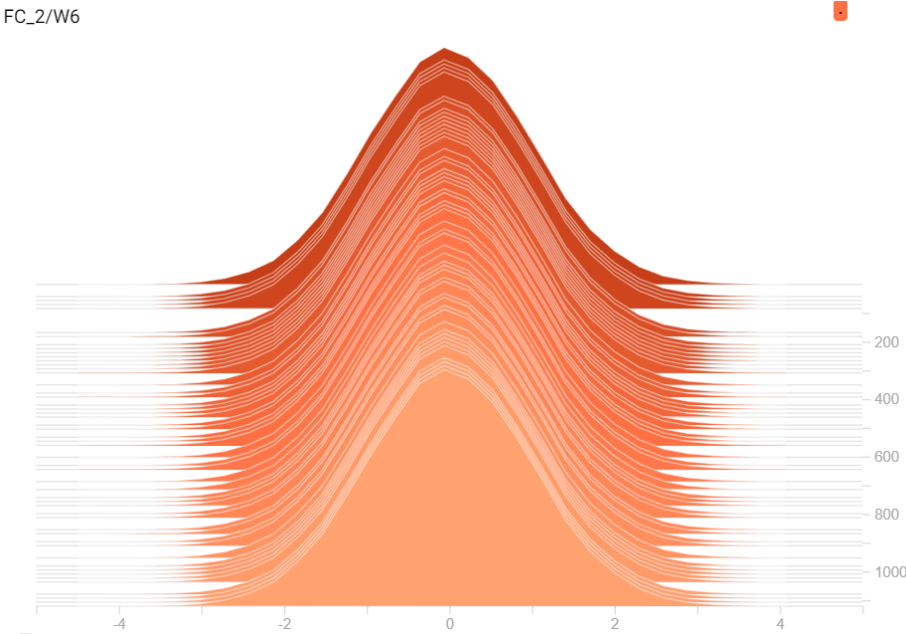


Figure 4.10: Image CNN Weights - Layer 6

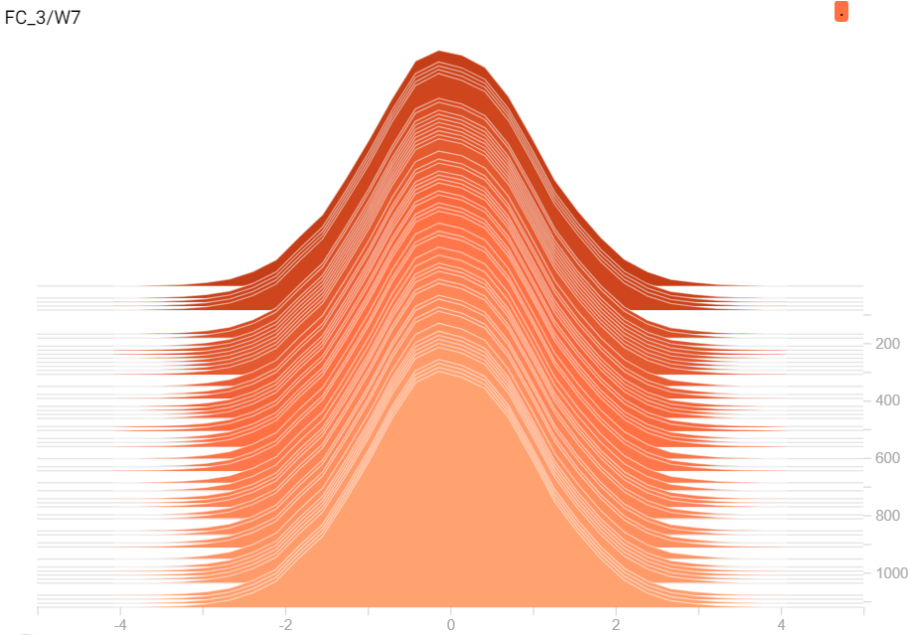


Figure 4.11: Image CNN Weights - Layer 7

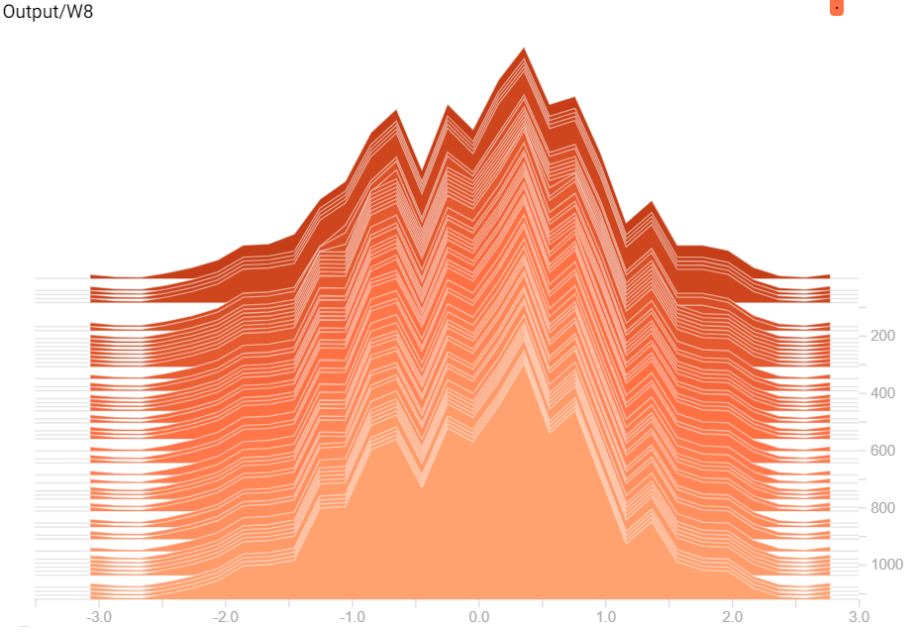


Figure 4.12: Image CNN Weights - Layer 8

4.4 Audio Classification using TensorFlow

The Audio Convolutional Neural Network (ACNN) neural network was trained on dataset taken from urban sound classification dataset. The convolutional neural network was implemented to classify sounds into three classes, they are sounds of sirens, car horn and children playing. The sound files are all in “.wav” format. All the audio wav files are put together in a single directory, and are labeled according to their respective classes in a spreadsheet. Based on the mean of the sum of lengths of these audio files in each class, the distribution pi chart is plotted. The distribution of data in each class is shown in Fig. 4.13. A few operations like FFT, Filter Banks, MFCC's and, envelope are performed on the wav files to extract features as shown in Fig. 4.14, Fig. 4.15, Fig. 4.16 and Fig. 4.17 respectively. The envelope function is applied to all the audio files to clean them off the redundant data or noise.

The distribution of data in each class after applying the envelope function is as shown in Fig. 4.18. The percentage of the data distributed in each class differs slightly from the distributions shown in Fig. 4.13. Similarly slight differences can be observed from Fig. 4.19, 4.20, and 4.21 in the FFT, filter bank and MFCC's respectively, after applying the envelope function.

4.4.1 Mel Frequency Cepstral Coefficients (MFCC)

The first step in any audio recognition system is to extract significant features from the audio files and to suppress the background noise that will help in recognizing the sound of interest from audio signal.

To perform MFCCs on the audio signal, the signal needs to be sliced into smaller windows. The smaller the window size the better the similarity in statistical information of the sample. The sample signal values vary from sample to sample. The signals are usually framed into 20-40 ms. A smaller size than that might result in oversampling and a larger signal will result

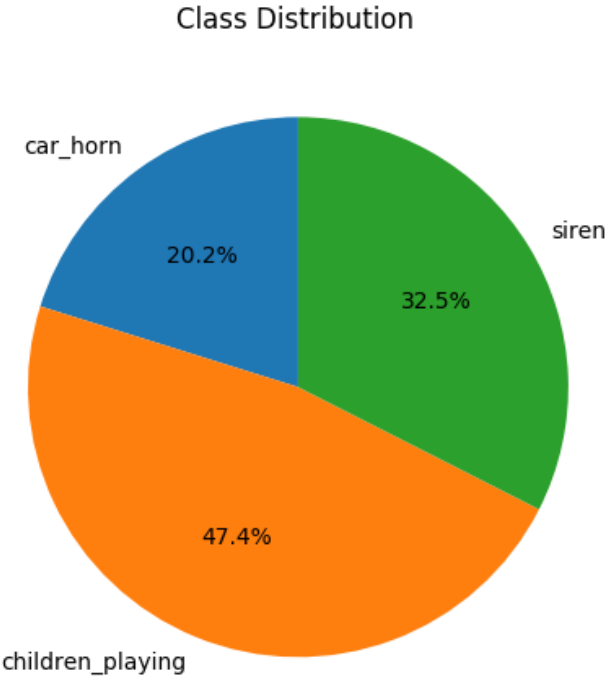


Figure 4.13: Distribution

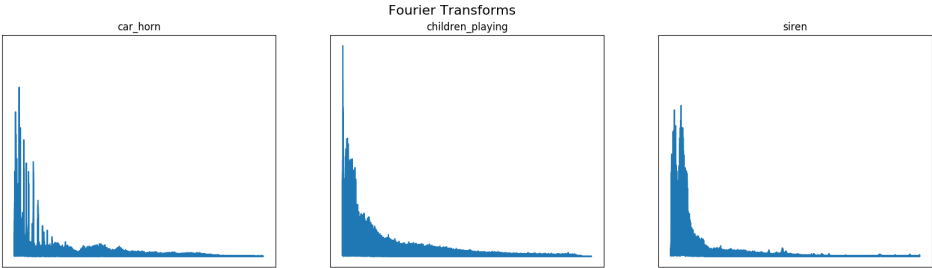


Figure 4.14: FFT

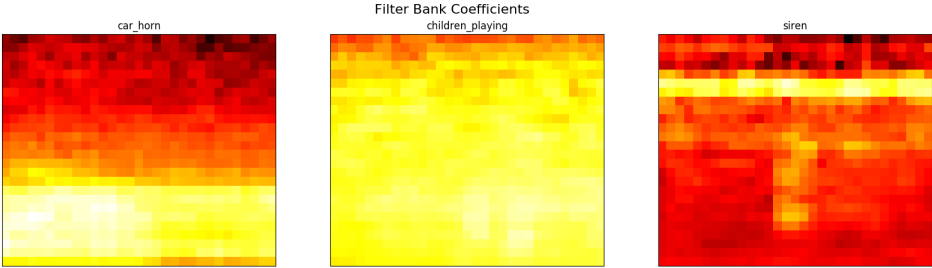


Figure 4.15: Fbank

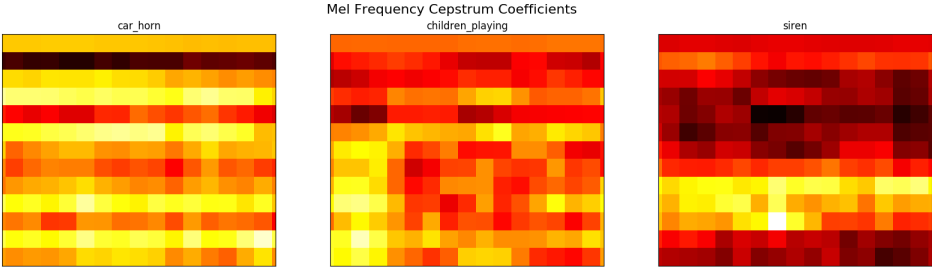


Figure 4.16: MFCCs

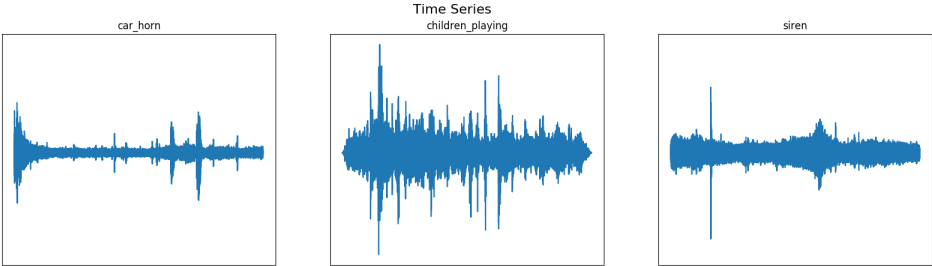


Figure 4.17: Envelope Signal

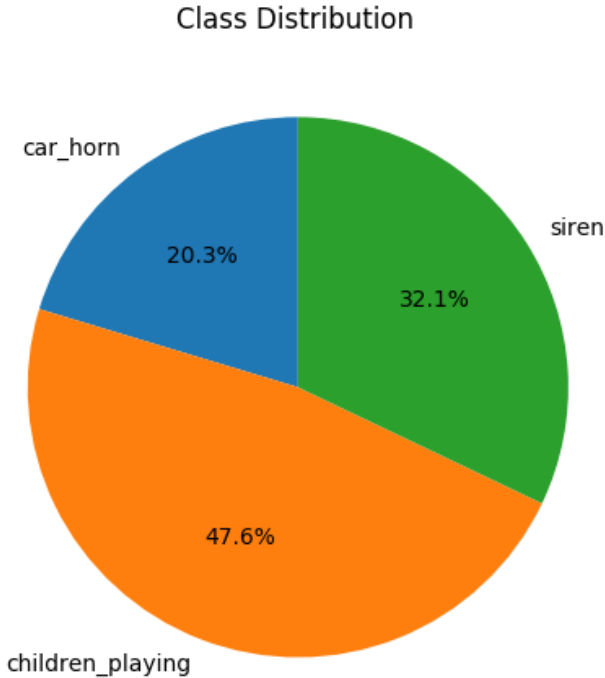


Figure 4.18: Distribution-Envelope

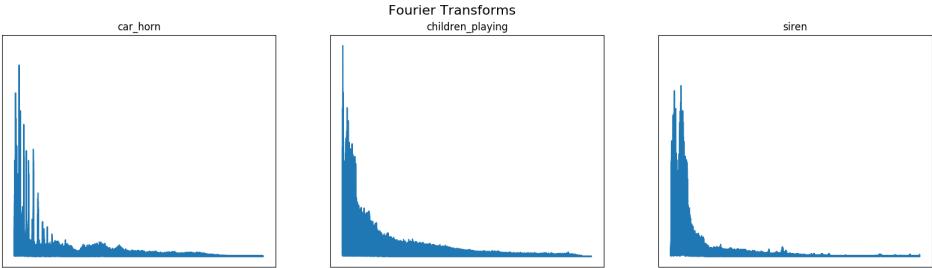


Figure 4.19: FFT Envelope

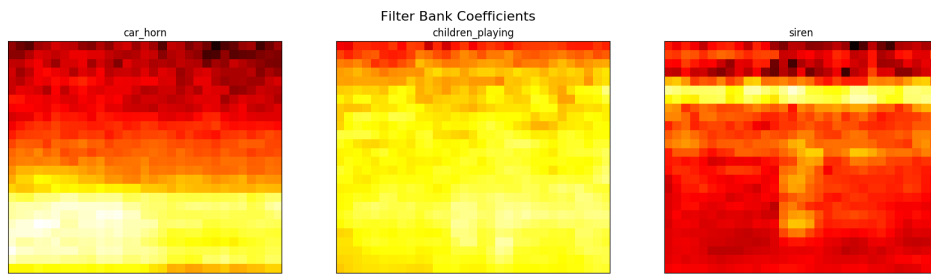


Figure 4.20: Fbank

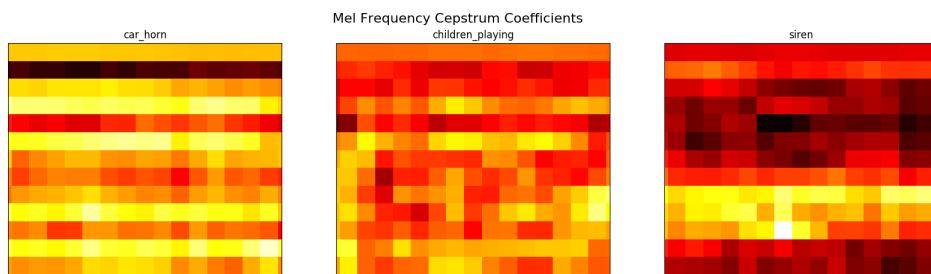


Figure 4.21: MFFCs

in a drastic change in sample values within one sample. A power spectrum for each of these frames is computed. To reduce the occurrence of closely spaced frequency signals, clumps of periodogram bins are taken and summed up to get a more pronounced effect that shows how much energy exists in different frequency regions. In Mel filter bank, the filters are narrower for low frequency signals and get wider for higher frequency signals. This gives a rough estimate on energy present at each spot. The Mel scale helps with the spacing of filter banks and the required width of filters. Upon calculating the energies, a logarithm is applied followed by a Discrete Fourier Transform to all of them.

Mel scale relates the frequency perceived by algorithm to the signal's actual frequency making the features match closely to what humans hear. the equation 4.11, converts normal frequency

to mel scale.

$$M(f) = 1125 \ln(1 + f/700) \quad (4.11)$$

To convert Mel scale to normal frequency the equation 4.12 is used.

$$M^{-1}(m) = 700(\exp(m/1125) - 1) \quad (4.12)$$

The filter bank energies are applied to audio files from each class as shown in Fig. 4.15. Mel Frequency Cepstral Coefficients are applied to audio files as shown in Fig. 4.16. As can be seen, the filter bank energies look dense compared to MFCC's.

4.4.2 Network Architecture

A network with eight layers is chosen which comprises of four convolutional layers, one max pool layer and, four fully connected layers with ReLu non linearity function applied after every layer. The Softmax activation function is applied to the output layer for network predictions. The current network structure was attained by a trial and error method. The process of importing data, dataset organization, network architecture and results were discussed succinctly as given below.

Data Organization

All the audio files from different classes are labeled as shown in Table 4.1 using a Python script. This data will be read as a database using Pandas library in Python. The file name labeled as fname in the spreadsheet is considered as index and the class name is the label. The file name read from the spread sheet is looked up in the folder where all the wave files are placed.

```
1 df = pd.read_csv('urbansound_v1.csv')
2 df.set_index('fname', inplace=True)
3 for f in df.index:
4     rate, signal = wavfile.read('clean_v1/'+f)
5     df.at[f, 'length'] = signal.shape[0]/rate
6
7 classes = list(np.unique(df.label))
8 class_dist = df.groupby(['label'])['length'].mean()
9 class_length = df.groupby(['label'])['length'].sum()
10 prob_dist = class_dist/class_dist.sum()
11 fig, ax = plt.subplots()
12 ax.set_title('Class Distribution', y=1.08)
13 ax.pie(class_dist, labels=class_dist.index, autopct='%1.1f%%',
14        shadow=False, startangle=90)
15 plt.savefig('figure/distribution_masked.png')
16 plt.show()
```

Listing 4.11: Audio Dataset Organization

The above snippet of code imports the data from from spread sheet and computes a probability distribution per class based on the mean of sum of the lengths of audio files in the dataset.

All the files in the dataset folder are cleaned using the envelope function and written to a directory named “clean”. To extract features from wav files for the network to be able to classify wav files into classes, Mel Frequency Cepstral Coefficients is applied to the wav files in “clean” directory.

Import Dataset

The `get_audio_data()` function (shown in the code below) slices all the data into one second slices. The one second slices are fed into the `mfccs` python library function. This performs MFCC on each slice and outputs a numpy array of size `NUMFRAMES` by `numcep`. In this case, `NUMFRAMES` is equal to 99 and `numcep` is equal to 13. A transpose of this matrix will result

Table 4.1: Labeled Audio Dataset Organization

S. No	fname	Label
1	2937.wav	car_horn
2	9223.wav	children_playing
3	16772.wav	siren
4	17009.wav	children_playing
5	17074.wav	car_horn
6	17124.wav	car_horn
7	17307.wav	car_horn
8	17973.wav	children_playing
9	18933.wav	car_horn
10	19026.wav	siren
11	22601.wav	siren
12	24074.wav	car_horn
13	24076.wav	children_playing
14	24077.wav	children_playing
15	26173.wav	siren

in a 13x99 matrix. This numpy matrix is appended to a list X. If the input wav file is less than a second then it enters the else condition and appends zeros around the signal to resize it to 13x99 matrix and appends it to the list X. In parallel the label of each input sample is appended to a list y. At the end, the two lists of equal sizes X and y are converted to NumPy arrays and returned by the function. At this point all the input dataset is imported as numpy array.

Now if the model is run for the data in X and y arrays as it is then the model will learn to classify one class and the accuracy increases with reducing loss. When it encounters a new sample from a different class the accuracy falls drastically and in the process of learning to classify current class samples, the weights learned by the model for previous class are diminished. This is why it is important to shuffle input data. The below snippet of code shuffles X and y arrays keeping the corresponding indices aligned.

```
1 def load_audio(directory = "../dataset/"):
2     # grabs all the audio files located in the given path
3     addrs = glob.glob(directory+'**/*')
4     # create a list to append processed audio data and
5     # corresponding label.
6     X = []
7     y = [0 if 'car_horn' in addr else 1 if 'children_playing' in
8         addr else 2 for addr in addrs]
9     # variable to hold maximum and minimum values
10    # in input data.
11    _min, _max = float('inf'), -float('inf')
12    numsamples = len(addrs)
13    for addr in addrs:
14        # reads the wavfiles as a numpy array.
15        rate, wav = wavfile.read(addr)
16        # check if the file is a stereo track.
17        if wav.shape[1] > 1:
18            wav = wav.astype(np.float32)
19            # Perform mean of the values to transform to mono track.
20            wav = (wav[:, 0] + wav[:, 1]) / 2
21            # performing MFCC's on input audio results in a 13x99
22            # numpy matrix
23            X_sample = mfcc(wav, rate, numcep=config.nfeat, nfilt=
24                config.nfilt, nfft=config.nfft).T
25            _min = min(np.min(X_sample), _min)
26            _max = max(np.max(X_sample), _max)
27            # append a 13x99 numpy matrix to training data.
28            X.append(X_sample)
29    X = np.array(X)
30    y = np.array(y)
31    X = (X - _min) / (_max - _min)
32    X = X.reshape(X.shape[0], X.shape[1], X.shape[2], 1)
33    X = X.astype(np.float32)
34    return X, y
```

Listing 4.12: Import Audio Dataset

```
1 # Randomize the input data and
2 # corresponding labels in a similar way.
3 randomize = np.arange(len(X))
4 np.random.shuffle(randomize)
5 X = X[randomize]
6 y = y[randomize]
```

Listing 4.13: Shuffle Audio Dataset

Network Architecture

The arrays with wav file sample data and corresponding labels are shuffled and imported into estimator in TensorFlow as shown below. A 60 percent of the dataset is used for training the model, the remaining 40 percent is split into halves for evaluation and prediction of the model. The dataset and labels array from indices zero to 60% of total number of samples is passed into `input_fn` with a batch size of 40 and number of epochs as 10. Number of epochs means number of times the given dataset has to be passed in. The batch size indicates the number of samples that will be passed into model at once. The global step indicates number of batches the model sees before the entire dataset is complete. For example the model was run for 67980 global steps with an increment of 330 global steps for each iteration. This implies that 40×330 samples were passed into the model for 330 times with a batch size of 40 each time per iteration. The model was run for $67980 / 330 = 206$ times for the entire training dataset.

The input layer or layer one has an input of 13×99 matrix that is convolved with a kernel of dimensions $3 \times 3 \times 1 \times 8$. The kernel dimensions $3 \times 3 \times 1 \times 8$ imply eight kernels of size 3×3 with one channel. The convolution in first layer results in a 8 channel feature map of size 11×97 which is added by a bias and passed on to ReLu non-Linearity function. The output from first layer is passed as input into layer two. The input in layer two is convolved with a kernel of size of $16 \times 8 \times 3 \times 3$ which means 16 kernels of size 3×3 with 8 channels to match with 8 channels in the input. The output of convolution is added by bias values and a ReLu non linearity function is

```
1 # feeding input data that is in the form
2 # of numpy matrices , to estimator model
3 # Training dataset
4 input_fn=tf.estimator.inputs.numpy_input_fn({"wav":X[0:int(0.6*
      len(X))]}),
5     y=y[0:int(0.6* len(y))], batch_size=40, shuffle=True ,
6     num_epochs=10)
7 # Evaluation Dataset
8 eval_fn=tf.estimator.inputs.numpy_input_fn({"wav":X[int(0.6* len
      (X)):int(0.8* len(X))]}),
9     y=y[int(0.6* len(y)) :int(0.8 * len(y))], batch_size=40,
10    shuffle=True , num_epochs=1)
11 # Testing Dataset
12 test_fn=tf.estimator.inputs.numpy_input_fn({"wav":X[int(0.8 *
      len(X)):]}), y=y[int(0.8 * len(y)):],
13     batch_size=40, shuffle=False , num_epochs=10)
14
15 count = 0
16 while count < 1000:
17     model.train(input_fn=input_fn , steps=1000)
18     result = model.evaluate(input_fn=eval_fn , checkpoint_path=
19         None)
20     print(result)
21     print("Classification accuracy:{0:.2%}" .format(result["
22         accuracy"]))
23     sys.stdout.flush()
24     count = count + 1
25 model.predict(input_fn=test_fn)
```

Listing 4.14: Distribution of Dataset

applied. The output from layer two of size $16 \times 9 \times 95$ is passed on to layer three and is convolved with a filter size of $32 \times 16 \times 3 \times 3$. The resulting output of size $32 \times 7 \times 93$ from layer three is input to layer four with a convolving kernel of size $64 \times 32 \times 3 \times 3$. The output from layer four of size $64 \times 5 \times 91$ is passed on to a max pooling layer after performing ReLu. The output from Max Pooling layer of size $64 \times 2 \times 45$ is flattened to be input to fully connected layers.

The flattened output is of the size 5760×1 and is input to a fully connected layer as shown in Fig. 5.4. The hidden layer, layer 5 has 5760 input neurons and 1024 output neurons with mapping weights of size 5760×1024 and bias values of size 1024×1 . In layer six, there are 1024 input neurons and 512 output neurons with mapping weights of size 1024×512 and bias of size 512×1 . In layer seven there are 512 input neurons and 256 output neurons with a mapping function of size 512×256 and a bias of size 256. In the output layer, i.e., layer 8 there are 256 input neurons and three output neurons with a mapping function of size 256×3 and bias values of size 3×1 . The Softmax function is chosen as an activation function for predictions. An `argmax` numpy/TensorFlow function is applied to get the model predictions.

Training & Results

The Adam Optimizer from TensorFlow's API is used for training the audio CNN model. The cross entropy is calculated using the `sparse_softmax_cross_entropy_with_logits()` from the TensorFlow's API. The actual labels and the model's predictions known as logits are passed as argument to the function to compute cross entropy. The cross entropy computed is passed as an input argument to a `reduce_mean()` function from TensorFlow's API to compute loss. The loss computed in previous step is passed into `minimize()` function to minimize the loss of the model.

Initially a seven layer model was designed and run for 12000 global steps without shuffling the data. The results are depicted in Fig. 4.22. As it can be observed the loss is reducing for a while and rises. This is because the model has been learning weights for one class and these

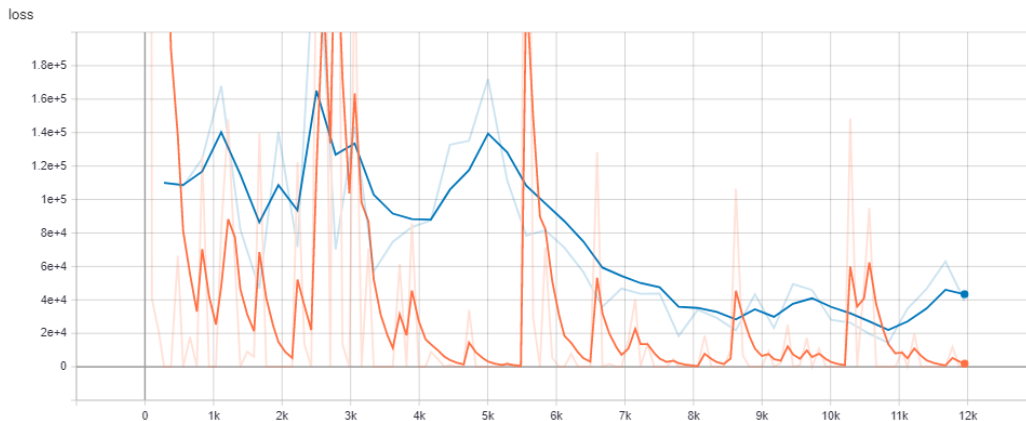


Figure 4.22: Loss of the Audio CNN Model

weights diminish when the model is learning to classify another class.

From Fig. 4.23 it is evident that the model has been learning to classify dataset into two classes and has got very few samples that fell into the second class although the model has been receiving different samples from each class. This is how it was identified that when the input dataset is not shuffled the CNN results in having diminishing network weights problem. Later, to improve the model's accuracy and loss, another hidden layer was added and an eight layer model was designed with shuffled dataset during training.

The eight layer model showed promising results at the beginning of training global steps and was run for about eight hours to give an accuracy of 98.55%. The audio CNN model's loss and accuracy plots are shown in Fig. 4.24 and Fig. 4.25 respectively.

The graph obtained from Tensorboard for the TensorFlow Audio CNN model is shown in Fig. 4.26, while the Fig. 4.27 depicts the flow chart for audio inference CNN model.

A summary of weights across all the training steps for all the layers of audio CNN model are plotted as Histograms on Tensorboard as shown in figures (Fig. 4.28, 4.29, 4.30, 4.31, 4.32, 4.33, 4.34, and 4.35).

The Softmax predictions of the model are shown in the Fig. 4.36. The audio files from all

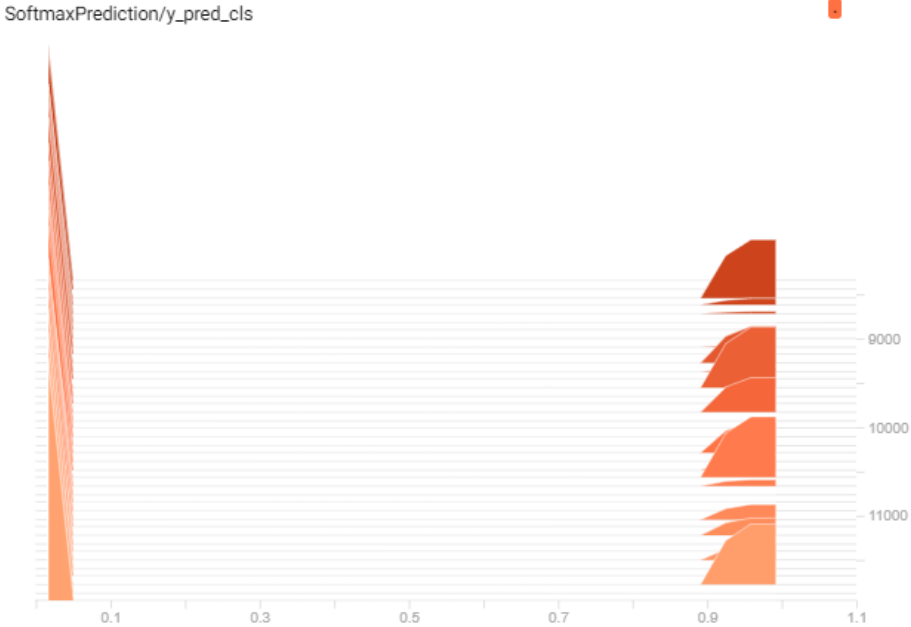


Figure 4.23: Audio CNN Model Predictions Histogram

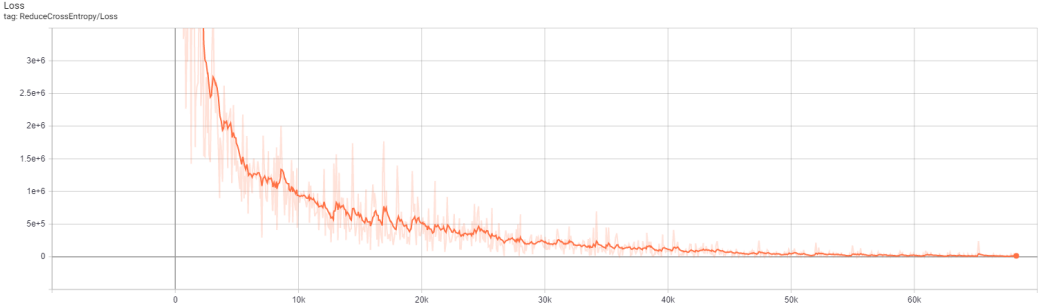


Figure 4.24: Audio CNN Model Loss

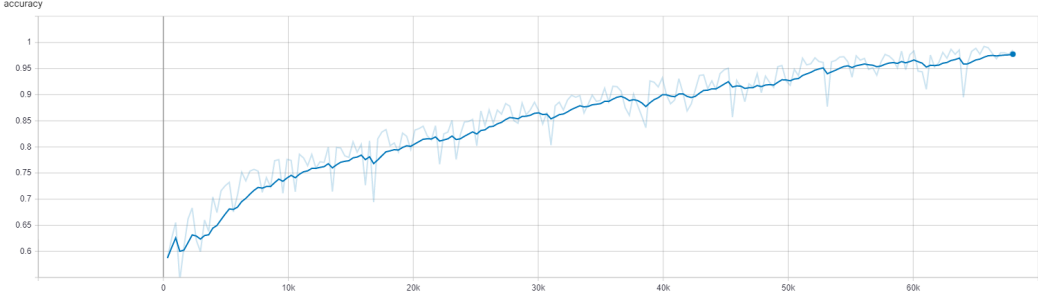


Figure 4.25: Audio CNN Model Accuracy

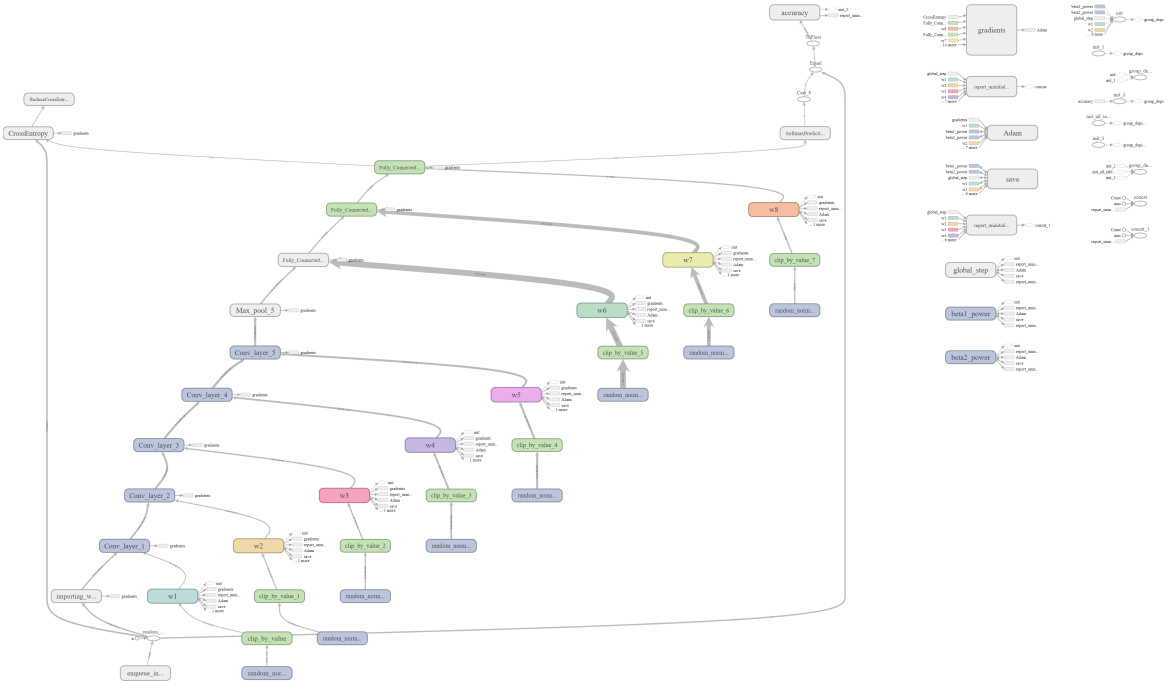


Figure 4.26: Audio CNN Model Graph - Tensorboard

three different classes were classified as inferred from this figure.

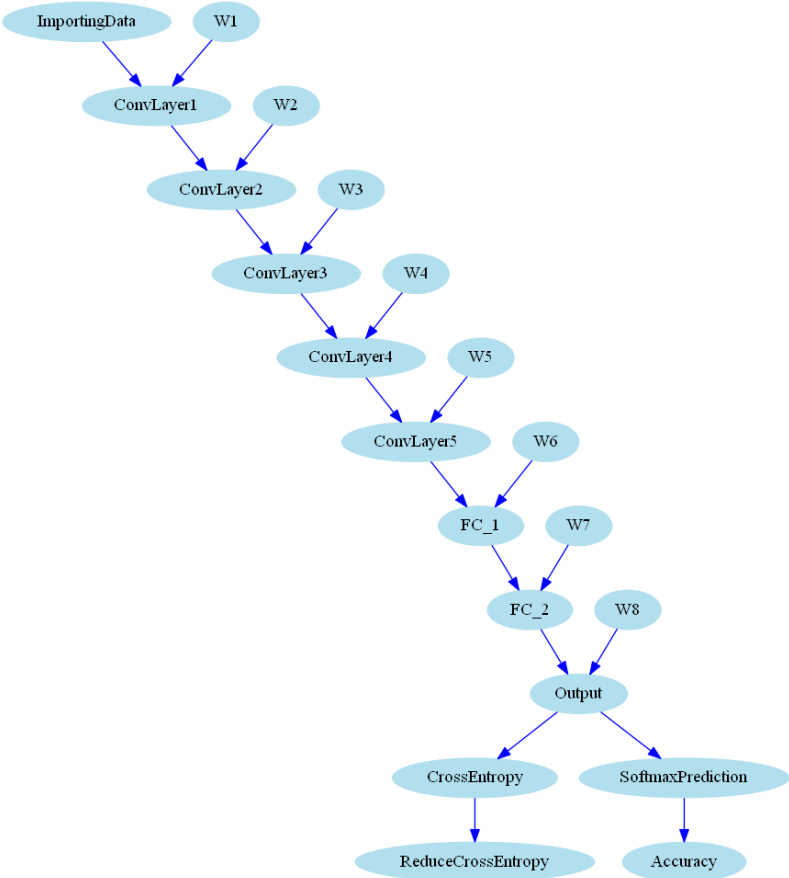


Figure 4.27: Audio Inference CNN Model Graph

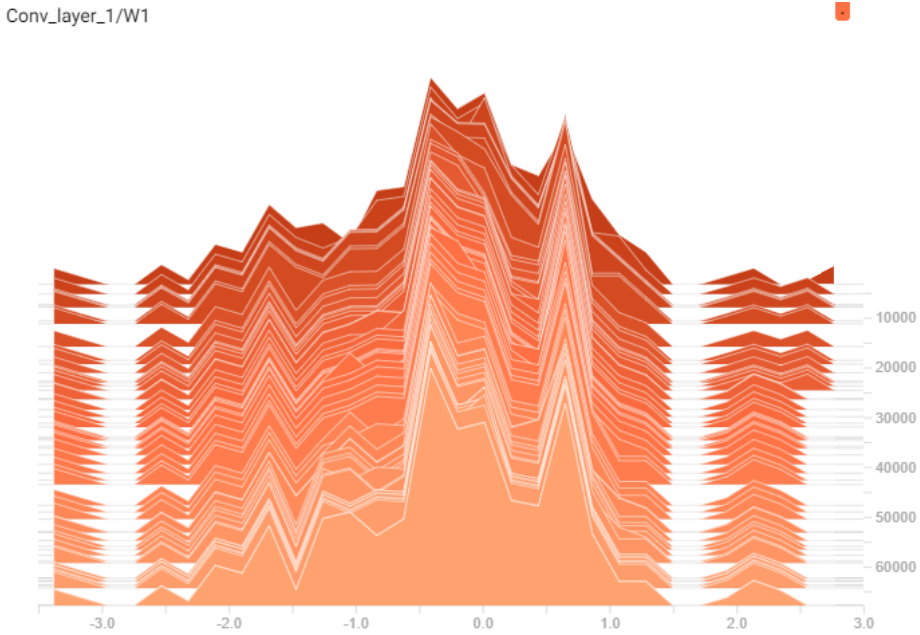


Figure 4.28: Audio CNN Weights - Layer 1

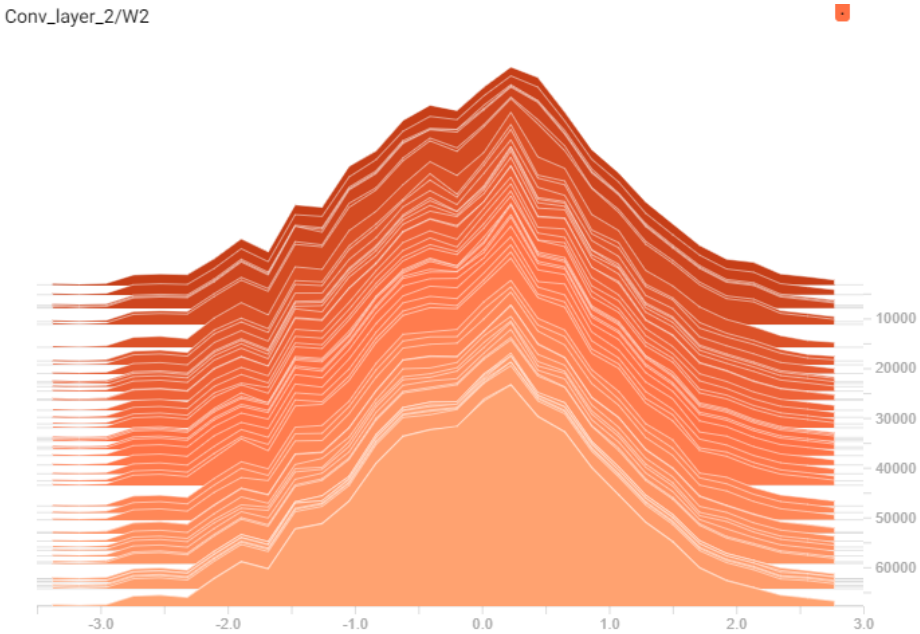


Figure 4.29: Audio CNN Weights - Layer 2

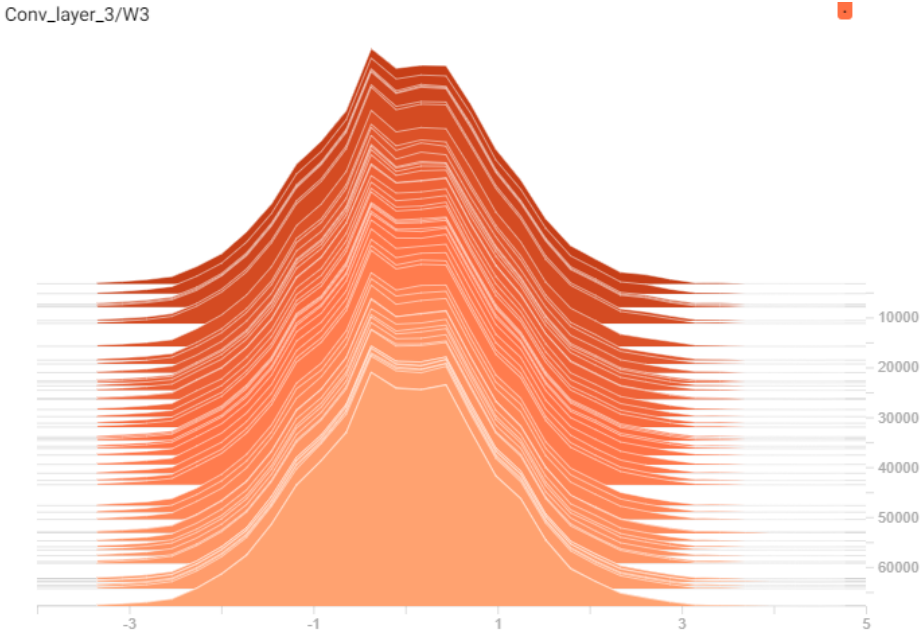


Figure 4.30: Audio CNN Weights - Layer 3

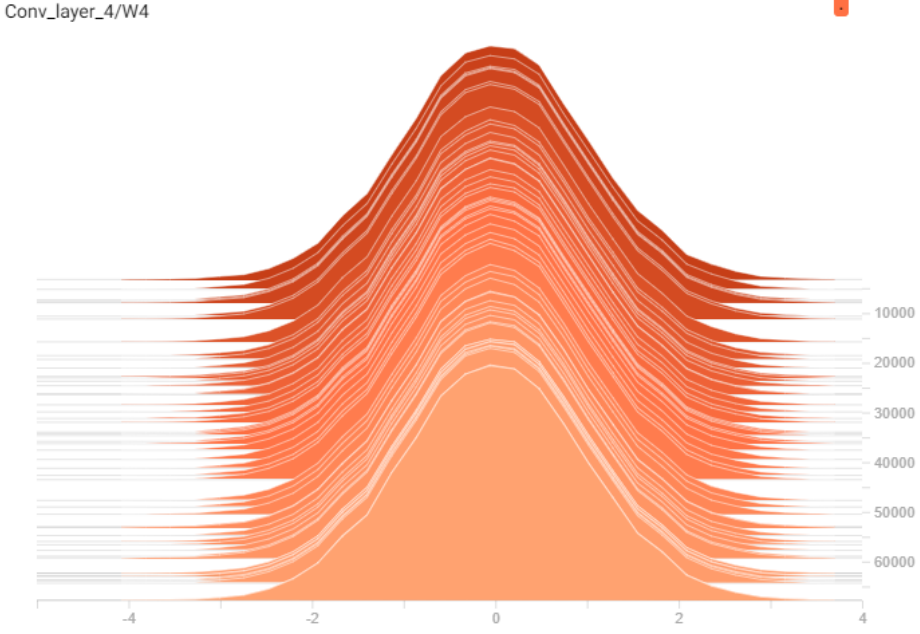


Figure 4.31: Audio CNN Weights - Layer 4

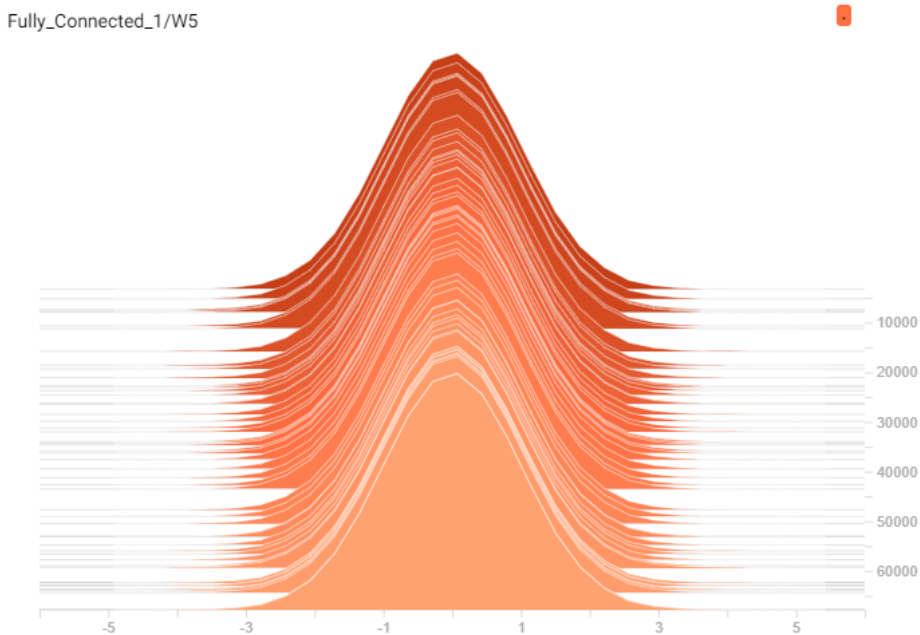


Figure 4.32: Audio CNN Weights - Layer 5

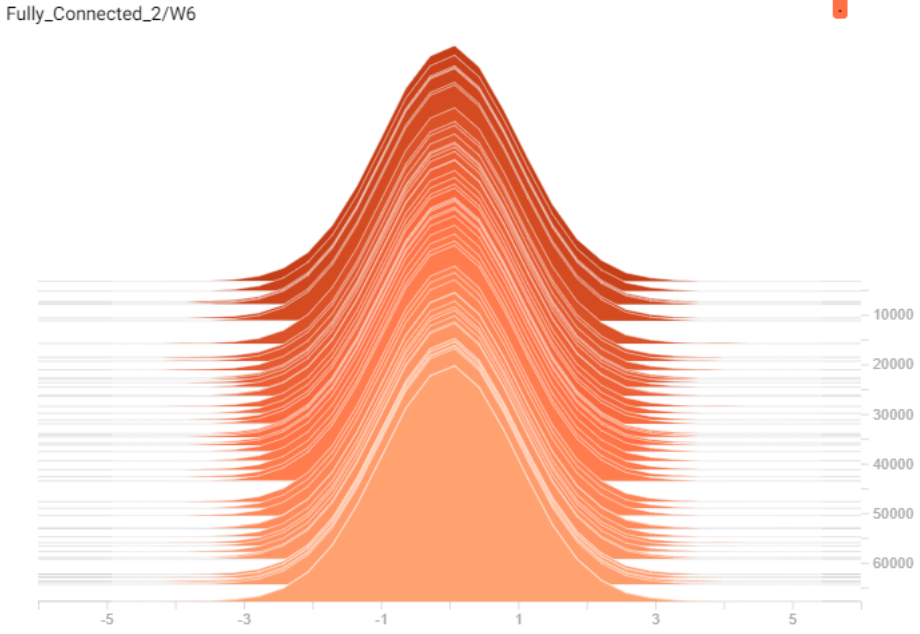


Figure 4.33: Audio CNN Weights - Layer 6

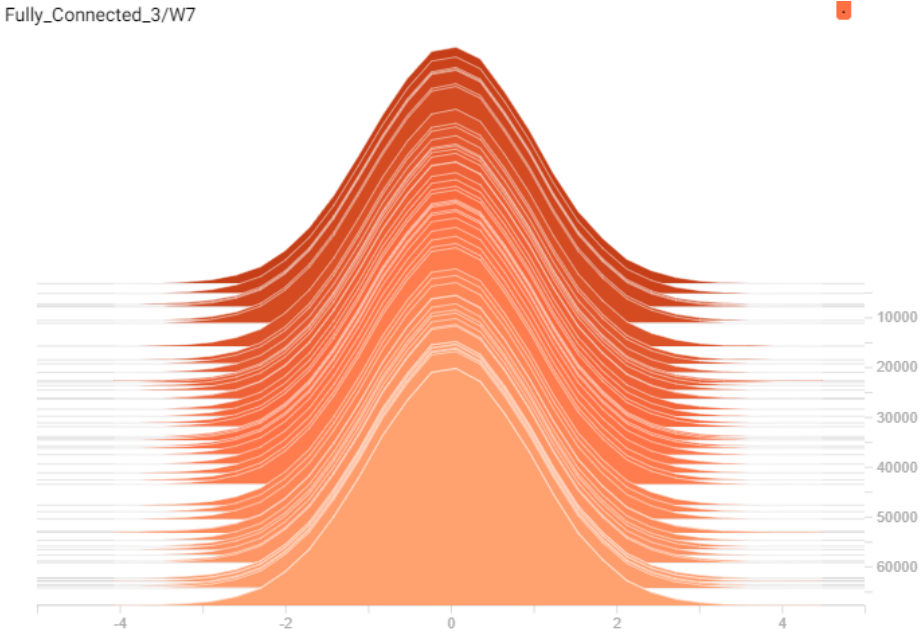


Figure 4.34: Audio CNN Weights - Layer 7

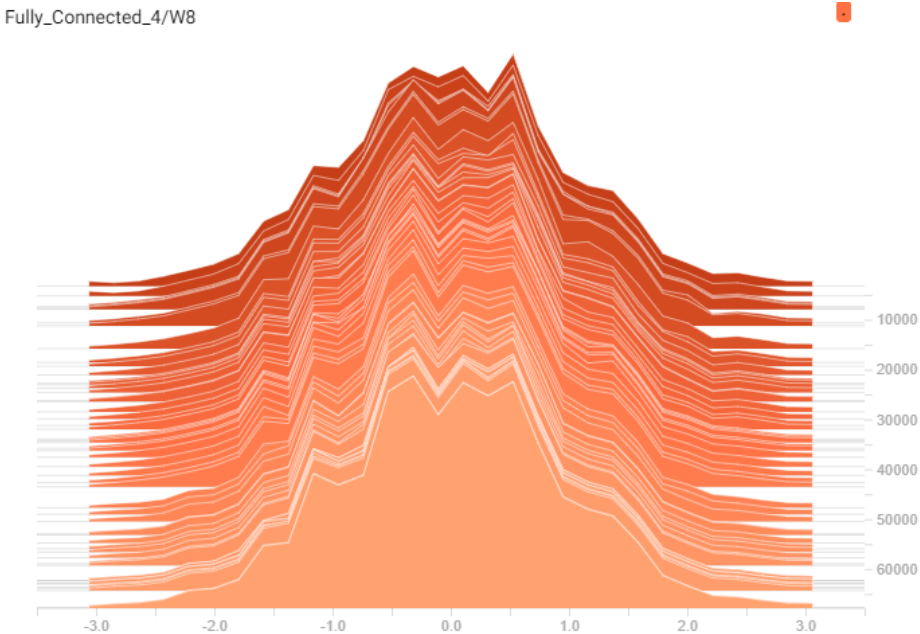


Figure 4.35: Audio CNN Weights - Layer 8

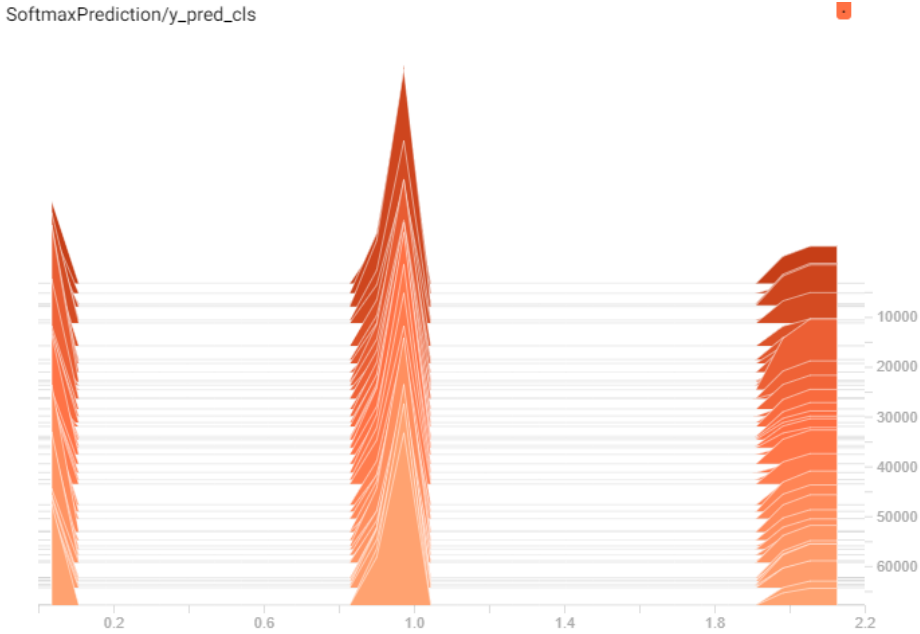


Figure 4.36: Audio CNN Model Softmax Prediction Histogram

Chapter 5

Network Architecture

In the proposed network architectures, each network-layer consists of a convolutional layer, a rectified linear unit (ReLU) activation layer, clipping, and an optional maximum pooling (Max-Pool) layer. The input neurons in the convolution layer convolve with the trained weights or kernels of matching dimensions and produce feature maps of size $(IPSIZE - 2) * (IPSIZE - 2)$, where $IPSIZE$ is input size of the input FM. The ReLU non-linear activation function, in combination with clipping, is applied to resulting FMs to avoid regularization and clip the values outside of the threshold region respectively. A max-pool function is applied to the FM to avoid any overfitting in the network. In the Max-Pool layer, a uniform window size of 2x2 with a stride length of 2, is used throughout the network. The output FM size (FM_{mp}) of the Max-Pool layer is given by equation 5.1.

$$FM_{mp} = ((IPSIZE - 2)/2) * ((IPSIZE - 2)/2) \quad (5.1)$$

The architectures in this project take inspiration from AlexNet [?] and VGG net [?]. The proposed network architectures have a uniform kernel size similar to the VGG16 net. The ICNN also has Max-Pool layers following the convolution and activation layers similar to AlexNet. A

series of fully connected layers following the convolutional layers is also a characteristic inspired by AlexNet. The AlexNet and VGG net are designed for 224x224x3 input images while the ICNN and ACNN are designed for 64x64x3 and 13x99x1 input sizes respectively.

The required depth of a CNN depends on factors such as the number of input neurons, output neurons or number of classes, the dataset, and how distinct the classes are from one another. The greater the number of classes, the more parameters are required to learn the features resulting in a more extensive network.

The network architectures differ for both image and audio accelerators, although both the ICNN and ACNN have eight network layers. As needed, network layers are reused in hardware, inputs are read from memory, and outputs are written back to memory, drastically reducing the area required to implement the accelerator. Sections 5.0.1 and 5.0.2 elaborate on the differences in the architectures of ICNN and ACNN, respectively, and draw comparisons with AlexNet and VGG net in terms of network complexity and architecture.

5.0.1 Network Architecture for Image Classification

The image classification network has four convolutional layers and four fully connected layers. Since the ICNN has only four output nodes, the number of parameters required to learn the features are fewer compared to AlexNet or VGG16/19 net. Therefore, to reduce the size of the network while keeping the accuracy at its best, a Max-Pool layer is added after every convolutional layer. To establish efficient utilization of hardware, a constant kernel size is maintained in all the convolutional layers. Four convolutional layers reduced the neurons to 1024, thus keeping the computations to minimum and thereby enhancing hardware performance.

A constant filter-size of 3x3, with a filter depth matching that of the input FMs are used in the network. The window size in Max-Pool layers is uniform 2x2 with a stride of 2. The architecture is depicted in Fig. 5.1. Convolutional Layers 1 through 3 are inclusive of ReLu, Clipping and

Max-Pool, layer 4 includes only ReLu.

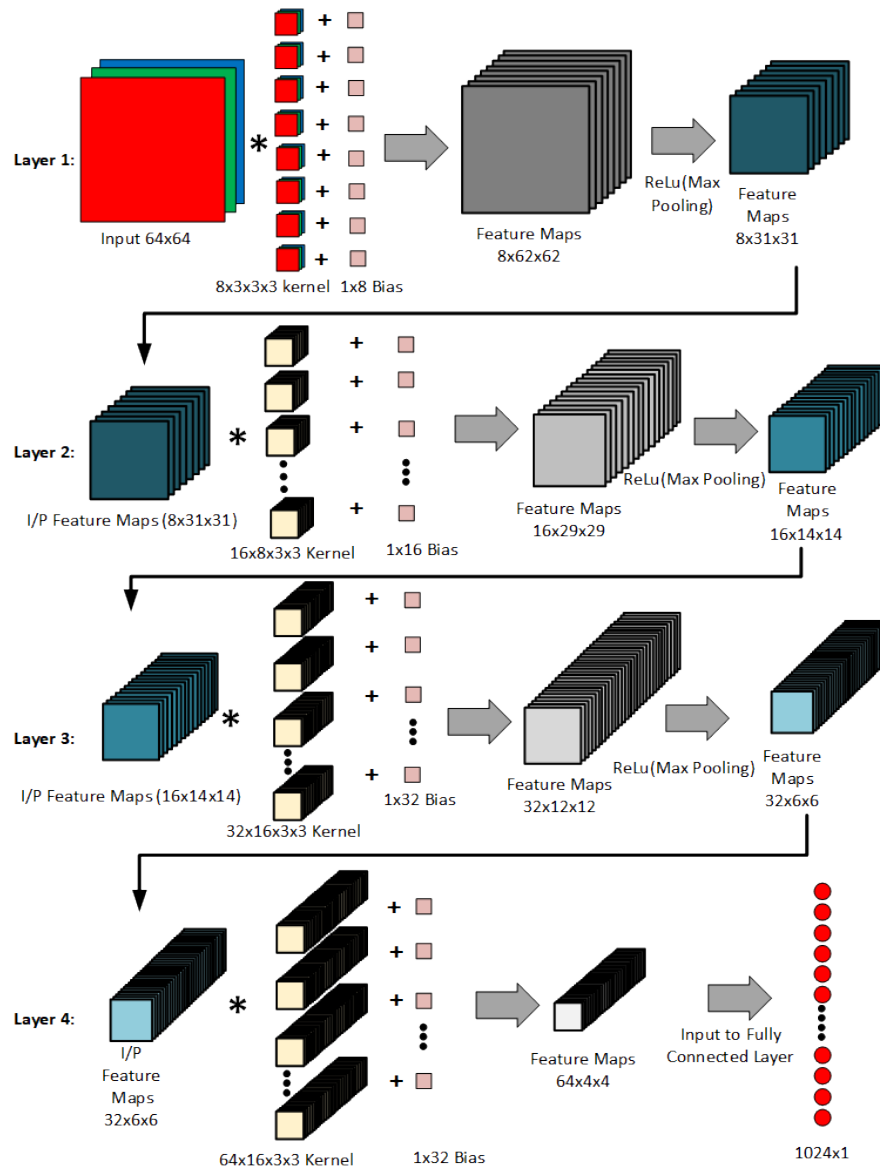


Figure 5.1: CNN Architecture for Image Classification - Convolutional Layers

The input image, with size $64 \times 64 \times 3$, is convolved in layer 1 with 8 filters, with a depth of 3 to produce an output FM of size $62 \times 62 \times 8$. The 8 output feature maps from layer 1 are read into layer 2, and convolved with 16 filters with a depth of 8. The resulting 16 feature-maps are

subjected to ReLu, Clipping and Max-Pool. This reduces the size of output FMs to $14 \times 14 \times 16$. In layer 3, the 16 FMs are convolved with 32 filters that are 16 channels deep. The resulting FMs of size $12 \times 12 \times 32$ are put through ReLu, Clipping, and Max-Pool layers. The feature maps in layer 4 of size $6 \times 6 \times 32$ are convolved with 64 filters, this results in output FMs of size $4 \times 4 \times 64$. The neurons are noticeably reduced, shrinking the size of the network. Hence, in layer 4 the Max-Pool layer is omitted. The output neurons of size 1024 are input to layer 5, to perform a full connection operation with a kernel size of 1024×512 , further reducing the neurons from 1024 to 512 output neurons. The layers 5, 6, 7, and 8 are fully-connected layers as shown in Fig. 5.2, that include ReLu activation layer. The output neuron sizes are reduced gradually from 1024 to 512, 512 to 256, 256 to 128, and 128 to 4 respectively. The sizes are reduced gradually with a minimum loss of extracted features from layer to layer. A Softmax function is applied to the output layer to normalize and back-propagate the loss during training.

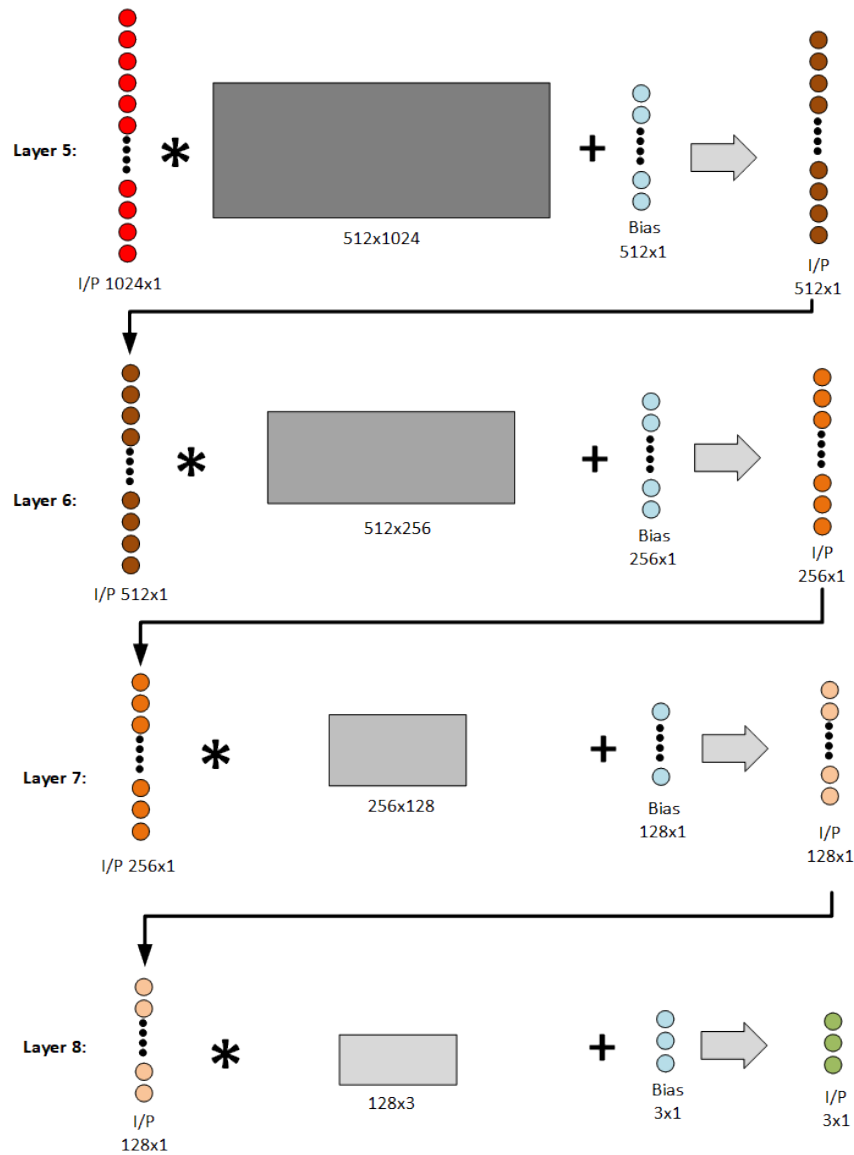


Figure 5.2: CNN Architecture for Image Classification - Fully Connected Layers

5.0.2 Network Architecture for Audio Classification

The audio classification network consists of a total of 8 layers. The first 5 layers are convolutional layers, and last 3 are fully connected layers. During testing different configurations of network

with different combinations of layers were experimented and it was determined that the CNN for Spectrograms required a different architecture from that of ICNN. When the Spectrograms are observed closely, they are not as crisp as input images of ICNN. This is the reason the audio network requires a greater number of parameters to learn the features. Hence the number of Max-Pool layers are reduced to one to reduce any loss of learning parameters.

In the ACNN, layers 1 through 4 are convolutional and are comprised of ReLu and Clipping layers. The Layer 5 is convolutional and comprised of ReLu, Clipping, and Max-Pool layers. Layer 1 takes an input of $13 \times 99 \times 1$ matrix that represents audio data and convolves with 4 single-channel kernels of size 3×3 i.e., $3 \times 3 \times 1 \times 4$. This convolution results in 4 feature maps of size 11×97 . The four output feature maps, $11 \times 97 \times 4$, of layer 1 convolve with 8, 4-channel filters in layer 2. In layer 3, the 8 feature maps of size 9×95 convolve with 16 filters each of depth 8. This results in an output feature map of size $7 \times 93 \times 16$. In layer 4, a filter with dimensions $32 \times 3 \times 3 \times 16$ is used to convolve with its input feature maps, that results in output feature maps of size $5 \times 91 \times 32$. The convolution in layer 5, with 32 filters of depth 32 results in an output feature map of size $3 \times 89 \times 32$. Layer 5 includes Max-Pooling which results in 32 feature maps of size 1×44 that accounts to a total of 1408 output neurons. Layers 6, 7, and 8 are fully-connected layers, where 1408 neurons are reduced to 512 neurons, 512 to 256, and 256 to 3 respectively. The network architecture is summarized in Table 5.2 of Section 5.0.3.

5.0.3 CNN Architecture Summary

The CNN for image classification is denoted by ICNN, and the one for audio classification is denoted by ACNN. The ICNN is built from a total of 713,048 weights or parameters, while ACNN is made of 868,036 weights or parameters. Tables 5.1 and 5.2 summarize the shapes of kernels deployed in every layer and total number of parameters in image and audio CNN accelerators respectively.

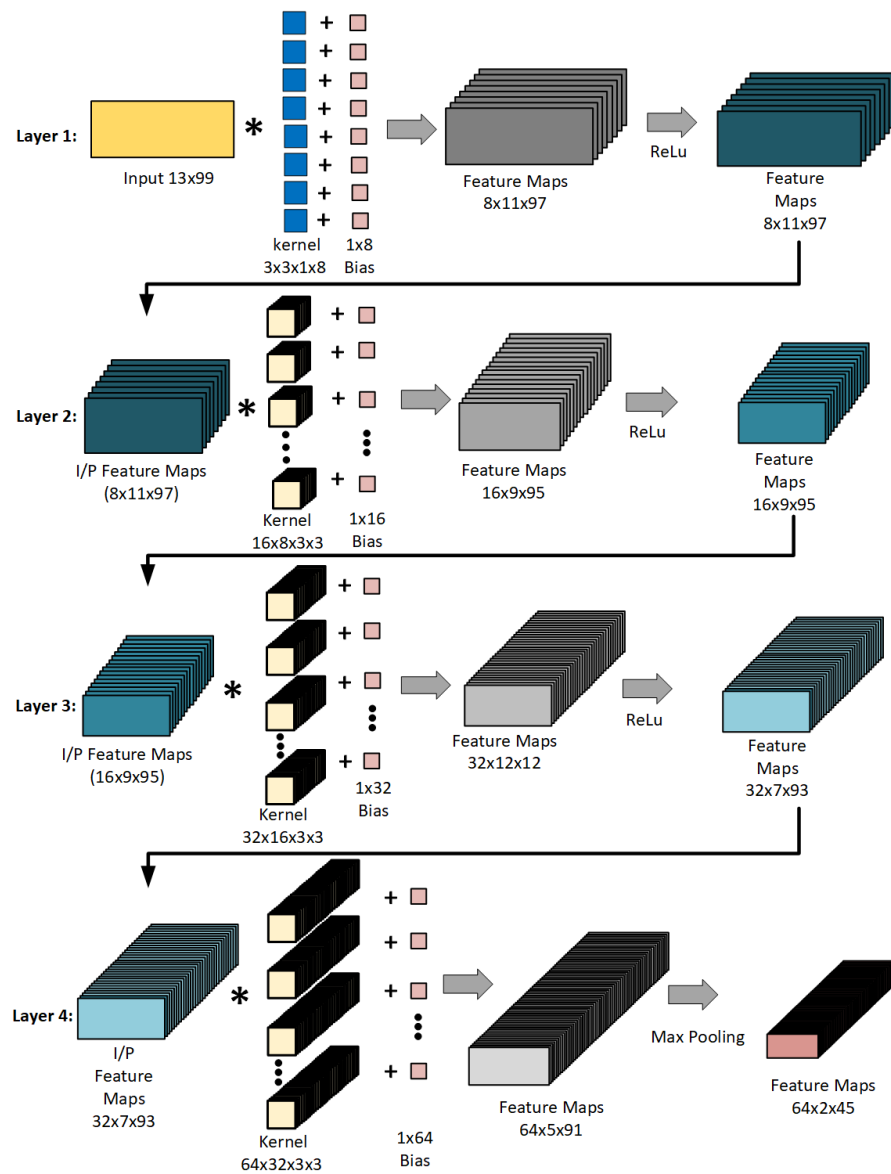


Figure 5.3: Convolutional Layers

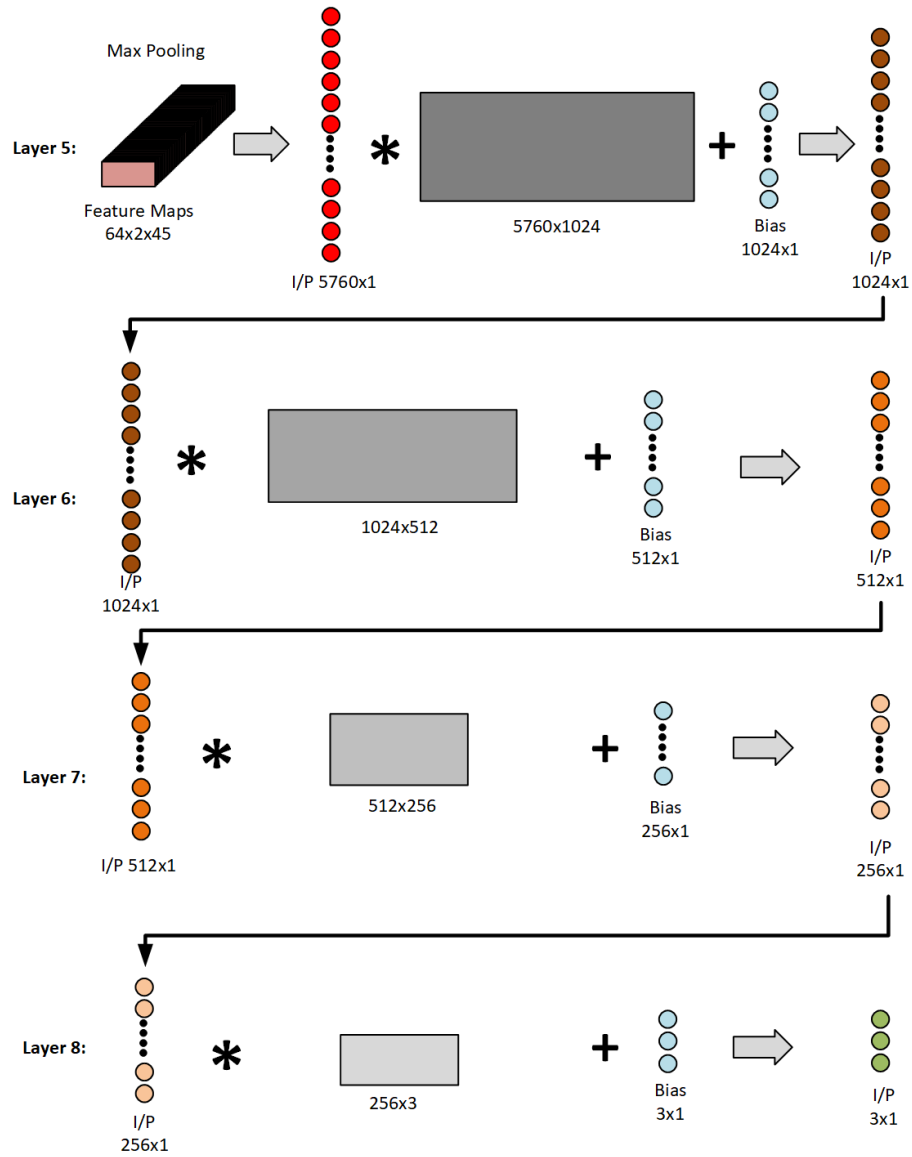


Figure 5.4: Fully connected Layers

Table 5.1: ICNN Architecture Summary

Layer	Type	FM Size	Kernels	Parameters
L1	Conv	$62 \times 62 \times 3$	$3 \times 3 \times 3 \times 8$	216
L1_a	MP	$31 \times 31 \times 3$		
L1_b	ReLu	$31 \times 31 \times 3$		
L2	Conv	$29 \times 29 \times 8$	$3 \times 3 \times 8 \times 16$	1152
L2_a	MP	$14 \times 14 \times 8$		
L2_b	ReLu	$14 \times 14 \times 8$		
L3	Conv	$12 \times 12 \times 16$	$3 \times 3 \times 16 \times 32$	4608
L3_a	MP	$6 \times 6 \times 16$		
L3_b	ReLu	$6 \times 6 \times 16$		
L4	Conv	$4 \times 4 \times 164$	$3 \times 3 \times 32 \times 64$	18432
L4_a	ReLu	$4 \times 4 \times 64$		
L5	FC	512×1	1024×512	524288
L5_a	ReLu	512×1		
L6	FC	256×1	512×256	131072
L6_a	ReLu	256×1		
L7	FC	128×1	256×128	32768
L7_a	ReLu	128×1		
L8	FC	4×1	128×4	512
L8_a	SM	4×1		
Total				713,048

5.1 Fixed Point Representation and Computation for Hardware Efficiency

The two networks ICNN and ACNN are trained in Tensorflow using 32-bit floating-point values. Since implementing hardware for floating-point computations can be exorbitantly expensive, fixed-point implementation for hardware efficiency was chosen. This section discusses the techniques adopted in the process of transitioning from a 32-bit floating-point to a fixed-point implementation of the networks.

The network parameters are retrieved from Tensorflow's trained model, using a reference

Table 5.2: ACNN Architecture Summary

Layer	Type	FM Size	Kernels	Parameters
L1	Conv	$11 \times 97 \times 4$	$3 \times 3 \times 14$	36
L1_a	ReLu	$11 \times 97 \times 4$		
L2	Conv	$9 \times 95 \times 8$	$3 \times 3 \times 48$	288
L2_a	ReLu	$9 \times 95 \times 8$		
L3	Conv	$7 \times 93 \times 16$	$3 \times 3 \times 8 \times 16$	1152
L3_a	ReLu	$7 \times 93 \times 16$		
L4	Conv	$5 \times 91 \times 32$	$3 \times 3 \times 16 \times 32$	4608
L4_a	ReLu	$5 \times 91 \times 32$		
L5	Conv	$3 \times 89 \times 32$	$3 \times 3 \times 32 \times 32$	9216
L5_a	ReLu	$3 \times 89 \times 32$		
L5_b	MP	$1 \times 44 \times 32$		
L6	FC	512×1	1408×512	720896
L6_a	ReLu	512×1		
L7	FC	256×1	512×256	131072
L7_a	ReLu	256×1		
L8	FC	3×1	256×3	768
L8_a	SM	3×1		
Total				868,036

model to perform inference. The reference model is verified meticulously against the Tensorflow model, in both floating-point and fixed-point representations. On comparison of floating-point network computations with fixed-point computations, a small delta was observed, implying the loss of precision from inference in 32-bit floating-point to 16-bit fixed-point values.

During fixed-point inference, the minute precision loss in the initial layers of CNN accumulates over the layers down the network and alters the accuracy of the network. To minimize the loss, a clipping function is introduced in combination with the ReLu activation function. This curtails the resulting values to lie within a range defined by the threshold value.

A threshold value is chosen based on factors like the maximum number of additions or accumulations that result in a single value, the distance of the current layer from the output layer, etc. The number of additions is taken into account specifically because the result of the multiplication of two fractional (fixed-point) numbers results in a smaller value as opposed to addition. If the feature map resulting from the first layer of the network has large values and is input to the next layer for further processing that involves additions, the resulting values will overflow in a 16-bit fixed-point implementation and cause erroneous computations down the network.

For example, in a fully connected layer with 1024 input neurons, and 512 output neurons, the input neurons are multiplied with a weight matrix or a kernel of size 512x1024. This results in a maximum of 1023 addition and 1024 multiplication operations per output neuron. Suppose the numbers are clipped to a maximum of 0.3 (which is 9630 in 16-bit fixed-point), and all 1024 elements to be accumulated equal to 0.3 each. This accumulation results in a value as large as 306.9 which is equivalent to 20112998 in 16-bit fixed-point representation, all with a delta as minute as 0.0000061035156.

However, a clipping function cannot be applied to lower level computations involved in convolution over volumes or the calculations in a fully connected layer as it might directly affect the mathematical characteristics of the function as it involves signed integers. For this reason

the adders and multipliers deployed in convolution are 23 bits wide for 16-bit implementation to prevent any loss of precision and incorporate a maximum value that is encountered during computation.

The floating point numbers are converted to fixed point representation of different bit widths using equation 5.2, where $fxpt$ represents fixed point value, flt represents float value, and Q represents the bit width for the two's complement fixed-point representation, also known as the Q format.

$$fxpt = flt * 2^Q \quad (5.2)$$

Chapter 6

Hardware Design & Implementation

The hardware is designed to incorporate all bit widths for different Q formats which represent the two's complement fixed-point values. In this case, the Q format represents fractional numbers only, there is no integer part. The following sections elaborate on application-specific system requirements that this design is based off, system design and architecture, and hardware implementation of basic blocks.

6.1 System Requirements

The present industry standard for a camera to capture a clear video is 30 frames per second (fps). If a video is captured at a frame rate of 30 fps, it leaves the hardware accelerator with a processing time of 0.033s or 33ms. The audio data is captured at different sample rates, like 44.1 kHz, 48 kHz, 32 kHz, etc. The ACNN processes a sample of length one second long with a default sampling rate of 44.1 kHz. Since the ACNN is processing one-second audio files, it leaves the hardware accelerator with a processing time of one second for each input audio sample.

In order to reduce the computational complexity, the numbers were clipped every time they

crossed a threshold value after every layer of computation. This helps maintain a range of values and keeps the bit-width of basic hardware blocks to a minimum.

There are a number of trained parameters that the network should be initialized with to start processing the data. This implies that the system should be able to store the trained parameters and intermediate data of feature-maps at every layer. The memory requirement is subject to change based on the bit-width.

6.2 System Design and Architecture

A top-down design methodology is followed in this design process. The system-level architecture is shown in Fig. 6.1, which consists of a host-processor, ICNN, ACNN, and an off-chip main-memory (MM). The accelerator design uses a 32-bit wishbone bus to establish communication between several cores, as shown in Fig. 6.1. The host-processor writes the network parameters and the real-time data received from a camera and a microphone to the main-memory. It also writes the network configurations of each accelerator to allocated memory space. The network configurations include parameters that select the mode of the layer (convolution or fully-connected), include or exclude the Max-Pool layer, select a clipping threshold, input, and output feature map and Kernel dimensions, the base address of corresponding layer parameters, input FMs, and output FMs.

After setting up the configuration space for all the layers of both the accelerators, the host-processor writes to the “Configuration Base Address” and then the “Go bit” registers of the accelerators. This sets off the accelerators with all the required parameters to run independently and write the final output results of the inference to the main-memory to be accessed by the host processor for further system processing.

The system bus arbiter is set up to have the highest priority for the host-processor, followed

by ICNN and then ACNN. For overall performance, the system bus runs at a clock frequency of 350 MHz, which is higher than the frequencies of either accelerator.

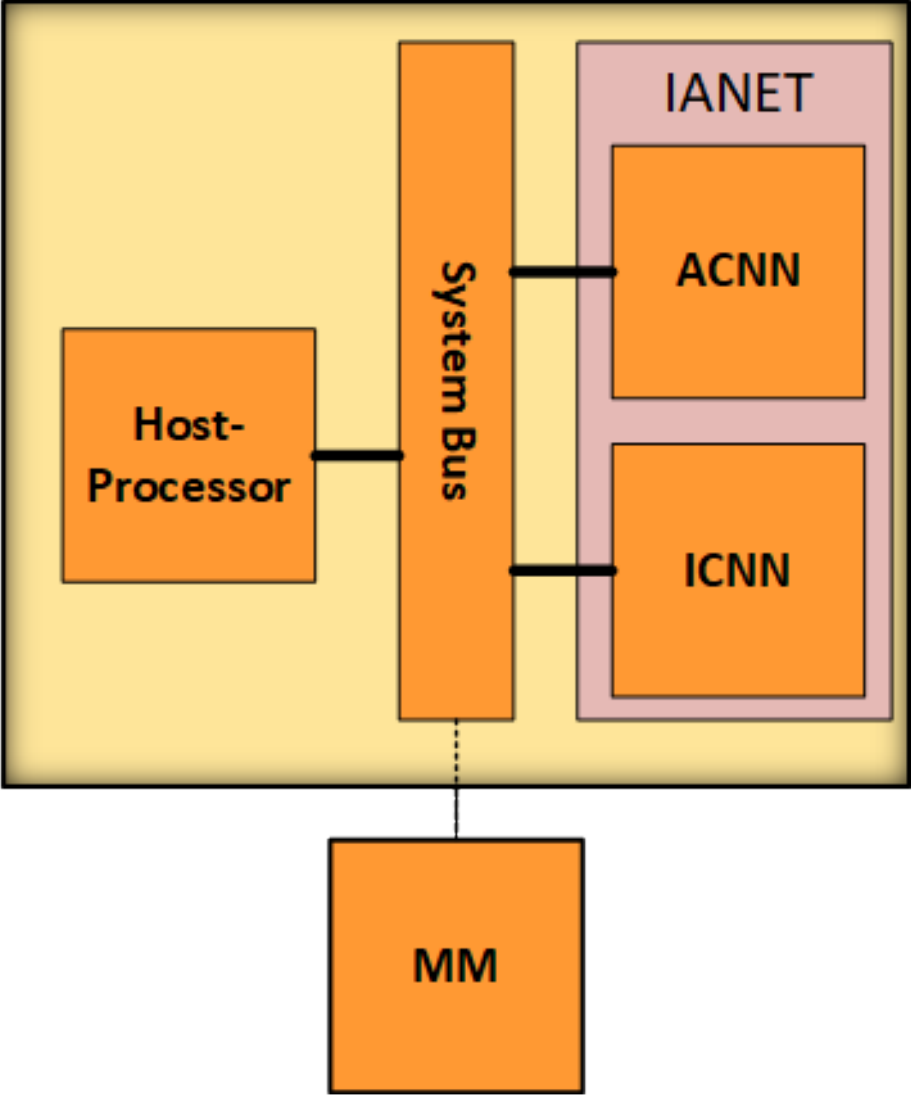


Figure 6.1: System-Level Architecture

Fig. 6.2, shows the high architecture of the accelerators. The accelerator consists of the Convolution unit, Fully-Connected unit, Arithmetic Logic Unit, the master and the slave cores, which communicate with the rest of the system via system bus interface that directly connects to the master and slave cores as depicted in Fig. 6.2. The ALU consists of an adder, multiplier,

accumulator, Clip & ReLu, and Max-Pool blocks that are reused between the two layers based on the select mode, as shown by the multiplexer in Fig. 6.2. The outputs of the blocks are decoded within the respective layers. The status-registers of the slave port are written to update the progress of the network. The host-processor can read the slave ports to monitor/poll the output.

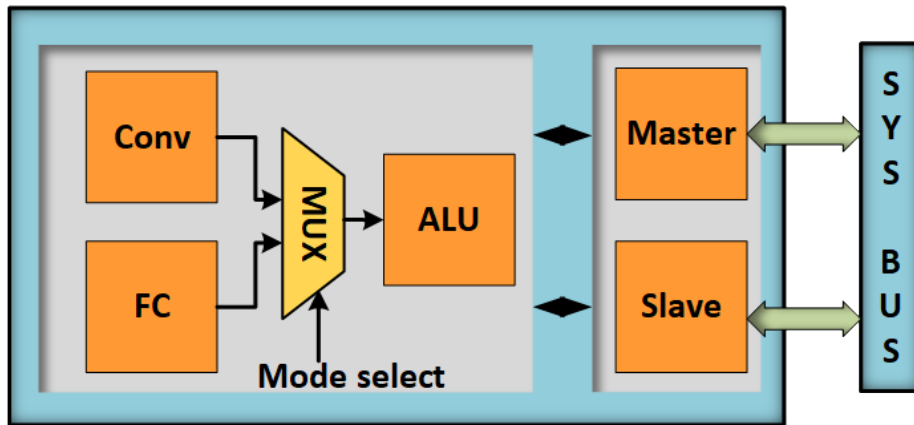


Figure 6.2: High Architecture of the Accelerators

Fig. 6.3 shows the top-level functioning of the system that combines all the state machines. The system is initially in the IDLE state until it receives a start bit. When the start bit is written to the accelerator, the configuration of first layer is read. When the accelerator reads the configuration of the first layer, it reads the input and output sizes, the base address of weights, feature maps, and output feature map. After acquiring all the required information, the valid bits for reading weights, feature maps, and writing outputs are written.

However, the “read weights” state machine starts to run first, and the other state machines that read feature maps or the computation state machine will remain in IDLE states. As shown in Fig. 6.3, RD_WT state machine starts right after the config state machine, although the valid bits for other state machines are written. The RD_FM state machine starts if and only if the val_base_fm_adr (valid bit) and has_wt signals are written. The has_wt signal indicates that the

weights required for the current iteration are read from memory and are ready for computation. On having the two signals set, the RD_FM state machine starts reading the current window of the feature map. On completion, the signal internal to RD_FM state machine that indicates the system has a current window of feature maps is set, which is has_fm signal.

Once the has_wt, has_fm, and val_base_op_adr signals are set, the ACCMUL state machine starts to run. This state machine computes the dot product of the two matrices that were read previously. After performing the dot product, it saves the result in the temporary storage array of internal registers. For the next output value, the next window of the feature map should be read. For this to happen, the ACCMUL state machine indicates that it is done with the dot product for the current window. The acc_mul_cmplt signal indicates this. When this signal is set, the has_fm signal turns low, and the RD_FM state machine starts to read the next set of values. After reading the current window of the feature map, the ACCMUL state machine computes the next output value and stores it in the array. This process continues until the end of the feature map is reached.

After reading and convolving the current feature map with the current weight matrix, the channel counter is incremented, fm_cmplt a signal that indicates the completion of convolution over one channel is turned high and has_fm is turned low. The channel count is incremented to keep track of the number of input channels being convolved. When the RD_FM state machine sets the fm_cmplt signal, the RD_WT state machine sets the has_wt signal low to stop other state machines from running and reads a new set of weights. This state machine then turns the has_wt signal on. This then sets off the RD_FM state machine to coordinate with the ACCMUL state machine to run and compute the convolution. The result of the convolution of previous channel FM and WT are stored in the array that is accumulated and written to main memory after performing ReLU and Max-pool operations. This process is explained as mentioned below.

After performing Max-pool operation, the system sets back to the “config” state machine,

where the number of layers processed is kept track of. The entire process repeats for every layer until the counter that keeps track of the number of layers is exhausted.

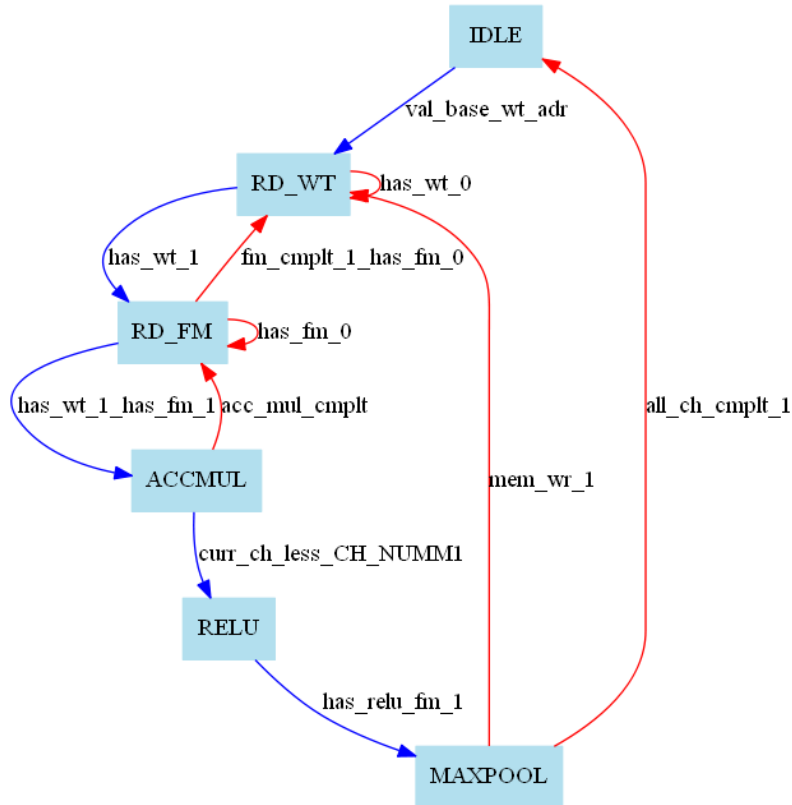


Figure 6.3: State Machine for Top Level Implementation

6.3 Network Configurations

The accelerator core reads layer-specific configurations from the main memory on receiving a start bit. On reading all the specifications like input size (row, col), output size (row, col), select bit for the type of network (Convolutional or Fully connected), select bit for the Max-Pool operation, select bit for the compare threshold value, the hardware is configured for further computation. The layer-specific FM base address and weights base address are read along with the `val_base_fm_adr` and `val_base_wt_adr` bits, which essentially act as active low resets to the

“read feature maps” and “read weights” state machines. The Fig. 6.4 shows the state machine for configuration of hardware before every layer.

The start bit of the Config state machine and the base address for layer configuration data is written through the slave port of the accelerator. When the start bit is written, the config state machine goes into CNFG_START_BIT state from CNFG_IDLE state, as shown in Fig. 6.4. In the “CNFG_START_BIT” state the wishbone address bus is set to the base address, the read request signal is set, so that the master sets the wishbone cycle and strobe signals to request access to the bus. On obtaining the control over the bus, the data at the required address pointer is read, and its validity is indicated by the acknowledgment signal. The 32-bit data that is read in this state is split into multiple signals like mp_flag, conv_fc_flag, cmp_flag, number of filters, and the number of channels. The total maximum number of filters in ICNN is 64, and in ACNN are 32, which take up a maximum of 6 bits at any time, but 8 bits are allocated to this field for uniformity. The total maximum number of channels in both the networks is 32, which requires a maximum of 5 bits, but 8 bits are allocated for uniformity. The other three signals are given 8 bits together, but the least significant 3 bits hold the data for three signals. The state machine reads 16-bit words. Prior to reading the data into their respective registers, the read request is turned low so that the bus is free and for the stability of the system. This process ensures every read and write transaction on the bus is distinguishable. The address pointer is incremented for next read, and the next state is assigned.

The state machine then goes into CNFG_IP_SIZE state from CNFG_START_BIT state, as shown in the Fig. 6.4. In this state, the dimensions of the input feature map, like the row size and column size, are read. The address pointer is incremented, and the read request is turned low. The next state is assigned based on the conv_fc_flag signal that was read in the previous state. This signal indicates whether the layer is convolutional or fully connected. If the layer is convolutional, CNFG_MP_SIZE is assigned else CNFG_WT_BASE_ADR is assigned, as can

be inferred from the Fig. 6.4. In CNFG_MP_SIZE, the output sizes of convolution and max-pool sizes are read so that the down counters for rows and columns can have a preset value of the sizes read. The CNFG_RC_SIZE is the next state-assigned in CNFG_MP_SIZE. In this state, the product of the row and column is read. In the next state CNFG_WT_BASE_ADR, as the name of the state indicates the base address of where the layer-specific weights are stored is read along with a valid bit. The valid bit is the most significant bit of the 32-bit word.

Similarly, in the next two states, the base address of input and output feature maps are read. The IDLE state is assigned after reading all the necessary configuration data. The state machine starts again when all the valid bits turn low; all the valid bits turn low when one single layer is complete.

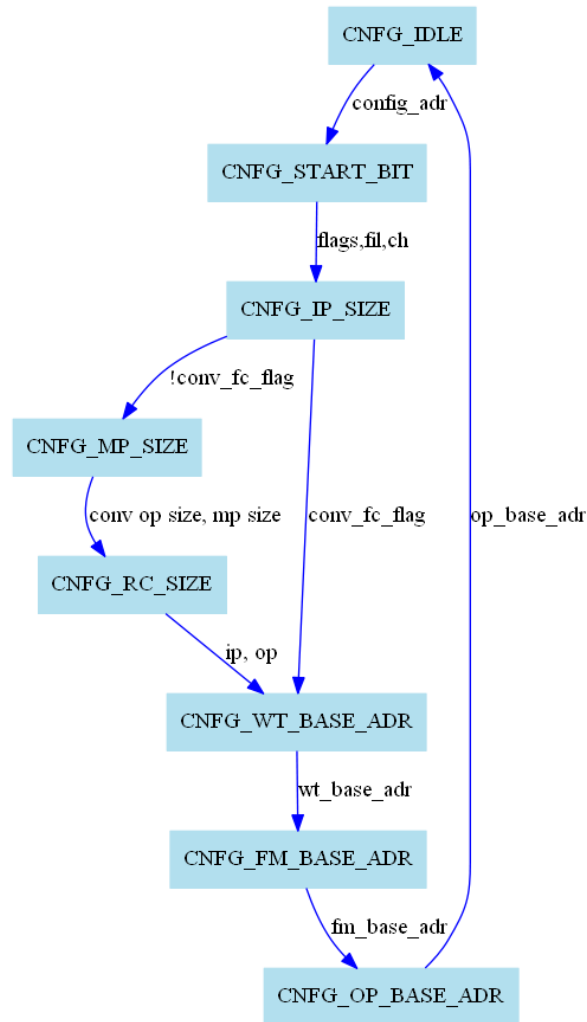


Figure 6.4: State Machine for Configuration of Layer Parameters

6.4 Memory Configuration and Transactions

The main-memory is organized as depicted in Fig. 6.5 to store weights and feature maps of both ACNN and ICNN networks. The ICNN weights are stored starting at 0x1000000 to 0x1900000, ACNN weights start at 0x4000000 and continue to 0x4A00000. The Tables 6.1 and 6.2 show the layer-wise base address for weights and feature maps of for both the networks respectively.

Through a close inspection of the base addresses listed in the table for weights, it is observed that the base address constantly varies with a difference of 0x100000 between the successive layers. However, there is a difference of 0x200000 between the base address of layer 5 and layer 6 of ICNN, a difference of 0x300000 between the base address of layer 5 and layer 6, and a difference of 0x200000 between the layers 6 and 7 of the ACNN. This difference is because of the varying sizes of kernels with respective layers. The kernel sizes of layer 5 in ICNN, and layer 5 and 6 of ACNN are bigger than the kernels of other layers.

From the Table 6.2, the base address of feature maps of all the layers of ICNN and ACNN are seen. The table lists number of the layer, type of the layer, ICNN feature map base address, and ACNN feature map base address. In the type of layer column, IP stands for input layer, Conv stands for convolutional layer, FC stands for Fully connected layer. As can be inferred from the table, the input feature maps for ICNN and ACNN are stored at 0x1A00000 and 0x3000000 respectively.

The configuration state machine reads the base address for weights and feature-maps that is supplied to address pointers in RD_WT and RD_FM (from Fig. 6.3) state machines. The input dimensions of weights and feature maps are used to determine the end of reading weights or feature-maps.

Table 6.1: Base Address of Weights

Layer No.	ICNN WT Base Address	ACNN WT Base Address
1	0x1000000	0x4000000
2	0x1100000	0x4100000
3	0x1200000	0x4200000
4	0x1300000	0x4300000
5	0x1400000	0x4400000
6	0x1700000	0x4700000
7	0x1800000	0x4900000
8	0x1900000	0x4A00000

Table 6.2: Base Address of Feature Maps

Layer No.	Type of Layer	ICNN FM Base Address	ACNN FM Base Address
0	IP	0x1A00000	0x3000000
1	Conv	0x2100000	0x3100000
2	Conv	0x2200000	0x3200000
3	Conv	0x2300000	0x3300000
4	Conv	0x2400000	0x3400000
5	FC	0x2500000	0x3500000
6	FC	0x2600000	0x3600000
7	FC	0x2700000	0x3800000
8	FC	0x2800000	0x3900000

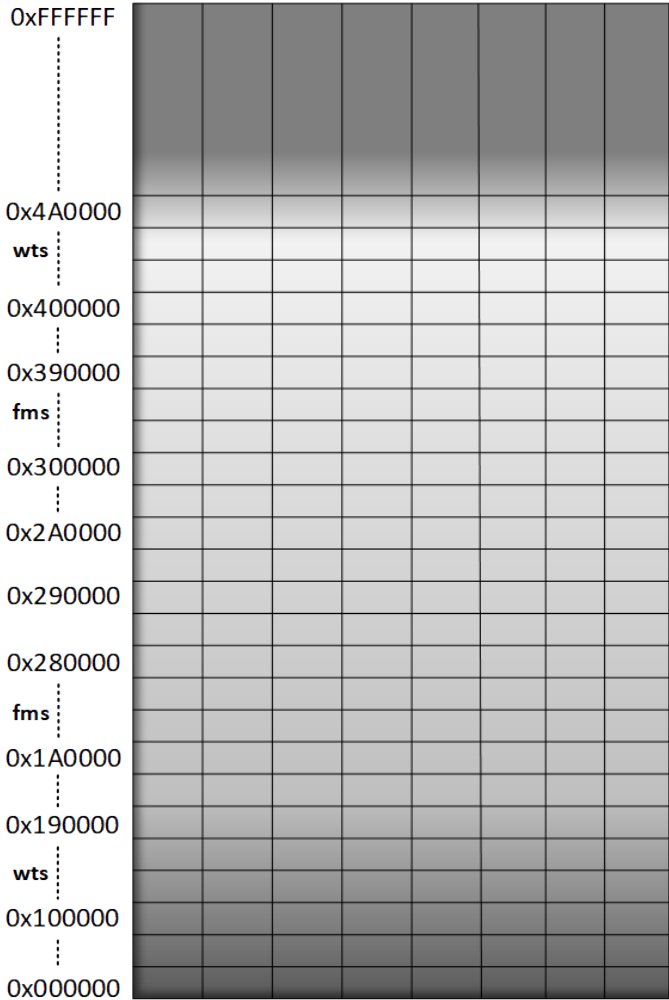


Figure 6.5: Main Memory Organization

The configuration data for ICNN and ACNN is stored starting at addresses 0x2900000 and 0x2A00000 respectively. It has been taken care that the memory space for both the accelerators doesn't overlap.

As the Fig. 6.5 depicts the main memory has 128 bit wide blocks with 16 eight bit words. The main memory has 4 select bits that select between the sixteen words. Since in this project, 16-bit words are used the select lines are used accordingly while reading or writing the data. Twenty-eight bits of 32-bit address bus are used to address a 128Mb memory. The address is

written to the base address slave ports of the accelerator, these are set to the address pointers of the accelerators. The least significant bits of the base address pointer are incremented to read the required values.

The Table 6.3 lists the slave ports and their respective addresses. The two accelerator slaves have unique addresses. The ICNN slave address is 0x3300 and ACNN slave address is 0x3400. To access the ports from the host processor, the host processor uses the slave address combined with the port address. The port names listed in table are SRESET (software reset), FM_BASE_ADR (feature map base address), WT_BASE_ADR (weights base address), OP_BASE_ADR (output base address), FM_CUR_PTR (feature map current pointer), WT_CUR_PTR (weights current pointer), OP_CUR_PTR (output current pointer), CUR_PTR (current pointer), CUR_CH (current channel), FM_STATUS (feature map status), DONE (CNN done), and LAYER_CONFIG (layer configuration base address). The most frequently used slave ports are LAYER_CONFIG and DONE. The LAYER_CONFIG port is used to write the valid base address for configuration data and the most significant bit being the start bit to the accelerator. The DONE port is set by the accelerator when one complete sample is processed and the end result is written back to memory. This port is a read only port, and is monitored to identify when the network has reached the end point.

Table 6.3: Accelerator Slave Ports and Addresses

Port Name	Address
SRESET	0x0000
FM_BASE_ADR	0x0004
WT_BASE_ADR	0x0008
OP_BASE_ADR	0x000C
FM_CUR_PTR	0x0010
WT_CUR_PTR	0x0014
OP_CUR_PTR	0x0018
CUR_FIL	0x001C
CUR_CH	0x0020
FM_STATUS	0x0030
DONE	0x005A
LAYER_CONFIG	0x0056

The state machines in Fig. 6.6 and Fig. 6.7 show the detailed hardware implementation of memory transactions between the accelerators and main-memory to read weights and feature maps.

In IDLE_WT state all weights in nine registers are cleared and the base address is set to the current pointer and the next state is assigned if the has_wts signal is low else the has_wts signal is cleared and the next state is assigned. The next state is RD_W0 as shown in Fig. 6.6. The signal has_wts acts as an active low read request. Every time this signal is low the master of the accelerator gains access over the bus and reads values at the required location in main memory.

The state machine ensures the presence of valid address in the address pointer at any given time. The state remains in RC_W0 until it receives the ack (acknowledgment) from the memory to the master of the accelerator core. On receiving the ack, the 32-bit data received is read into two 16-bit registers. The address pointer is kept 16-bit word addressable and not block addressable. This means that out of the four least significant bits of the address pointer, the 2 least significant bits are kept constant which results in selecting 16-bit words instead of 8-bit words. The address pointer is incremented by two and the next state is assigned. However, it was observed that due to the odd number (odd number - 9) of parameters being read each time, reading exactly two values at any given time results in extra reads and wrong values.

For example if in the first iteration, the eight values of the filter are read by incrementing the pointer by 2 and reading exactly two values at once each time. To read the ninth value of the filter the pointer is set in the previous state and is pointing to the right value, the 32 bit data is returned from main-memory. But only the required 16 bit data is read by the state-machine, and for this reason the pointer is incremented by 1 and not 2, so that it points to the adjacent value for the next read and not skip the value. After being incremented by one, the pointer is pointing at the most significant 32-bits of the 128-bit block of the main memory. In the previous state least significant bits of most significant 32-bit word were read and now when the pointer is odd i.e., say 9, the most significant 16-bits of the most significant 32-bit word are and the pointer is incremented by 1. This makes the pointer even and from next location until another odd pointer occurs, two values are read each time and the pointer is incremented twice. The state-machine in Fig. 6.6 depicts the same logic while it points from state to state based on the least significant bit of the pointer. After reading every nine filter values and before reading another set of nine filter values the state remains in IDLE_WT state and waits for the convolution to complete.

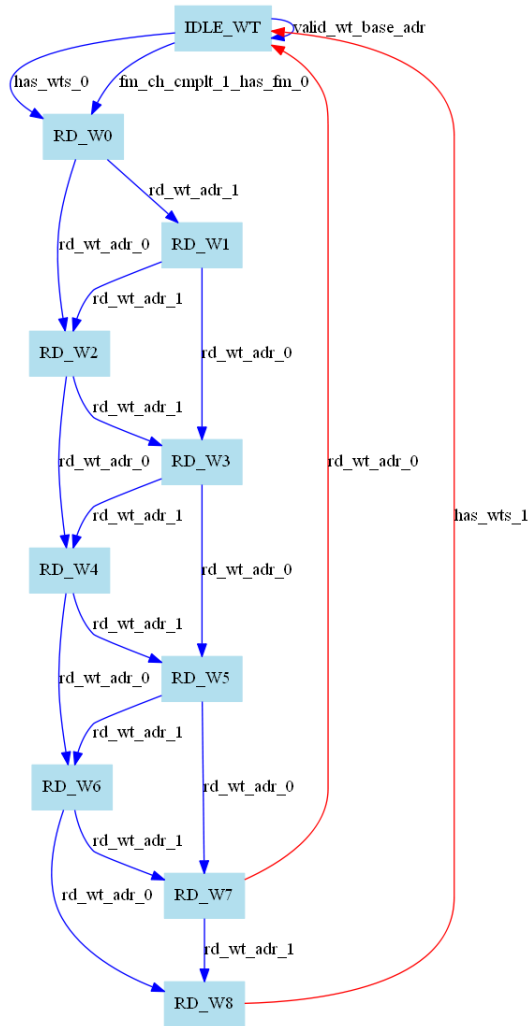


Figure 6.6: State Machine to Read Weights of the Layer

Reading Feature maps follows a similar reading pattern as that of reading weights. However reading weights is much simpler than reading feature maps as it is a simple process of reading odd number of continuous values. The process of reading feature maps is complicated because convolution involves reading three continuous values from each row of the feature map, and reading values in similar pattern from all the channels until the channel count exhausts and continue the process for number of filter count times. The counters are shown by the red arrows in the Fig. 6.7. This state machine has three pointers because it has to read and keep track of the

previous locations in three different rows of the feature map. The window keeps sliding until the row count exhausts as mentioned earlier.

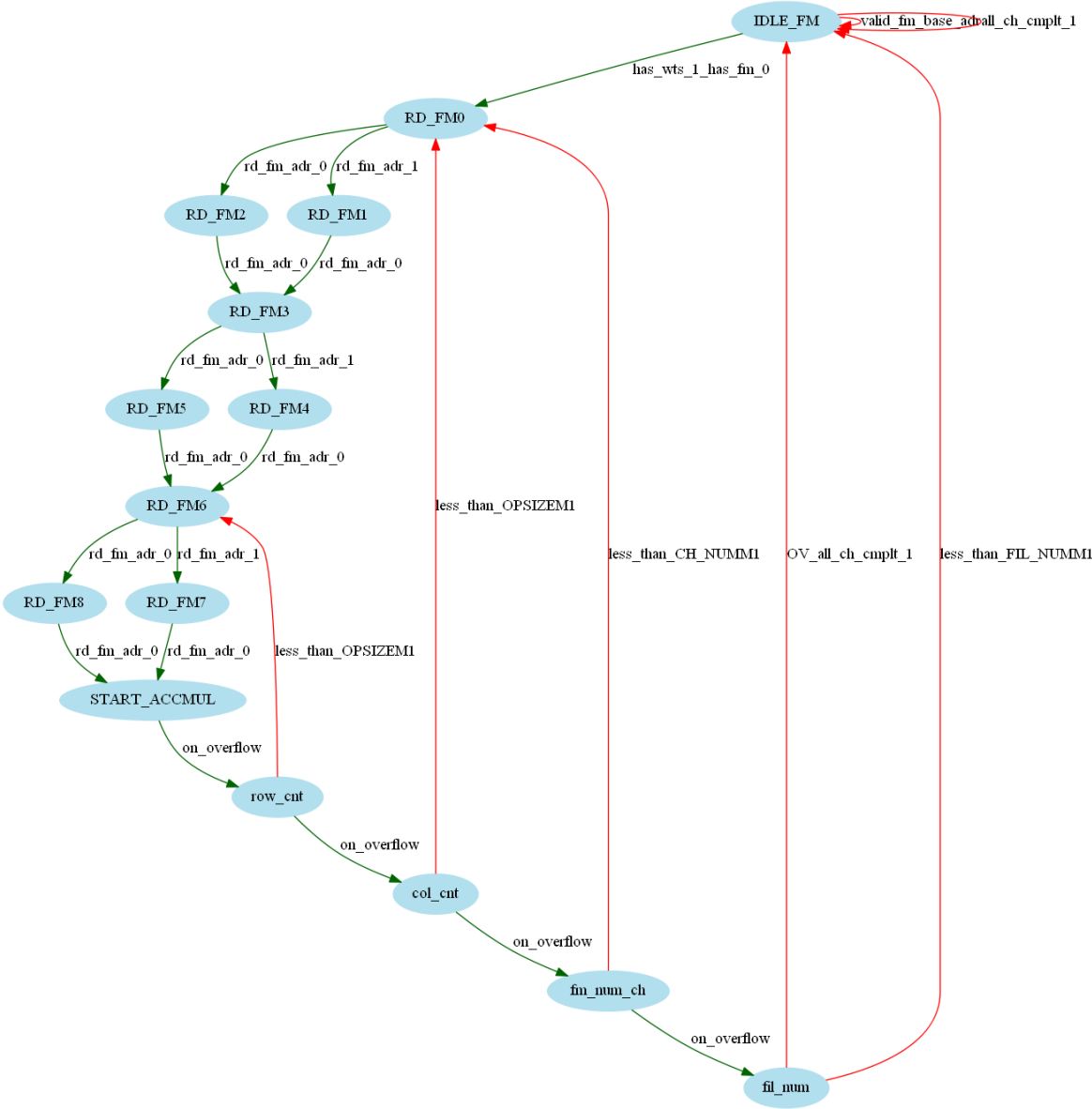


Figure 6.7: State Machine to Read Feature Map of the Layer

6.5 Design and Implementation of Convolutional Layers

The convolutional layer is designed, as shown in Fig. 6.8. Since both CNN accelerators have a uniform 3x3 kernel size, two 16-bit arrays of size nine are deployed, each for FM and weights, and named FMCH1 and WCH1, respectively. The weights are read from memory once every channel (taking advantage of uniform kernel size). In contrast, 9 FM values are read every new row and 3 per new column (making column-wise strides) until the row and column counters exhaust (as shown in Fig. 6.8). This method significantly reduces memory reads for every column stride. Values from each Feature Map Channel 1 (FMCH1) and Weight Channel 1 (WCH1) array are input successively to a multiplier performing the dot product followed by nine continuous accumulations accomplishing convolution operation. The result of convolution is stored in FeatureMap block, as shown in Fig. 6.8. The result of every channel is stored in the FeatureMap block to accumulate forthcoming channel data. A channel counter counts the number of times a row counter exhausts, to track the operations of a single filter. The output values are written to main-memory after performing a convolution of every filter, ReLu, and/or Max-Pool as per the layer configurations. The process continues until the filter count exhausts, indicating the end of a single convolutional layer.

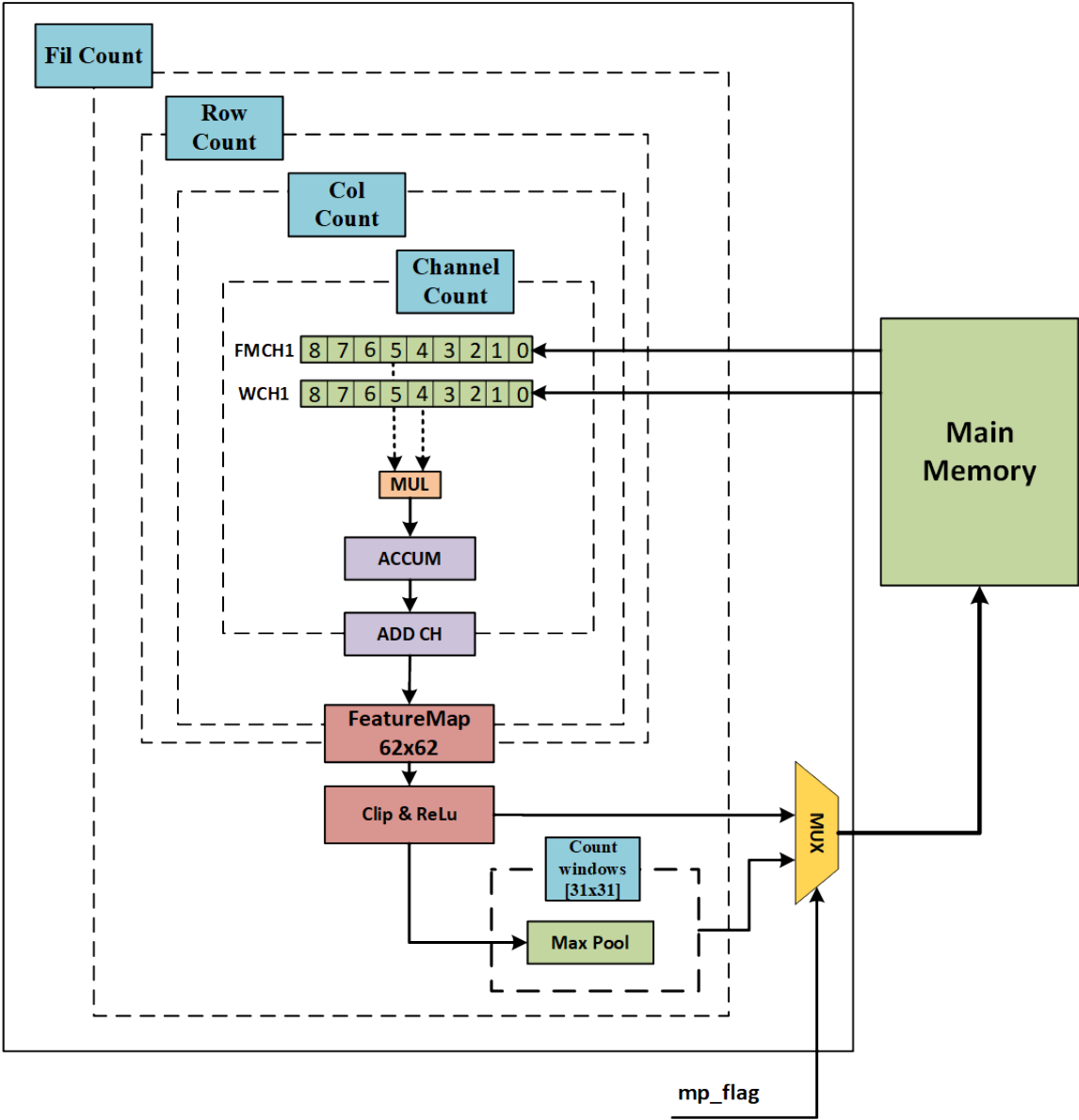


Figure 6.8: Structure of the Convolutional Layer

Fig. 6.9 shows the state machine for the convolution operation of CNN. The state is ACCMUL_IDLE until the input weights and the current feature map window is read. Once this data is valid and available, the ACCMUL state machine assigns the next state where the dot product is initiated. The idle state determines if valid weights and feature map values are available based

on `has_wts` and `has_fm` signals. The dot product of a 3x3 matrix involves nine multiplications and eight accumulations. The first multiplication is performed in an IDLE state on determining the valid numbers in the source matrix. There are eight multiplication states followed by the idle state. In MUL1 state, the second value of weights array and feature map array are assigned to the inputs of the multiplier, and the accumulator is activated by setting the start bit. In MUL2 state, the output of the first multiplication is available and is input to the multiplier. By the end of MUL8 state, the nine values are multiplied, and seven values are accumulated. The next two states are added to accumulate the following two values. The start bit is turned low after storing the output from the accumulator in the CMPLT state. The current row and column of the output feature map that is being processed are stored into temporary register values for future reference. The data is stored using these provisional reference row and column values in the two-dimensional matrix. In the ADD state, if the current channel number is equal to a total number of channels, then the `fm_ch_cmplt` and `fm_cmplt` signals are compared to check if the convolution over volumes for the current feature map is completed. If it is completed, then the state raises a `has_relu_fm` signal indicating the feature map is ready for the ReLU operation. The next state ADD1 is assigned, where the channel accumulated data is stored into its respective location of the two-dimensional array using the temporary reference row and column pointers. On completion of channel accumulation, ReLU is performed on each value of the 2-D matrix, and the output of ReLU is collected at the next clock cycle. The Max-Pool flag is checked, if it is set MAXPOOL state is assigned to next state else, the 2D matrix is written as-is after performing ReLU. In MAXPOOL state, the window is slid row-wise and column-wise similar to convolution, and the two values are written to memory at once, saving two clock cycles for every other value.

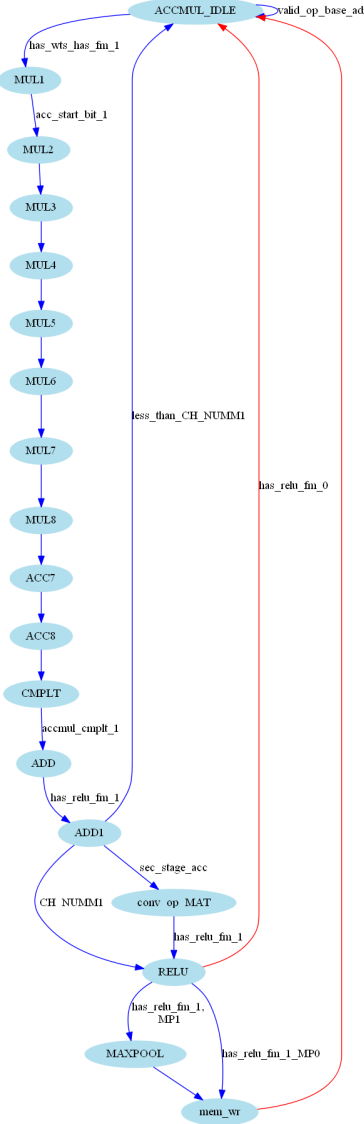


Figure 6.9: State Machine for acc-mul in Convolution

6.6 Design and Implementation of Fully Connected Layers

In a fully-connected layer, a single row matrix of length “R” is multiplied by a weight matrix of size R*C, C is the number of output neurons of the layer. The feature-map buffer (FM_BUF) and weight buffer (WT_BUF) are filled with values read from memory, which multiplied and

accumulated per column width number of times, which results in one output neuron. Every increment in row count results in a new output neuron that is written to memory after subjecting to ReLu and Clipping functions, as shown in Fig. 6.10.

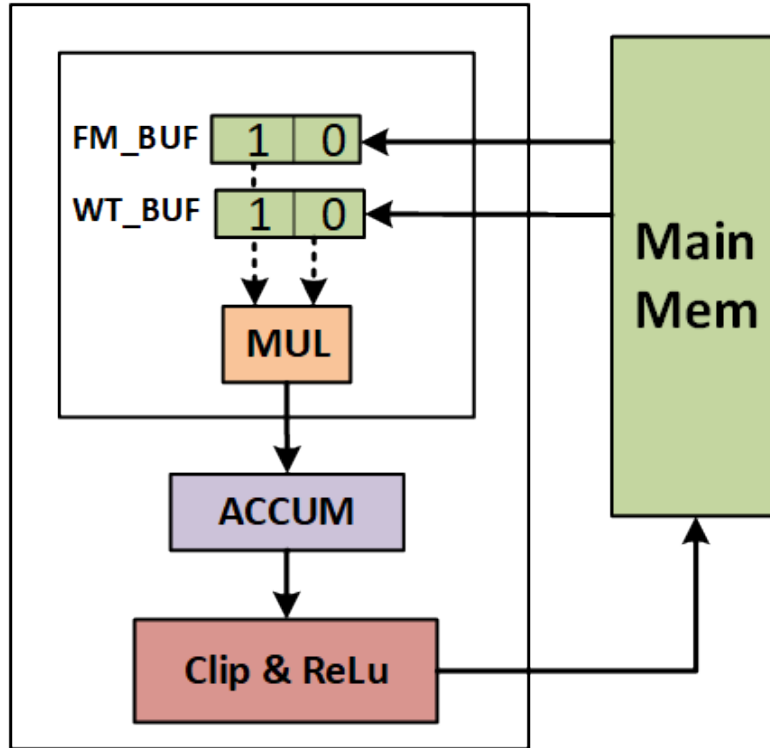


Figure 6.10: Structure of the Fully-Connected Layer

6.7 Basic Hardware Blocks

The basic hardware blocks are parameterized to incorporate all the bit-widths. The adder is designed, as shown in Fig. 6.11. The inputs are of width $BW7P$, which indicates the chosen bit-width plus seven. The bit width of inputs to adder are chosen to be plus seven because the highest threshold of 0.3 equivalent to 9860 in 16-bit Fixed Point representation, in every layer when added a maximum number of times that is 1408 times in both accelerators sums up to a number that requires a maximum of 23 bits including the sign bit.

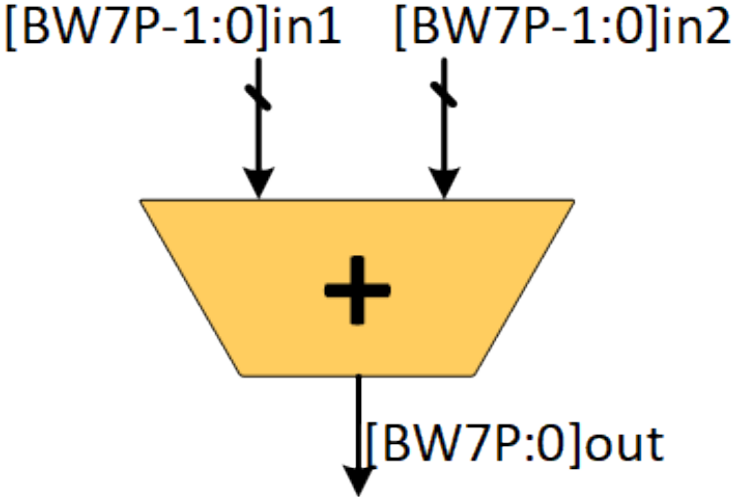


Figure 6.11: Hardware design for Adder

The accumulator is designed to take inputs with chosen bit-width as it is always used in the first stage of additions in convolution operation. The accumulated result can sum up to a value that can fit in a maximum bit width of $BW7P$. The accumulator starts accumulating the inputs on receiving a start-bit as high. The start-bit should be turned low to collect the final output and reset the accumulator. The output of the accumulator is fed back as an input to the accumulator, as shown in Fig. 6.12.

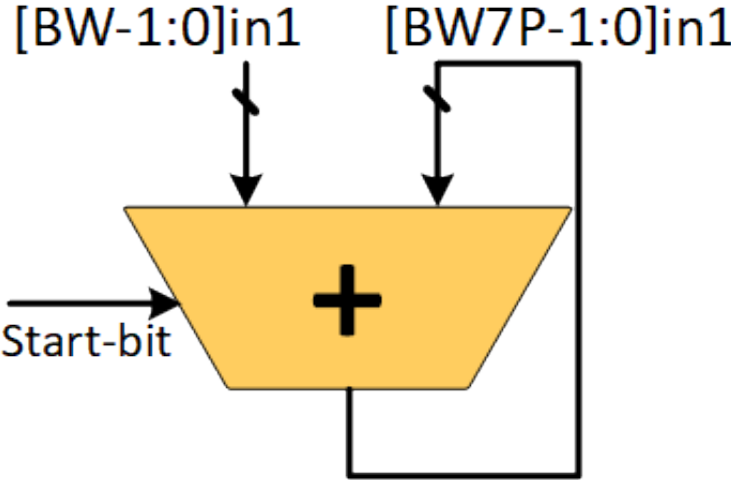


Figure 6.12: Hardware design for Accumulator

Another basic block includes a multiplier that is designed with inputs and output, as shown in Fig. 6.13. The inputs and outputs have the same bit-widths as the result of multiplying two fractional fixed-point numbers is always less than or equal to the inputs. The product of the two 16-bit numbers is stored in a 32-bit register and is shifted right by 16 times, saving the most significant bits and getting rid of the least significant bits.

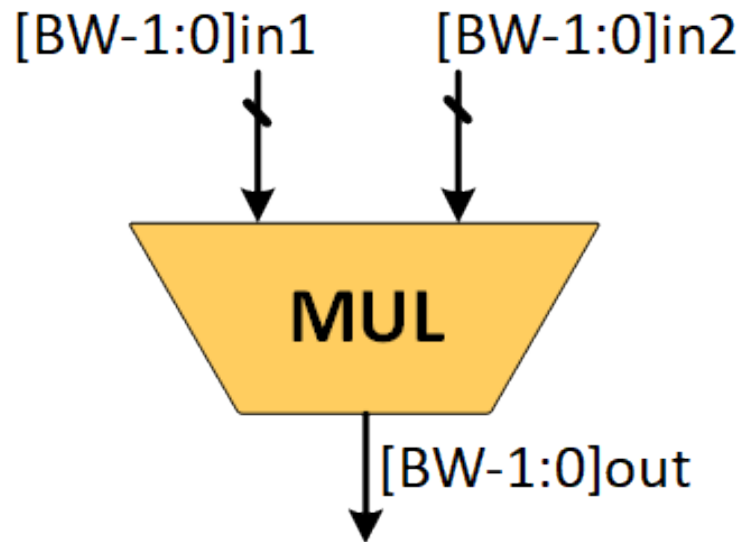


Figure 6.13: Hardware design for Multiplier

The ReLU and clipping operation happens in the same block named ReLU as both the operations include comparing with a threshold value. The input value is determined to be a positive or negative number. If the number is negative, it is replaced by a zero else; it is compared with a threshold of 0.3, i.e., 9860 in fixed-point representation. If the value is greater than 9860, it is clipped to 9860 and is left as it is if less than 9860. Fig. 6.14 depicts the hardware design of ReLU.

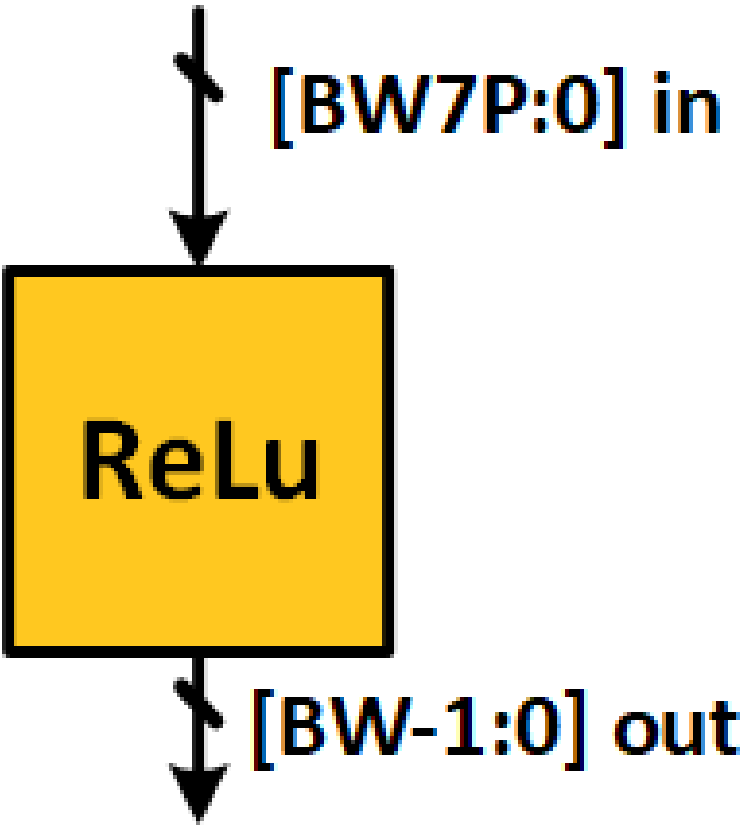


Figure 6.14: Hardware design for ReLU

The Max-pool operation has four inputs and one output, as shown in Fig. 6.15. The inputs and outputs have equal bit-width that is chosen for computation. The operation has three comparisons, where the two comparisons happen between in1 & in2 and in3 and in4. The next one happens between the result of the first two comparisons.

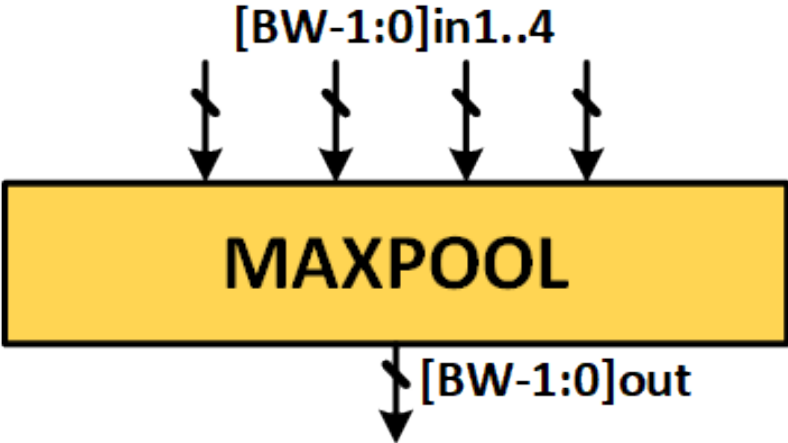


Figure 6.15: Hardware design for Max-pool

Chapter 7

Verification of IANET

A SystemVerilog test-bench environment is created to verify every single layer of the CNN. The trained network parameters are read from a CSV file using DPI C functions. The C function returns the values read from file. These values are collected and written to main memory from a SystemVerilog task utilizing the 32-bit wide wishbone bus. The C-model is created to imitate the RTL - model. The outputs from both the models are compared to see if they match. These models are later verified to match with vectors for each layer for both the CNN accelerators. The networks are also subjected to functional coverage and assertion coverage.

7.1 SystemVerilog

The IANET integrates ICNN and ACNN modules and is represented at the pin level in Fig. 7.1. This figure shows in detail the connections between each module and the wishbone bus. The naming convention followed in the figure is as follows: o_m_wb_icnn_adr meaning o indicates the output signal, m indicates the master signal, s indicates slave signal, wb stands for wishbone, icnn or acnn or mm stand for Image CNN, Audio CNN and main memory respectively, ack

means acknowledgment, err is for error, cyc is for cycle, stb for strobe, adr for address, and dat for data.

The ICNN and ACNN modules were verified prior to verifying IANET. As shown in the Fig. 7.1 IANET is the DUT while the wishbone bus, main memory and the host processor are instantiated in the test-bench. As mentioned in earlier chapters the weights and feature maps are stored in main memory.

The DPI functions in C are used to read the trained weights from CSV files and are written to main memory by the driver in SystemVerilog test-bench. The memory space is segregated into different sections for weights, input and output feature maps for image and audio processing units as shown in Fig. 6.5. The weights of different layers of CNN are organized into different directories based on the bit-width. These weights are written to their respective locations in the main memory. The image and audio dataset are written to CSV files along with their labels using Python scripts. One image and one audio file are written to main memory at once.

Once the required data is set up for the system to start up and run, the LAYER_CONFIG slave ports of both the ICNN and ACNN networks are written with a valid base address and a start bit concatenated in most significant bit of the 32-bit data. By setting the start-bit high, the accelerator is set to go. In order to achieve layer by layer verification, the LAYER_CMPLT slave port is monitored which when high indicates the completion of a single layer. When the LAYER_CMPLT port is high, the output feature map of that layer can be read from main memory in test-bench. The golden-reference model is run in parallel to running the hardware model or the DUT. The results from the golden reference model are stored into a static array declared in scoreboard. This array with the output feature map of current layer is read back as input for next layer. It is also read by the monitor to compare the output of every layer of reference model with the output of DUT.

The accelerators can be interrupted or paused by stopping the CNN at current layer. This can

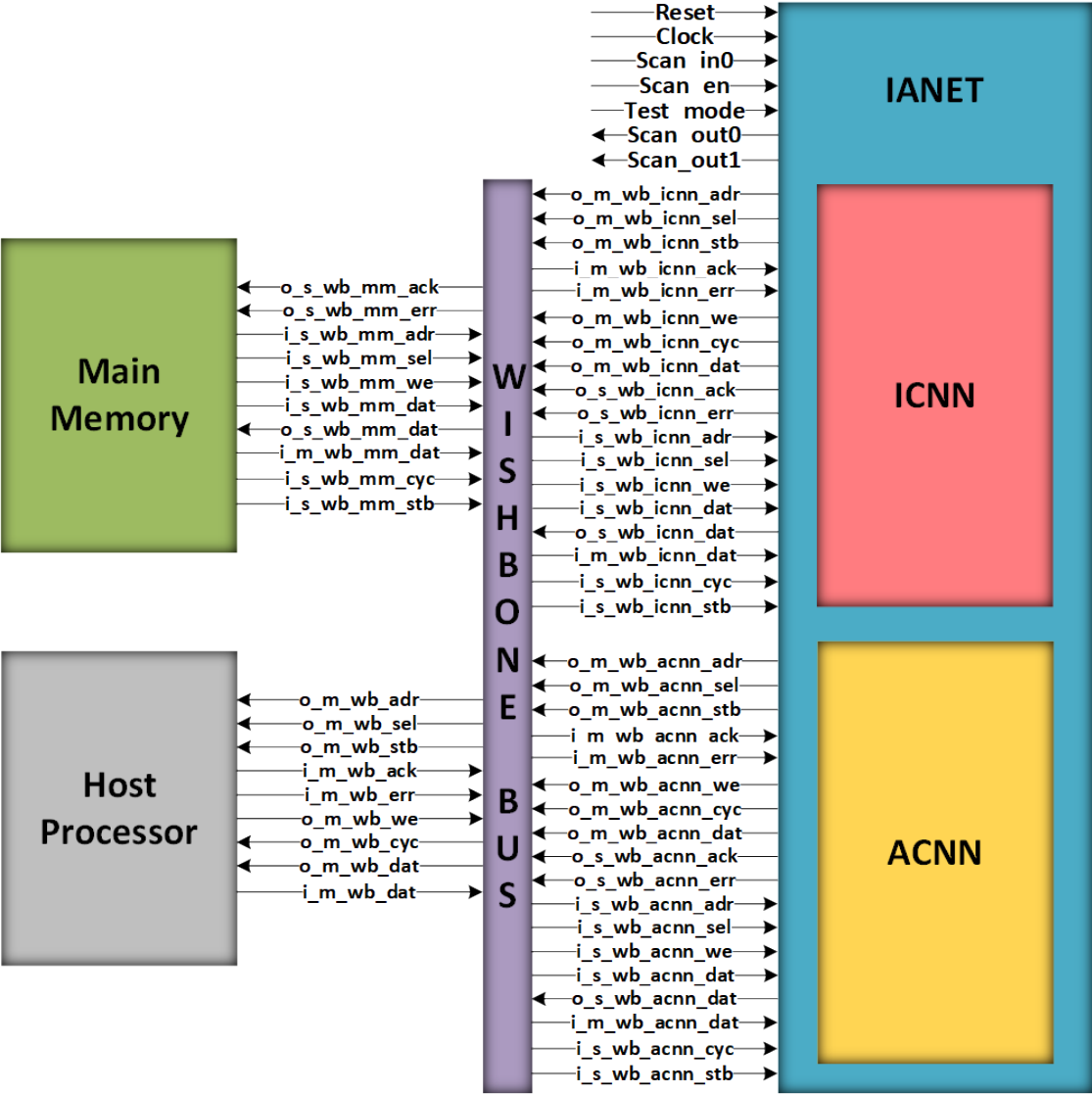


Figure 7.1: Pin Level Representation of IANET

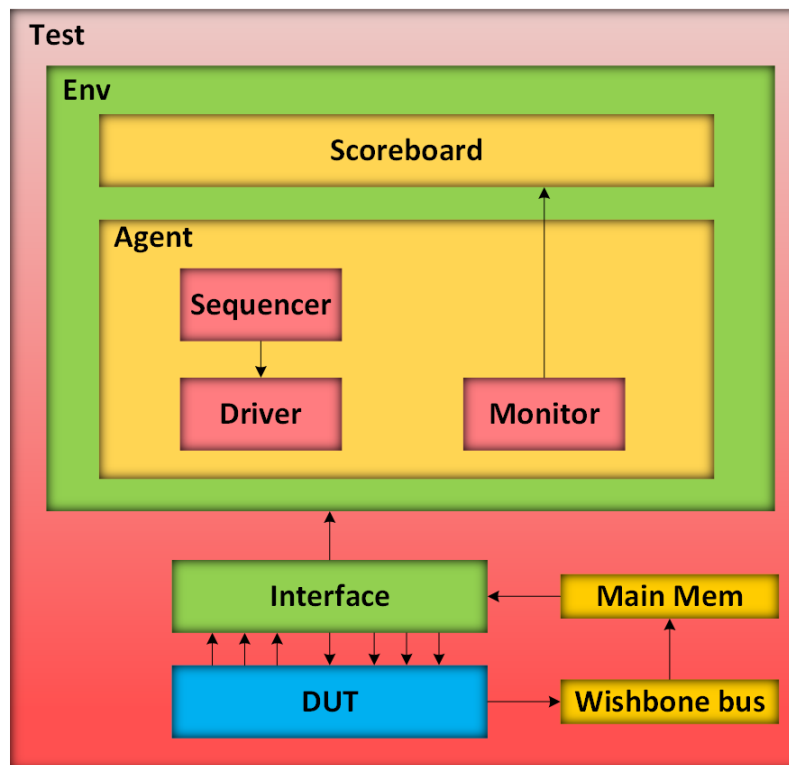


Figure 7.2: SystemVerilog Test-bench Block Diagram

be done by clearing the start-bit by using the LAYER_CONFIG port. The accelerators indicate if they are done processing a single sample by setting the DONE port.

Fig. 7.2 shows that block diagram of the test-bench. The scoreboard has all the common variables declared. The sequencer is used to generate random input values that are fed to the network as input feature map. The Monitor compares the outputs from reference model and DUT to display the results. The Driver performs all the DUT related transactions. The monitor compares the label read from CSV file with the label computed by reference model and the DUT to ensure the accurate result. A functional coverage was performed on the data bus and the address bus with bins restricted to values within range.

Chapter 8

Results

The two networks, ICNN and ACNN, were run in inference mode for all the fixed-point bit-widths ranging from 8-bits to 16-bits. The inference mode was run on all the evaluation and testing datasets. The accuracy at every fixed-point representation ranging from 8-bits to 16-bits was recorded for ICNN and ACNN as shown in Fig. 8.1. The accuracy during inference for both networks is composed at 16 bit fixed point representation and gradually alters for lower bit widths as inferred from Fig. 8.1.

The hardware accelerators are designed and Table 8.1 summarizes the implementation results. The area of ICNN is observed to be approximately twice that of ACNN. The IANET combines the two accelerators and includes the wishbone bus that establishes all the necessary connections between the modules. Note in the case of IANET, while ICNN will operate at 180 MHz, an integrated clock divider will produce the 20 MHz necessary for ACNN.

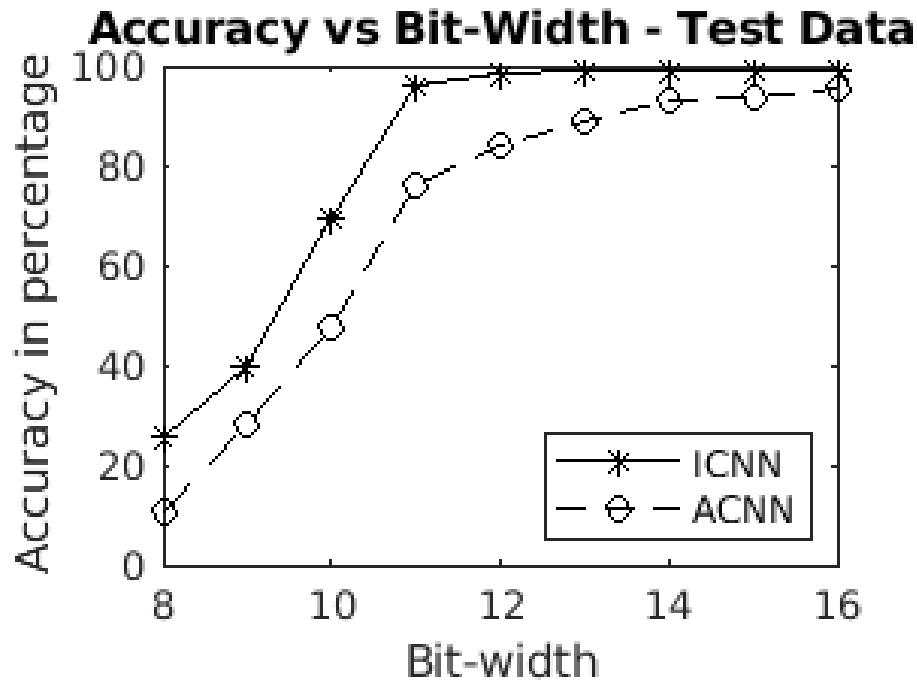


Figure 8.1: Accuracy vs Fixed Point representation ICNN and ACNN

Table 8.1: 28 nm Implementation Results

Parameter	ICNN	ACNN	IANET
Total Area (μm^2)	2,137,738.50	1,199,745.10	3,838,106.14
Gate Count	1,401,921.00	786,789.00	2,517,016.00
Power (mW)	45.18 @ 180 MHz	13.48 @ 20 MHz	63.36 @ 180 MHz
Worst Case Delay (ns)	0.2297	0.2297	0.2297
Maximum Frequency of Operation	4.3 GHz	4.3 GHz	4.3 GHz
DFT Coverage (%)	100.00	100.00	100.00

The ICNN achieves image frame rate of 30 fps at 180 MHz; the ACNN achieves an audio frame rate of 1 fps at 20 MHz. At 50 MHz each clock cycle is of 20 ns time-period, 10 ns at 100 MHz, 5ns at 200 MHz, so on, and so forth. The accelerators can be run at any frequency

up to a maximum of 4.3 GHz based on the application requirements. Table 8.2 summarizes the performance of the accelerators.

Table 8.2: Accelerator Performance Metrics

Parameter	ICNN	ACNN
Frames per second	30	1
Time per frame (<i>ms</i>)	33.3333	1,000.0000
System Processing Time per Frame (<i>ms</i>)	5	5
Accelerator Processing Time (<i>ms</i>)	28.3333	995.0000
Accelerator Computations ¹	16,214,393	28,249,662
Accelerator Cycles	5,099,994	19,900,000
Time per Cycle (<i>ns</i>)	5.5555	50.0000
Accelerator Frequency	180 MHz	20 MHz

The memory requirements of the accelerators vary based on the bit-width of the fixed-point numbers used for computation. The Table 8.3 summarizes the memory requirement of each accelerator according to the bit-width. The required memory of each accelerator considers the storage space required for network parameters, intermediate results of each layer, one input sample, and host processor configuration space.

¹ Accelerator computations are performed in parallel, therefore computations will be greater than cycles.

Table 8.3: Memory Requirements for Accelerators as per the Bit Width of Fixed Point Implementation

Bit-Width	ICNN (MB)	ACNN (MB)
16	1.5	1.8
15	1.4	1.7
14	1.3	1.6
13	1.2	1.5
12	1.1	1.4
11	1.0	1.3

Chapter 9

Conclusion

The image and audio processing units were successfully developed in hardware that help close the design gap in the decision making process between a self-driving car and a human driving a car. There are several hardware CNN accelerator designs, that classify image and audio information on different platforms but are not implemented in combination on a single die till date. Also, the hardware CNN accelerators in previous designs utilize more hardware resources and run at higher frequencies comparatively to achieve high speed processing. However, not all the applications require such high speed processing resulting in high power consumption than necessary. And although the networks can be implemented on tradition x86 CPU's or GPU's, even for mobile versions these components have a typical TDP ≥ 10 watts. The IANET accelerator is designed for low power applications such as low power, high performance Arm based SoC's.

Through experimentation of different architectures of CNN's, the network size was reduced. The number of trainable parameters was reduced by removing the bias parameters and by reducing the size of the input sample. The accuracy was maintained at lower bit-width fixed-point representations by introducing clipping during training and inference. Several threshold values were experimented to determine the best suitable clipping threshold value.

The present investigations identified the gap in existing technology and successfully implemented efficient integrated hardware for the two accelerators achieving an image frame rate of 30 fps at 180 MHz for the ICNN; the ACNN processes a 1 second frame of audio data sampled at 44.1 KHz at 20 MHz. The accelerators were verified using a modular SystemVerilog test-bench embedded with a golden C-reference model.

The accuracy of 96% and 93% is achieved with a 16-bit fixed-point implementation of ICNN and ACNN in hardware respectively. Note that the fixed-point bit-width can be further reduced with equivalent accuracy once training using 16 bit floating point precision in Tensorflow is achieved.

This project can have a wide variety of applications based on the platform requirements. Its primary purpose of design was to deploy this kind of IC in the field of automotive industry where the combination of vision and audio information can together provide valuable information to the end user.

9.1 Future Work

There is a broad scope of improvement to this project in various aspects, such as reducing the network size, fixed-point representation, optimizing the hardware for CNN, combinations of algorithms, etc.

- The YOLO algorithm can be applied as a pre-processing technique before feeding the inputs to the network. This will significantly improve the accuracy of object detection and classification. Another implementation of the project could be to implement hardware for the YOLO algorithm directly. The RNN and LSTM algorithms can be explored for audio classification, and the results from hardware implementation of the RNN or LSTM model should be compared with the CNN implementation of the audio processing unit.

The above-discussed implementations can contribute interesting comparisons to research and development in this field.

- It is known from the background research that CNN can learn features of image and audio data. This project is using two separate CNN's for Audio and Image classification. However, it is unknown whether a single CNN architecture can learn features of image and audio data at the same time. Through trial and error methods, an architecture of CNN to learn combined features of the image and audio data can be obtained in the future. However, in this case, the system will be limited to one input at a time, making it inept for this particular application.
- In terms of architecture, through experimentation, different combinations of convolution and fully-connected layers can be used. This architecture makes use of the max-pool layer, which can be substituted by average pooling for experimentation and research. The stride length can be increased, which will result in reduced computations.
- The audio network in the future implementations should be trained for collision sound effects in the future so that it can indicate or convey the impact of collision to a deaf or disabled or an impaired person.
- In this project, the ACNN classifies mono audio files into their respective classes. However, in some scenarios, multiple microphones act as sources to a single audio file resulting in a stereo channel audio file. In this case, the direction of the audio data can be determined by training the CNN accordingly.
- A sophisticated testbench with several features to verify the entire hardware can be developed in UVM.

-
- Other sensory information (such as Lidar) can be taken as input to the system with a corresponding processing unit in addition to audio and visual data to make the decision making the process better, getting closer to imitating the working of the human brain. The work in [40] can be taken as inspiration.

References

- [1] M. Zhu, Q. Kuang, C. Yang, and J. Lin. Optimization of convolutional neural network hardware structure based on FPGA. pages 1797–1802, May 2018. doi:10.1109/ICIEA.2018.8398000.
- [2] L. Cavigelli and L. Benini. Origami: A 803-GOp/s/W Convolutional Network Accelerator. *IEEE Transactions on Circuits and Systems for Video Technology*, 27(11):2461–2475, Nov 2017. doi:10.1109/TCSVT.2016.2592330.
- [3] Pierce T. Hamilton. Communicating through Distraction: A Study of Deaf Drivers and Their Communication Style in a Driving Environment. Master’s thesis, Rochester Institute of Technology, Nov 2015. URL: <https://scholarworks.rit.edu/theses/8917/>.
- [4] Y. Zhang, C. Hong, and W. Charles. An efficient real time rectangle speed limit sign recognition system. pages 34–38, June 2010. doi:10.1109/IVS.2010.5548140.
- [5] M. Al-Qizwini, I. Barjasteh, H. Al-Qassab, and H. Radha. Deep learning algorithm for autonomous driving using GoogLeNet. pages 89–96, June 2017. doi:10.1109/IVS.2017.7995703.
- [6] L. Mioulet, D. Tsishkou, R. Bendahan, and F. Abad. Efficient combination of Lidar inten-

- sity and 3D information by DNN for pedestrian recognition with high and low density 3D sensor. pages 257–263, 2017.
- [7] F. Piewak, T. Rehfeld, M. Weber, and J. M. Zollner. Fully convolutional neural networks for dynamic object detection in grid maps. pages 392–398, 2017.
- [8] V. Fremont, S. A. R. Florez, and B. Wang. Mono-vision based moving object detection in complex traffic scenes. pages 1078–1084, 2017.
- [9] H. Jaspers, M. Himmelsbach, and H. Wuensche. Multi-modal local terrain maps from vision and LiDAR. pages 1119–1125, 2017.
- [10] M. Kilicarslan and J. Y. Zheng. Direct vehicle collision detection from motion in driving video. pages 1558–1564, 2017.
- [11] C. Zhu, Y. Li, Y. Liu, Z. Tian, Z. Cui, C. Zhang, and X. Zhu. Road Scene Layout Reconstruction based on CNN and its Application in Traffic Simulation. pages 480–485, 2019.
- [12] S. Schubert, P. Neubert, J. Poschmann, and P. Pretzel. Circular Convolutional Neural Networks for Panoramic Images and Laser Data. pages 653–660, 2019.
- [13] R. Fan, M. J. Bocus, Y. Zhu, J. Jiao, L. Wang, F. Ma, S. Cheng, and M. Liu. Road Crack Detection Using Deep Convolutional Neural Network and Adaptive Thresholding. pages 474–479, 2019.
- [14] N. De Rita, A. Aimar, and T. Delbruck. CNN-based Object Detection on Low Precision Hardware: Racing Car Case Study. pages 647–652, 2019.
- [15] Y. Kim and D. Kum. Deep Learning based Vehicle Position and Orientation Estimation via Inverse Perspective Mapping Image. pages 317–323, 2019.

-
- [16] Tomasz Nowak, Michal Nowicki, Krzysztof Cwian, and Piotr Skrzypczynski. How to Improve Object Detection in a Driver Assistance System Applying Explainable Deep Learning. pages 226–231, 2019.
- [17] M. Weber, M. Furst, and J. M. Zollner. Direct 3D Detection of Vehicles in Monocular Images with a CNN based 3D Decoder. pages 417–423, 2019.
- [18] R. Theodosc, D. Denis, C. Blanc, T. Chateau, and P. Checchin. Vehicle Detection based on Deep Learning Heatmap Estimation. pages 108–113, 2019.
- [19] F. Milletari, N. Navab, and S. Ahmadi. V-Net: Fully Convolutional Neural Networks for Volumetric Medical Image Segmentation. pages 565–571, Oct 2016. [doi:10.1109/3DV.2016.79](https://doi.org/10.1109/3DV.2016.79).
- [20] K. Nguyen, C. Fookes, and S. Sridharan. Improving deep convolutional neural networks with unsupervised feature learning. pages 2270–2274, Sep 2015. [doi:10.1109/ICIP.2015.7351206](https://doi.org/10.1109/ICIP.2015.7351206).
- [21] Tomas Mikolov, Martin Karafiat, Lukas Burget, Jan Cernocky, and Sanjeev Khudanpur. Recurrent neural network based language model. *INTERSPEECH-2010*, 1045-1048, 2010.
- [22] T. Hayashi, S. Watanabe, T. Toda, T. Hori, J. Le Roux, and K. Takeda. Duration-Controlled LSTM for Polyphonic Sound Event Detection, 2017.
- [23] Antoni B. Chan Tom LH, Li and Andy H. W. Chun. Automatic Musical Pattern Feature Extraction Using Convolutional Neural Network. *IMECS*, 2010.
- [24] S. Hershey, S. Chaudhuri, D. P. W. Ellis, J. F. Gemmeke, A. Jansen, R. C. Moore, M. Plakal, D. Platt, R. A. Saurous, B. Seybold, M. Slaney, R. J. Weiss, and K. Wil-

- son. CNN architectures for large-scale audio classification. pages 131–135, March 2017. [doi:10.1109/ICASSP.2017.7952132](https://doi.org/10.1109/ICASSP.2017.7952132).
- [25] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E. Hinton. ImageNet Classification with Deep Convolutional Neural Networks. *Advances in Neural Information Processing Systems*, pp. 1097-1105, pages 1797–1802, May 2012. [doi:10.1109/ICIEA.2018.8398000](https://doi.org/10.1109/ICIEA.2018.8398000).
- [26] K. Simonyan and A. Zisserman. Very Deep Convolutional Neural Networks for Large-Scale Image Recognition. *arXiv:1409.1556*, 2014.
- [27] Christian Szegedy, Vincent Vanhoucke, Sergey Ioffe, Jon Shlens, and Zbigniew Wojna. Re-thinking the Inception Architecture for Computer Vision. *The IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, June 2016.
- [28] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep Residual Learning for Image Recognition. *The IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, June 2016.
- [29] Inkyu Choi Soo Hyun Bae and Nam Soo Kim. Acoustic Scene Classification Using Combination of LSTM and CNN. *Detection and Classification of Acoustic Scenes and Events*, 2016.
- [30] Wei Qi, Lie Gu, Hao Jiang, Xiang-Rong Chen, and Hong-Jiang Zhang. Integrating visual, audio and text analysis for news video. 3:520–523 vol.3, 2000.
- [31] Y. Lin and T. S. Chang. Data and Hardware Efficient Design for Convolutional Neural Network. *IEEE Transactions on Circuits and Systems I: Regular Papers*, 65(5):1642–1651, May 2018. [doi:10.1109/TCSI.2017.2759803](https://doi.org/10.1109/TCSI.2017.2759803).

-
- [32] Jonathan Krause, Michael Stark, Jia Deng, and Li Fei-Fei. 3D Object Representations for Fine-Grained Categorization. In *4th IEEE Workshop on 3D Representation and Recognition, at ICCV 2013 (3dRR-13)*, Sydney, Australia, 2013. URL: "https://ai.stanford.edu/~jkrause/cars/car_dataset.html".
- [33] Mohamed Alaa El-Dien Aly. Caltech Lanes Dataset. URL: <http://www.mohamedaly.info/datasets/caltech-lanes>.
- [34] Caltech. Caltech Pedestrian Detection Benchmark. URL: http://www.vision.caltech.edu/Image_Datasets/CaltechPedestrians/.
- [35] Morten Jensen and Mark Philip. LISA Traffic Light Dataset. URL: <https://www.kaggle.com/mbornoe/lisa-traffic-light-dataset>.
- [36] J. Salamon, C. Jacoby, and J. P. Bello. A Dataset and Taxonomy for Urban Sound Research. In *22nd ACM International Conference on Multimedia*, pages 1041–1044, Orlando, FL, USA, 2014. URL: <https://urbansounddataset.weebly.com/>.
- [37] Online tool for sound tracks. URL: <https://audio.online-convert.com/convert-to-wav>.
- [38] Sebastian Ruder. An overview of gradient descent optimization algorithms. *arXiv:1609.04747*, 55(6):720–731, 2017.
- [39] Joseph C. Kolecki. An Introduction to Tensors for Students of Physics and Engineering, 2002.
- [40] U. E. Manawadu, M. Kamezaki, M. Ishikawa, T. Kawano, and S. Sugano. A multimodal human-machine interface enabling situation-adaptive control inputs for highly automated vehicles. pages 1195–1200, 2017.

Appendix I

Source Code

I.1 TensorFlow Image CNN

```
1 from __future__ import absolute_import
2 from __future__ import division
3 from __future__ import print_function
4 from random import shuffle
5 import glob
6 import sys
7 import cv2
8 import tensorflow as tf
9 import xlswriter
10 #
    =====
11 num_inputs = 690
```

```
12 num_classes = 4
13 dropout = 0.5
14 dataset_path = "../.. / datasets / tfrecords /"
15 # =====TRAIN NETWORK
    =====
16
17
18 def parser(record):
19     keys_to_features = {
20         "image_raw": tf.FixedLenFeature([], tf.string),
21         "label": tf.FixedLenFeature([], tf.int64)
22     }
23     parsed = tf.parse_single_example(record, keys_to_features)
24     image = tf.decode_raw(parsed["image_raw"], tf.uint8)
25     # Normalizing the image
26     image = tf.cast(image, tf.float32)/256
27     label = tf.cast(parsed["label"], tf.int32)
28
29     return {'image': image}, label
30
31
32 def input_fn(filenamees):
33     with tf.name_scope("importingDataset"):
34         dataset = tf.data.TFRecordDataset(filenamees=filenamees,
            num_parallel_reads=40)
```

```
35     # print(" dataset: ", dataset)
36     dataset = dataset.apply(
37         tf.data.experimental.shuffle_and_repeat(1024, 2)
38     )
39     dataset = dataset.apply(
40         tf.data.experimental.map_and_batch(parser, 30)
41     )
42     #
43     # dataset = dataset.map(parser, num_parallel_calls=12)
44     # dataset = dataset.batch(batch_size=1000)
45     dataset = dataset.prefetch(buffer_size=2)
46     # FIFO Buffer
47     return dataset
48
49
50 def train_input_fn():
51     return input_fn( filenames=[ dataset_path+" train / train .
52         tfrecords", dataset_path+" test / test . tfrecords" ])
53
54 def val_input_fn():
55     return input_fn( filenames=[ dataset_path+" val / val . tfrecords"
56         ])
57
```

```
58 def test_input_fn():
59     return input_fn(filenamees=[dataset_path+"test/test.
        tfrecords"])
60
61
62 feature_spec = {'image': tf.FixedLenFeature([64,64,3], tf.
        float32)}
63
64 # serving_input_receiver_fn = tf.estimator.export.
        build_parsing_serving_input_receiver_fn(feature_spec)
65
66
67 def serving_input_receiver_fn():
68     serialized_tf_example = tf.placeholder(dtype=tf.float32,
        shape=[64,64,3], name='input_serve')
69     receiver_tensors = {'examples': serialized_tf_example}
70     features = {'image': serialized_tf_example}
71     return tf.estimator.export.ServingInputReceiver(features,
        receiver_tensors)
72
73
74 def convolution_img(x, w, strides=1):
75     x = tf.nn.conv2d(x, w, strides=[1, strides, strides, 1],
        padding="VALID")
76     x = tf.clip_by_value(x, -0.3, 0.3)
```

```
77     # x = tf.nn.bias_add(x, b)
78     return tf.nn.relu(x)
79
80
81 def max_pooling_layer(x, k=3, name="MaxPool"):
82     return tf.nn.max_pool(x, ksize=[1, k, k, 1], strides=[1, k,
83         k, 1], padding="VALID")
84
85 def model_fn(features, labels, mode, params):
86
87     W = {"w1": tf.Variable(tf.clip_by_value(tf.random.normal
88         ([3, 3, 3, 8]), -0.3, 0.3), name='w1'),
89         "w2": tf.Variable(tf.clip_by_value(tf.random.normal
90         ([3, 3, 8, 16]), -0.3, 0.3), name='w2'),
91         "w3": tf.Variable(tf.clip_by_value(tf.random.normal
92         ([3, 3, 16, 32]), -0.3, 0.3), name='w3'),
93         "w4": tf.Variable(tf.clip_by_value(tf.random.normal
94         ([3, 3, 32, 64]), -0.3, 0.3), name='w4'),
95         "w5": tf.Variable(tf.clip_by_value(tf.random.normal
96         ([1024, 512]), -0.3, 0.3), name='w5'),
97         "w6": tf.Variable(tf.clip_by_value(tf.random.normal
98         ([512, 256]), -0.3, 0.3), name='w6'),
99         "w7": tf.Variable(tf.clip_by_value(tf.random.normal
100        ([256, 128]), -0.3, 0.3), name='w7'),
```

```
94         "w8": tf.Variable(tf.clip_by_value(tf.random.normal
95             ([128, num_classes]), -0.3, 0.3), name='w8'})
96     with tf.name_scope("importingImage"):
97         net = features["image"]
98         net = tf.identity(net, name="input_tensor")
99         net = tf.reshape(net, [-1, 64, 64, 3])
100        # net = tf.identity(net, name="input_tensor_after")
101        # tf.summary.image('IMAGE', net)
102        # convolution
103        with tf.name_scope("ConvLayer1"):
104            net = convolution_img(net, W["w1"], strides=1)
105            print("Layer1 shape: ", net.shape)
106            net = max_pooling_layer(net, k=2)
107            tf.summary.histogram('W1', W["w1"])
108            print("Layer1 MP shape: ", net.shape)
109
110        with tf.name_scope("ConvLayer2"):
111            net = convolution_img(net, W["w2"], strides=1)
112            print("Layer2 shape: ", net.shape)
113            net = max_pooling_layer(net, k=2)
114            tf.summary.histogram('W2', W["w2"])
115            print("Layer2 MP shape: ", net.shape)
116
117        with tf.name_scope("ConvLayer3"):
```



```
118         net = convolution_img(net, W["w3"], strides=1)
119         print("Layer3 shape: ", net.shape)
120         net = max_pooling_layer(net, k=2)
121         tf.summary.histogram('W3', W["w3"])
122         print("Layer3 MP shape: ", net.shape)
123
124     with tf.name_scope("ConvLayer4"):
125         net = convolution_img(net, W["w4"], strides=1)
126         tf.summary.histogram('W4', W["w4"])
127         print("Layer4 shape: ", net.shape)
128
129     # fully connected
130     with tf.name_scope("FC_1"):
131         net = tf.reshape(net, [-1, W["w5"].get_shape().as_list
132                               ()[0]])
132         net = tf.matmul(net, W["w5"])
133         net = tf.clip_by_value(net, -0.3, 0.3)
134         net = tf.nn.relu(net)
135         tf.summary.histogram('W5', W["w5"])
136         print("Layer5 shape: ", net.shape)
137
138     with tf.name_scope("FC_2"):
139         net = tf.matmul(net, W["w6"])
140         net = tf.clip_by_value(net, -0.3, 0.3)
141         net = tf.nn.relu(net)
```

```
142         tf.summary.histogram('W6', W["w6"])
143         print("Layer6 shape: ", net.shape)
144
145     with tf.name_scope("FC_3"):
146         net = tf.matmul(net, W["w7"])
147         net = tf.clip_by_value(net, -0.3, 0.3)
148         # net = tf.add(net, B["b7"]/8)
149         net = tf.nn.relu(net)
150         tf.summary.histogram('W7', W["w7"])
151         print("Layer7 shape: ", net.shape)
152
153     # Regularization
154     # Dropout
155     # with tf.name_scope("DropOut"):
156         # net = tf.nn.dropout(net, dropout)
157     # output
158     with tf.name_scope("Output"):
159         net = tf.matmul(net, W["w8"])
160         net = tf.clip_by_value(net, -0.8, 0.8)
161         tf.summary.histogram('W8', W["w8"])
162         print("Layer8 shape: ", net.shape)
163
164     with tf.name_scope("SoftmaxPrediction"):
165         logits = net
166         # out = tf.clip_by_value(logits, 1e-10, 100.0)
```

```
167     y_pred = tf.nn.softmax(logits=logits)
168     y_pred = tf.identity(y_pred, name="output_pred")
169     y_pred_cls = tf.argmax(y_pred, axis=1)
170     y_pred_cls = tf.identity(y_pred_cls, name="output_cls")
171     tf.summary.histogram('y_pred_cls', y_pred_cls)
172
173     if mode == tf.estimator.ModeKeys.PREDICT:
174         spec = tf.estimator.EstimatorSpec(mode=mode,
175                                           predictions=
176                                               y_pred_cls)
177
178     else:
179         with tf.name_scope("AdamOptimizer"):
180             optimizer = tf.train.AdamOptimizer(learning_rate=
181                                                 params["learning_rate"])
182
183         with tf.name_scope("CrossEntropy"):
184             # labels = tf.one_hot(indices=labels, depth=4)
185             cross_entropy = tf.nn.
186                 sparse_softmax_cross_entropy_with_logits(labels=
187                                                           labels, logits=logits)
188             tf.summary.histogram('cross_entropy', cross_entropy
189                                 )
190
191             # cross_entropy = tf.nn.
192                 sparse_softmax_cross_entropy_with_logits(labels=
193                                                           labels, logits=out)
194
195         with tf.name_scope("ReduceCrossEntropy"):
```

```
232 converter = tf.lite.TFLiteConverter.from_saved_model(path)
233 converter.post_training_quantize=True
234 # converter.optimizations = [tf.lite.Optimize.Default]
235 converter.target_ops = [tf.lite.OpsSet.TFLITE_BUILTINS, tf.lite
    .OpsSet.SELECT_TF_OPS]
236 tflite_model = converter.convert()
237 open("./temp/model.tflite", "wb").write(tflite_model)
238
239 # converter.post_training_quantize=True
240 # tflite_quantized_model=converter.convert()
241 # open("quantized_model.tflite", "wb").write(
    tflite_quantized_model)"""
```

Listing I.1: TensorFlow Image CNN

I.2 TensorFlow Audio CNN

```
1  """=====
2  added ReLu in 7 and 8 layers
3  clipped weights from -0.5 to 0.5
4  clipping after convolution - -0.5 to 0.5
5  learning rate: 10^-4
6
7  Observations:
8  accuracy picked up gradually.
9  (-0.5, 0.5) range used to see if it
10 gives better results at low bit width
11
12 making another version because aim is
13 to reduce number of multiplications and additions
14 in first fully connected layer
15
16 Saving this version, to run fixed point and check accuracy.
17 """
18 from __future__ import absolute_import
19 from __future__ import division
20 from __future__ import print_function
21
22 import tensorflow as tf
23 import sys
```

```
24 sys.path.insert(1, '../.../Lib')
25 import inspect_checkpoint_csv as ic
26
27 num_classes = 3
28 path_to_train_dataset = '../.../datasets/csv/train/flt_32/train.
    csv'
29 path_to_test_dataset = '../.../datasets/csv/test/flt_32/test.csv
    ,
30 path_to_val_dataset = '../.../datasets/csv/val/flt_32/val.csv'
31 path_to_label = '../.../datasets/csv/labels/Y_'
32
33 def _int64_feature(value):
34     return tf.train.Feature(int64_list=tf.train.Int64List(value
        =[value]))
35
36
37 def _bytes_feature(value):
38     return tf.train.Feature(bytes_list=tf.train.BytesList(value
        =[value]))
39
40
41 def convolution(x, w, strides=1):
42     x = tf.nn.conv2d(x, w, strides=[1, strides, strides, 1],
        padding="VALID")
43     x = tf.clip_by_value(x, -0.3, 0.3)
```



```
44     return tf.nn.relu(x)
45
46
47 def max_pooling_layer(x, k=3, name="MaxPool"):
48     return tf.nn.max_pool(x, ksize=[1, k, k, 1], strides=[1, k,
49         k, 1], padding="VALID")
50
51 def model_fn(features, labels, mode, params):
52     W = {"w1": tf.Variable(tf.clip_by_value(tf.random_normal
53         ([3, 3, 1, 4]), -0.3, 0.3), name='w1'),
54         "w2": tf.Variable(tf.clip_by_value(tf.random_normal
55         ([3, 3, 4, 8]), -0.3, 0.3), name='w2'),
56         "w3": tf.Variable(tf.clip_by_value(tf.random_normal
57         ([3, 3, 8, 16]), -0.3, 0.3), name='w3'),
58         "w4": tf.Variable(tf.clip_by_value(tf.random_normal
59         ([3, 3, 16, 32]), -0.3, 0.3), name='w4'),
60         "w5": tf.Variable(tf.clip_by_value(tf.random_normal
61         ([3, 3, 32, 32]), -0.3, 0.3), name='w5'),
62         "w6": tf.Variable(tf.clip_by_value(tf.random_normal
63         ([1408, 512]), -0.3, 0.3), name='w6'),
64         "w7": tf.Variable(tf.clip_by_value(tf.random_normal
65         ([512, 256]), -0.3, 0.3), name='w7'),
66         "w8": tf.Variable(tf.clip_by_value(tf.random_normal
67         ([256, 3]), -0.3, 0.3), name='w8')}
```

```
60
61     W["w1"] = tf.cast(W["w1"], tf.float32)
62     W["w2"] = tf.cast(W["w2"], tf.float32)
63     W["w3"] = tf.cast(W["w3"], tf.float32)
64     W["w4"] = tf.cast(W["w4"], tf.float32)
65     W["w5"] = tf.cast(W["w5"], tf.float32)
66     W["w6"] = tf.cast(W["w6"], tf.float32)
67     W["w7"] = tf.cast(W["w7"], tf.float32)
68     W["w8"] = tf.cast(W["w8"], tf.float32)
69
70     with tf.name_scope("importing_wav"):
71         net = features["wav"]
72         net = tf.identity(net, name="input_tensor")
73         print("Input tensor shape: ", net.shape)
74         net = tf.reshape(net, [-1, 13, 99, 1])
75         net = tf.identity(net, name="input_tensor_after")
76         tf.summary.image('image', net)
77
78     with tf.name_scope("Conv_layer_1"):
79         net = convolution(net, W["w1"], strides=1)
80         print("Conv_layer_1 shape: ", net.shape)
81         # tf.summary.image('features1', feature_plt)
82         tf.summary.histogram('W1', W["w1"])
83
84     with tf.name_scope("Conv_layer_2"):
```

```
85     net = convolution(net, W["w2"], strides=1)
86     print("Conv_layer_2 shape: ", net.shape)
87     # tf.summary.image('features2', net[1, :, :, 3])
88     tf.summary.histogram('W2', W["w2"])
89
90     with tf.name_scope("Conv_layer_3"):
91         net = convolution(net, W["w3"], strides=1)
92         # net = max_pooling_layer(net, k=2)
93         print("Conv_layer_3 shape: ", net.shape)
94         # tf.summary.image('features3', net[1, :, :, 3])
95         tf.summary.histogram('W3', W["w3"])
96
97     with tf.name_scope("Conv_layer_4"):
98         net = convolution(net, W["w4"], strides=1)
99         print("Conv_layer_4 shape: ", net.shape)
100        # tf.summary.image('features4', net[1, :, :, 3])
101        tf.summary.histogram('W4', W["w4"])
102
103    with tf.name_scope("Conv_layer_5"):
104        net = convolution(net, W["w5"], strides=1)
105        print("Conv_layer_5 shape: ", net.shape)
106        # tf.summary.image('features4', net[1, :, :, 3])
107        tf.summary.histogram('W5', W["w5"])
108
109    with tf.name_scope("Max_pool_5"):
```

```
110     net = max_pooling_layer(net, k=2)
111     print("Pool_layer_5 shape: ", net.shape)
112     # tf.summary.image('features5', net)
113
114     with tf.name_scope("Fully_Connected_1"):
115         poolshape = net.get_shape().as_list()
116         net = tf.reshape(net, [-1, poolshape[1] * poolshape[2]
117                               * poolshape[3]])
117         net = tf.matmul(net, W["w6"])
118         net = tf.clip_by_value(net, -0.3, 0.3)
119         net = tf.nn.relu(net)
120         tf.summary.histogram('W6', W["w6"])
121
122     with tf.name_scope("Fully_Connected_3"):
123         net = tf.matmul(net, W["w7"])
124         net = tf.clip_by_value(net, -0.3, 0.3)
125         net = tf.nn.relu(net)
126         tf.summary.histogram('W7', W["w7"])
127
128     with tf.name_scope("Fully_Connected_4"):
129         net = tf.matmul(net, W["w8"])
130         net = tf.clip_by_value(net, -0.8, 0.8)
131         net = tf.nn.relu(net)
132         # print("Output layer: ", net.shape)
133         tf.summary.histogram('W8', W["w8"])
```

```
134
135     with tf.name_scope("SoftmaxPrediction"):
136         logits = net
137         y_pred = tf.nn.softmax(logits=logits)
138         y_pred = tf.identity(y_pred, name="output_pred")
139         y_pred_cls = tf.argmax(y_pred, axis=1)
140         y_pred_cls = tf.identity(y_pred_cls, name="output_cls")
141         tf.summary.histogram('y_pred_cls', y_pred_cls)
142
143     if mode == tf.estimator.ModeKeys.PREDICT:
144         spec = tf.estimator.EstimatorSpec(mode=mode,
145                                           predictions=
146                                               y_pred_cls)
147
148     else:
149         with tf.name_scope("AdamOptimizer"):
150             optimizer = tf.train.AdamOptimizer(learning_rate=
151                 params["learning_rate"])
152
153         with tf.name_scope("CrossEntropy"):
154             cross_entropy = tf.nn.
155                 sparse_softmax_cross_entropy_with_logits(labels=
156                     labels, logits=logits)
157
158         with tf.name_scope("ReduceCrossEntropy"):
159             loss = tf.reduce_mean(cross_entropy)
160
161         tf.summary.scalar('Loss', loss)
```

```
155
156     train_op = optimizer.minimize(
157         loss=loss, global_step=tf.train.get_global_step())
158     metrics = {
159         "accuracy": tf.metrics.accuracy(labels, y_pred_cls)
160     }
161
162     spec = tf.estimator.EstimatorSpec(
163         mode=mode,
164         loss=loss,
165         train_op=train_op,
166         eval_metric_ops=metrics)
167
168     return spec
169
170
171 sess = tf.Session()
172 sess.run(tf.global_variables_initializer())
173
174 model = tf.estimator.Estimator(model_fn=model_fn,
175                                params={"learning_rate": 1e-4},
176                                model_dir="./
                                model_no_bias_v4_csv_dataset/"
                                )
```

```
177 X, y = ic.audio_read_dataset_csv(path_to_train_dataset ,
    path_to_label+'train.csv')
178 input_fn = tf.estimator.inputs.numpy_input_fn({"wav":X}, y=y,
    batch_size=40, shuffle=True, num_epochs=1)
179 X, y = ic.audio_read_dataset_csv(path_to_val_dataset ,
    path_to_label+'val.csv')
180 eval_fn = tf.estimator.inputs.numpy_input_fn({"wav":X}, y=y,
    batch_size=40, shuffle=True, num_epochs=1)
181 X, y = ic.audio_read_dataset_csv(path_to_test_dataset ,
    path_to_label+'test.csv')
182 test_fn = tf.estimator.inputs.numpy_input_fn({"wav":X}, y=y,
    batch_size=40, shuffle=False, num_epochs=1)
183 count = 0
184 while count < 1000:
185     model.train(input_fn=input_fn, steps=1000)
186     result = model.evaluate(input_fn=eval_fn, checkpoint_path=
        None)
187     print(result)
188     print("Classification accuracy: {:.2%}".format(result["
        accuracy"]))
189     sys.stdout.flush()
190     count = count + 1
191
192 model.predict(input_fn=test_fn)
```

Listing I.2: TensorFlow Audio CNN

I.3 C Reference Model and Fixed Point Functions

```
1 #include <stdio.h>
2 #include <stdint.h>
3 #include <math.h>
4
5 #define DEBUG 1
6 #undef ELAB_DEBUG 1
7
8 #define FIXED_POINT_FRACTIONAL_BITS 15
9 #define IP_SIZE      64
10 #define OP_SIZE     62
11 #define DEPTH       3
12 #define WT_SIZE     3
13 #define NUM_FIL     3
14 #define STRIDE      1
15 #define PADDING     0
16 #define NUM_LAYER   0
17 #define NUM_IMG     0
18 #define MP_FLAG     0
19 #define MP_SIZE     31
20 #define CMP         9830
21 #define MAX_MP     31
22
23 // Single image multiple filters
```

```
24 //For multiple images -> prevent the file handle from resetting
    to beginning of file.
25 //Hint: use NUM_IMG
26
27 typedef int16_t fixed_point_t;
28 typedef int32_t fixed_point_o;
29
30 int read_csv_1(char str[], int flag);
31 fixed_point_o c_model(fixed_point_t wt[], fixed_point_t fm[]);
32 fixed_point_t fixed_point_mul(fixed_point_t ip1, fixed_point_t
    ip2, uint8_t fractional_bits);
33 fixed_point_o fixed_point_add(fixed_point_o ip1, fixed_point_o
    ip2);
34 fixed_point_t clipping_fxpt(fixed_point_o ip1, fixed_point_o
    flag);
35 fixed_point_t max_pool(fixed_point_t mp[]);
36 fixed_point_t * layer_c_model(int ip_size, int depth, int
    wt_size, int num_fil, int fil, int stride,
37     int padding, int num_layer, int num_img, int
    mp_flag, int compare);
38     //char wt_path[], char fm_path[]);
39
40 int main()
41 {
42     printf("Hello World\n");
```

```
43
44     char wt_path[] = "../.. / Python/ICNN/ results / weights / fxpt_16
        / weights . csv ";
45     char fm_path[] = "../.. / Python/ICNN/ datasets / csv / demo / test /
        fxpt_16 / test . csv ";
46     int op_size , mp_size , mp_neus;
47     int r , c , k , fil , layer , sample;
48
49     op_size = ((IP_SIZE - WT_SIZE + 2*PADDING) / STRIDE) + 1;
50     mp_size = (int)((op_size - 2) / 2) + 1;
51     mp_neus = mp_size * mp_size;
52
53     int op_fm[mp_size*mp_size*NUM_FIL] , *fm; // layer output
54
55     k=0;
56     for (fil=0; fil < NUM_FIL; fil++)
57     {
58         fm = layer_c_model(IP_SIZE , DEPTH, WT_SIZE, NUM_FIL, fil ,
            STRIDE, PADDING, NUM_LAYER, NUM_IMG, MP_FLAG, CMP); // ,
            wt_path , fm_path);
59         for (r=0; r < mp_neus; r++)
60         {
61             op_fm[r] = *(fm+r);
62             #ifdef DEBUG
63                 printf("mp: %04d; " , op_fm[r]);
```

```
64     #endif
65 }
66 }
67
68     return 0;
69 }
70
71
72 static FILE *fp;
73
74 fixed_point_t * layer_c_model(int ip_size , int depth , int
75     wt_size , int num_fil , int fil , int stride , int padding ,
76     int num_layer , int num_img , int mp_flag , int
77     compare) // , char wt_path[] , char fm_path[])
78 {
79
80     fp = fopen("channell.txt" , "w");
81
82     // Calc variables
83     int op_size , mp_size;
84     int tot_neus , tot_wts , fixed_len , label;
85     char wt_path[] = "weights_0.5.csv";
86     char fm_path[] = "test.csv";
87     // iterators
88     int i , j , k , l , r , c , ch , r_ptr;
```

```
87
88 // calculations
89     op_size    = ((ip_size - wt_size + 2*padding)/stride) + 1;
90     tot_neus   = ip_size*ip_size*depth;
91     tot_wts    = wt_size*wt_size*depth*num_fil;
92     r_ptr      = ip_size;
93     fixed_len  = wt_size*wt_size;
94
95     if(mp_flag)
96     {
97         mp_size = (int)((op_size - 2)/2) + 1;
98     }
99     else
100    {
101        // iterates twice r,c.
102        mp_size = op_size;
103    }
104
105 // Arrays
106 fixed_point_t wt_slice[fixed_len], wt[tot_wts], fm_slice[
    fixed_len], fm[tot_neus];
107 fixed_point_t mp[4];
108 static int mp_fm[MAX_MP*MAX_MP];
109 static fixed_point_o op_fm[OP_SIZE][OP_SIZE], op_fm_cp[
    OP_SIZE][OP_SIZE];
```

```
110
111     // Initializing array
112     for (r=0;r<op_size;r++)
113     {
114         for (c=0;c<op_size;c++)
115         {
116             op_fm_cp[r][c] = 0;
117         }
118     }
119
120     for (r=0;r<MAX_MP*MAX_MP;r++)
121     {
122         mp_fm[r] = 0;
123     }
124     for (j=0; j<tot_neus; j++)
125     {
126         fm[j] = 0;
127     }
128     for (j=0; j<tot_wts; j++)
129     {
130         wt[j] = 0;
131     }
132
133     //READING WEIGHTS
134     wt[0] = read_csv_1(wt_path, num_layer); //stripping label
```

```
135     for (i=0; i<tot_wts; i++)
136     {
137         wt[i] = read_csv_1(wt_path, i+1); //+1 while stripping
            label
138     }
139
140     // Printing weights
141     #ifdef DEBUG
142     fprintf(fp, "WEIGHTS FILTER 1\n");
143     l=0;
144     for (i=0; i<num_fil; i++)
145     {
146         for (j=0; j<depth; j++)
147         {
148             for (k=0; k<fixed_len; k++)
149             {
150                 fprintf(fp, "wt[%02d]:%05d", k, wt[l]);
151                 l++;
152             }
153             fprintf(fp, "\n");
154         }
155         fprintf(fp, "\n");
156     }
157     fprintf(fp, "\n");
158     #endif
```

```
159
160 //READING FEATURE MAP
161
162 label = read_csv_1(fm_path, num_img); //stripping the
    label
163 //num_img as file_handle to prevent
    resetting the pointer and read same
    image repetitively.
164 for(j=0; j<tot_neus; j++)
165 {
166     fm[j] = read_csv_1(fm_path, j+1);
167 }
168
169
170 // Printing FM
171 #ifdef DEBUG
172 fprintf(fp, "\n\n");
173 fprintf(fp, "Feature Map\n");
174 k=0;
175 for(i=0; i<depth; i++)
176 {
177     fprintf(fp, "CH: %d\n", i);
178     for(j=0; j<ip_size; j++)
179     {
180         for(l=0; l<ip_size; l++)
```



```
181     {
182         fprintf(fp, "%05d; ", fm[k]);
183         k++;
184     }
185     fprintf(fp, "\n");
186 }
187 fprintf(fp, "\n");
188 }
189 fprintf(fp, "\n");
190 #endif
191
192 // Slicing Feature Map
193 for(ch=0;ch<depth;ch++)
194 {
195     #ifdef DEBUG
196         fprintf(fp, "CH: %d\n", ch);
197     #endif
198     // Slice weights every new channel
199     for(i=0;i<fixed_len;i++)
200     {
201         wt_slice[i] = wt[i+(ch*wt_size*wt_size)+(fil*fixed_len*
                depth)];
202         #ifdef DEBUG
203             fprintf(fp, "wt_slice[%02d]:%05d ", i, wt_slice[i]);
204         #endif
```

```
205     }
206     #ifdef DEBUG
207     fprintf(fp, "\n");
208     #endif
209
210     // Slice feature map row wise
211     for(r=0; r<op_size; r++)
212     {
213         for(i=0; i<3; i++)
214         {
215             fm_slice[i] = fm[i+r+(ch*ip_size*ip_size)];
216             fm_slice[i+3] = fm[i+r_ptr+r+(ch*ip_size*ip_size)];
217             fm_slice[i+6] = fm[i+(2*r_ptr)+r+(ch*ip_size*ip_size)];
218         }
219         // Printing Row Slice
220         #ifdef DEBUG
221         k=0;
222         for(i=0; i<3; i++)
223         {
224             for(j=0; j<3; j++)
225             {
226                 fprintf(fp, "fm_slice[%02d]:%04d; ", k, fm_slice[k]);
227                 k++;
228             }
229             fprintf(fp, "\n");
```

```
230     }
231     fprintf(fp, "\n");
232     #endif
233
234     op_fm[0][r] = c_model(wt_slice, fm_slice);
235     #ifdef DEBUG
236     fprintf(fp, "op_fm[0][%d]: %d;\n", r, op_fm[0][r]);
237     #endif
238
239     for(c=1; c<op_size; c++)
240     {
241         // shift elements
242         for(i=0; i<6; i++)
243         {
244             fm_slice[i] = fm_slice[i+wt_size];
245         }
246         fm_slice[6] = 0;
247         fm_slice[7] = 0;
248         fm_slice[8] = 0;
249
250         // striding column wise
251         for(i=0; i<3; i++)
252         {
253             //c+2 cuz the first two rows of this col slice are
                shift elements
```

```
254         fm_slice [ i+6] = fm[ i+((c+2)*(r_ptr))+r+(ch*ip_size*
                ip_size )];
255     }
256
257     // Printing Col Slice
258     #ifdef DEBUG
259     k=0;
260     for (i=0; i<3; i++)
261     {
262         for (j=0; j<3; j++)
263         {
264             fprintf (fp , "fm_slice[%d]:%d; ", k, fm_slice [k]);
265             k++;
266         }
267         fprintf (fp , "\n");
268     }
269     fprintf (fp , "\n");
270     #endif
271
272     op_fm [c][r] = c_model (wt_slice , fm_slice);
273     #ifdef DEBUG
274     fprintf (fp , "op_fm[%d][%d]: %d;\n" , c , r , op_fm [c][r]);
275     #endif
276 }
277
```

```
278     #ifdef DEBUG
279         fprintf(fp, "\n");
280     #endif
281 }
282
283 #ifdef DEBUG
284     fprintf(fp, "\n\n");
285 #endif
286
287 // Accumulate channel data
288 #ifdef DEBUG
289     fprintf(fp, "\nAccumulating OUTPUT FM's '\n");
290 #endif
291     for (r=0; r<op_size; r++)
292     {
293         for (c=0; c<op_size; c++)
294         {
295             // op_fm_cp[r][c] = op_fm_cp[r][c] + op_fm[r][c];
296             op_fm_cp[r][c] = fixed_point_add(op_fm_cp[r][c], op_fm[
                r][c]);
297             #ifdef DEBUG
298                 // printf("op_fm_cp[%d][%d]: %d; ", r, c, op_fm_cp[r][c
                ]);
299             fprintf(fp, "%d; ", op_fm_cp[r][c]);
300             #endif
```

```
301     }
302     #ifdef DEBUG
303     fprintf(fp, "\n");
304     #endif
305 }
306 }
307 #ifdef DEBUG
308 fprintf(fp, "\n\n");
309 #endif
310
311 // Print output FM
312 #ifdef DEBUG
313 fprintf(fp, "\nRELU OUTPUT FM\n");
314 #endif
315 for (r=0; r<op_size; r++)
316 {
317     for (c=0; c<op_size; c++)
318     {
319         //RELU
320         op_fm_cp[r][c] = clipping_fxpt(op_fm_cp[r][c], compare);
321         #ifdef DEBUG
322         // printf("op_fm[%d][%d]: %d; ", r, c, op_fm_cp[r][c]);
323         fprintf(fp, "%d; ", op_fm_cp[r][c]);
324         #endif
325     }
```

```
326     #ifdef DEBUG
327         fprintf(fp, "\n");
328     #endif
329 }
330
331 #ifdef DEBUG
332     fprintf(fp, "\n\n");
333 #endif
334
335 //MAX-POOL
336 if (mp_flag)
337 {
338     k=0;
339     #ifdef DEBUG
340         fprintf(fp, "Max-Pool\n");
341     #endif
342     for (r=0; r<mp_size; r++)
343     {
344         for (c=0; c<mp_size; c++)
345         {
346             mp[0] = op_fm_cp[r+(r*1)][c+(c*1)];
347             mp[1] = op_fm_cp[r+(r*1)][c+(c*1)+1];
348             mp[2] = op_fm_cp[r+(r*1)+1][c+(c*1)];
349             mp[3] = op_fm_cp[r+(r*1)+1][c+(c*1)+1];
350             mp_fm[k] = max_pool(mp);
```

```
351     #ifdef DEBUG
352     fprintf(fp, "mp_fm[%d]:%06d; ", k, mp_fm[k]);
353     #endif
354     k++;
355 }
356 }
357 #ifdef DEBUG
358 fprintf(fp, "\n\n");
359 #endif
360 }
361 else
362 {
363     k=0;
364     #ifdef DEBUG
365     printf("Max-Pool\n");
366     #endif
367     for(r=0;r<mp_size;r++)
368     {
369         for(c=0;c<mp_size;c++)
370         {
371             //simply converting to single dimensional array.
372             mp_fm[k] = op_fm[r][c];
373             #ifdef DEBUG
374             fprintf(fp, "mp_fm[%d]:%d; \n", k, mp_fm[k]);
375             #endif
```



```
376         k++;
377     }
378 }
379 #ifdef DEBUG
380     fprintf(fp, "\n");
381 #endif
382 }
383
384 fclose(fp);
385 return mp_fm;
386 }
387
388
389 int read_csv_1(char str[], int flag)
390 {
391     char char_read;
392     char str_read[15];
393     int val1;
394     int len =0;
395     int i, j;
396     static FILE* stream;
397
398     if (flag==0)
399     {
400         stream = fopen(str, "r");
```

```
401     // printf("stream: %d\n", stream);
402 }
403
404 char_read = getc(stream);
405 while((char_read != ',') & (char_read != '\n') & (char_read
    != EOF))
406 {
407     str_read[len] = char_read;
408     len++;
409     char_read = getc(stream);
410 }
411
412 len = 0;
413 val1 = atoi(str_read);
414
415 for(j = 0; j < 15; j++)
416     str_read[j] = 0;
417
418 return val1;
419 }
420
421
422 fixed_point_o c_model(fixed_point_t wt[], fixed_point_t fm[])
423 {
424
```

```
425     int wt_i, fm_i, i;
426     fixed_point_t dot_prod;
427     fixed_point_o accum;
428     wt_i = sizeof(wt);
429     fm_i = sizeof(fm);
430     if(wt_i == fm_i)
431     {
432         for(i=0;i<9;i++)
433         {
434             dot_prod = fixed_point_mul(wt[i], fm[i],
435                                     FIXED_POINT_FRACTIONAL_BITS);
436             fprintf(fp, "wt[%d]: %d * fm[%d]: %d, dot_prod = %d\n", i,
437                   wt[i], i, fm[i], dot_prod);
438             accum = fixed_point_add(accum, dot_prod);
439             fprintf(fp, "accum: %d;\n", accum);
440         }
441     }
442
443
444     fixed_point_t float_to_fixed(float input, uint8_t
445                                fractional_bits)
446     {
447         //Input float value is multiplied by 7FFF => (2^15 - 1)
```

```
447  if(input < 0)
448  {
449      fixed_point_t val;
450      val = (fixed_point_t) (input *(-1)* ((1 << fractional_bits)
451                               -1));
452      return ~(val) + 1;
453  }
454  else
455  {
456      return (fixed_point_t) (input * ((1 << fractional_bits)-1))
457          ;
458  }
459
460
461 double fixed16_to_double(fixed_point_t input, uint8_t
462                          fractional_bits)
463 {
464     //Input Fixed point value is divided by 7FFF => (2^15 - 1)
465     //check if input is negative
466     if (0x8000 && input)
467         //convert to 2's complement
```

```
468     double val;
469     val = ~(input - 1) * (-1);
470     return ((double)val / (double)((1 << fractional_bits)-1));
471 }
472 else
473 {
474     //return as it is if input is positive.
475     return ((double)input / (double)((1 << fractional_bits)-1))
        ;
476 }
477 }
478
479 double fixed32_to_double(fixed_point_o input, uint8_t
    fractional_bits)
480 {
481     //Input Fixed point value is divided by 7FFF => (2^15 - 1)
482     //check if input is negative
483     if (input < 0)
484     {
485         //convert to 2's complement
486         double val;
487         val = ~(input - 1) * (-1);
488         return ((double)val / (double)((1 << fractional_bits)-1));
489     }
490     else
```

```
491  {
492      //return as it is if input is positive.
493      return ((double)input / (double)((1 << fractional_bits)-1))
         ;
494  }
495 }
496
497
498 fixed_point_t fixed_point_mul(fixed_point_t ip1, fixed_point_t
         ip2, uint8_t fractional_bits)
499 {
500     fixed_point_o op1;
501     fixed_point_t op2;
502     op1 = ((fixed_point_t)ip1 * (fixed_point_t)ip2);
503     op2 = op1>>fractional_bits;
504     return op2;
505 }
506
507
508 fixed_point_o fixed_point_add(fixed_point_o ip1, fixed_point_o
         ip2)
509 {
510     fixed_point_o op1, temp;
511     fixed_point_t op2, warning=0;
512     op1 = ip1 + ip2;
```

```
513
514     return op1;
515 }
516
517 fixed_point_t clipping_fxpt(fixed_point_o ip1, fixed_point_o
    flag)
518 {
519     fixed_point_t op1;
520
521     if(ip1 > flag)
522     {
523         op1 = flag;
524     }
525     else if(ip1 < 0)
526     {
527         op1 = 0;
528     }
529     else
530     {
531         op1 = (fixed_point_t) ip1;
532     }
533
534     return op1;
535 }
536
```

```
537
538 fixed_point_t max_pool(fixed_point_t mp[])
539 {
540     fixed_point_t max;
541     int i;
542
543     for (i=0; i < 4; i++)
544     {
545         if (mp[i] > max)
546         {
547             max = mp[i];
548         }
549     }
550     return max;
551 }
```

Listing I.3: C Reference Model and Fixed Point Functions

I.4 Shared Object file Functions

```
1 import ctypes
2 import os
3 # from ctypes import *
4 from numpy.ctypeslib import ndpointer
5
6 # _fun = ctypes.CDLL("fixed_point.so")
7 _fun = ctypes.CDLL(".././.././.././Lib/fixed_point.so")#os.path.
    join(os.getcwd(), "fixed_point.so"))
8 # _fun = ctypes.CDLL("C:/Users/RJ/RIT/rjg7712_grad_thesis/
    Python/Lib/fixed_point.so")
9 # libc = cdll.msvcrt
10 # printf = libc.printf
11
12 _fun.float_to_fixed.argtypes = (ctypes.c_float, ctypes.c_uint8)
13 _fun.float_to_fixed.restype = (ctypes.c_int16)
14
15 _fun.fixed16_to_double.argtypes = (ctypes.c_int16, ctypes.
    c_uint8)
16 _fun.fixed16_to_double.restype = (ctypes.c_double)
17
18 _fun.fixed32_to_double.argtypes = (ctypes.c_int32, ctypes.
    c_uint8)
19 _fun.fixed32_to_double.restype = (ctypes.c_double)
```

```
20
21 _fun.fixed_point_mul.argtypes = (ctypes.c_int16, ctypes.c_int16
    , ctypes.c_uint8)
22 _fun.fixed_point_mul.restype = (ctypes.c_int16)
23
24 _fun.fixed_point_add.argtypes = (ctypes.c_int32, ctypes.c_int32
    )
25 _fun.fixed_point_add.restype = (ctypes.c_int32)
26
27 _fun.clipping_fxpt.argtypes = (ctypes.c_int32, ctypes.c_int32)
28 _fun.clipping_fxpt.restype = (ctypes.c_int16)
29
30 def float_to_fixed(input, q_format):
31     global _fun
32     result = _fun.float_to_fixed(input, q_format)
33     return result
34
35
36 def fixed16_to_double(input, q_format):
37     global _fun
38     result = _fun.fixed16_to_double(input, q_format)
39     return result
40
41 def fixed32_to_double(input, q_format):
42     global _fun
```

```
43     result = _fun.fixed32_to_double(input , q_format)
44     return result
45
46 def fixed_point_mul(input1 , input2 , input3):
47     global _fun
48     result = _fun.fixed_point_mul(input1 , input2 , input3)
49     # printf("result from libfun: %d", result)
50     return result
51
52 def fixed_point_add(input1 , input2):
53     global _fun
54     result = _fun.fixed_point_add(input1 , input2)
55     return result
56
57 def clipping_fxpt(input1 , input2):
58     global _fun
59     result = _fun.clipping_fxpt(input1 , input2)
60     return result
```

Listing I.4: Shared Object file Functions

I.5 NumPy Functions

```
1 # This file contains all functions for
2 # Floating point format
3 # Fixed Point Format
4 # Fixed Point to Floating Point Format
5
6 import numpy as np
7 import matplotlib.pyplot as plt
8 import os
9 import sys
10 import cv2
11 import libfun
12 import glob
13
14 """=====
15     """
16 """ Step 1: Load images from dataset and resize if necessary.
17     """
18 """=====
19
20 def load_data(load_dir , num_samples):
21     train_datasets = os.listdir(load_dir)
```

```
21     num_class = len(train_datasets)
22     # print("number of classes: ", numClass)
23     i = 0
24     img_data = np.empty((num_samples, 64, 64, 3), dtype="
        float32")
25     labels = np.empty(num_samples)
26     for dataset in train_datasets:
27         image_list = os.listdir(load_dir + dataset + '/')
28         for img in image_list:
29             img_ip = cv2.imread(load_dir + dataset + '/' + img)
30             img_resize = cv2.resize(img_ip, (64, 64),
                interpolation=cv2.INTER_CUBIC)
31             img_reshape = cv2.cvtColor(img_resize, cv2.
                COLOR_BGR2RGB)
32             img_data[i, :, :, :] = img_reshape
33             if dataset == "cars":
34                 labels[i] = np.uint16(0)
35             elif dataset == "lanes":
36                 labels[i] = np.uint16(1)
37             elif dataset == "pedestrians":
38                 labels[i] = np.uint16(2)
39             elif dataset == "lights":
40                 labels[i] = np.uint16(3)
41             i = i + 1
42             plt.show()
```

```
43     return img_data, num_class, labels
44
45
46 def load_image(directory = "dataset/"):
47     address = glob.glob(directory+'**/*')
48     # labels = [0 if 'cars' in add else 1 if 'lanes' in add
49                 else 2 for add in address]
49     num_samples = len(address)
50     (X_train, num_classes, labels) = load_data(directory,
51         num_samples)
51     # get Number of images in dataset
52     num_samples = X_train.shape[0]
53     # print("number of samples: ", num_samples)
54     return X_train, labels, address
55
56
57 def convert_img_np(image_tf_glob, scale, fractional_bits, flag)
58     :
59     # print("=====")
60     # print("Converting tensor-flow image to numpy format")
61     img_org = np.zeros([image_tf_glob.shape[0], image_tf_glob.
62                         shape[3], image_tf_glob.shape[1], image_tf_glob.shape[2]])
63     for k in np.arange(0, image_tf_glob.shape[0]):
64         for l in np.arange(0, image_tf_glob.shape[1]):
65             for m in np.arange(0, image_tf_glob.shape[2]):
```

```
64         for n in np.arange(0, image_tf_glob.shape[3]):
65             if flag == 0:
66                 # Read values as float32
67                 img_org[k, n, l, m] = np.float32(
68                     image_tf_glob[k, l, m, n] / (255*
69                         scale))
70             elif flag == 1:
71                 # Convert float32 to fixed
72                 img_org[k, n, l, m] = np.int16(libfun.
73                     float_to_fixed(image_tf_glob[k, l, m,
74                         n] / (255*scale), fractional_bits))
75             elif flag == 2:
76                 # Convert float32 to 16bit fixed point
77                 x_value = np.int16(libfun.
78                     float_to_fixed(image_tf_glob[k, l, m,
79                         n] / (255*scale), fractional_bits))
80                 # Convert fixed point 16 to float 16
81                 img_org[k, n, l, m] = np.float16(libfun
82                     .fixed16_to_double(x_value,
83                         fractional_bits))
84             # print("Converted to numpy format: \n image_org: \n",
85                 img_org, "\n image_org shape: ", img_org.shape)
86         return img_org
```

```
80 def convert_fm_np_to_tf(img):
81
82     image_tf_glob = np.zeros([1, img.shape[1], img.shape[2],
83                               img.shape[0]])
84
85     # print("initializing matrix for image_tf: ", image_tf_glob
86           .shape)
87
88     for k in np.arange(0, image_tf_glob.shape[0]):
89         for l in np.arange(0, image_tf_glob.shape[1]):
90             for m in np.arange(0, image_tf_glob.shape[2]):
91                 for n in np.arange(0, image_tf_glob.shape[3]):
92                     image_tf_glob[k, l, m, n] = img[n, l, m]
93
94     # print("Converted to tensor-flow format: \n image_tf shape
95           : ", image_tf_glob.shape)
96
97     # print("numpy array converted to tensor test: \n image_tf:
98           \n", image_tf)
99
100    return image_tf_glob
101
102    """=====
103    """
104
105    """ Step 2: Apply convolution on the image. """
106
107    """=====
108    """
109
110    # only one image is processed at a time inside the function,
```



```
99 # hence check for number of channels in image
100 # convolution for nxn image only.
101 # TODO: include padding
102 """=====
    """
103
104
105 def convolve(img, filter , f, result_size_n , result_size_m , p, s
    , fractional_bits , flag):
106     # p-padding and s-stride
107     # print("Cache size",resultSize)
108     cache = np.zeros((result_size_n + 1, result_size_m + 1))
109     i = 0
110     j = 0
111     # Convolution over different regions of image with filter.
112     for r in np.uint16(np.arange(f / 2, img.shape[0] - f / 2 +
        1, s)):
113         # pointers to store results in resultant matrix
114         i = i + 1
115         j = 0
116         for c in np.uint16(np.arange(f / 2, img.shape[1] - f /
            2 + 1, s)):
117             j = j + 1
118             # Get current region to convolve
```

```
119         # f/2 to get the current center column that needs
           to be
120         curr_reg = img[r - np.uint16(np.floor(f / 2)):r +
           np.uint16(np.ceil(f / 2)),
121                 c - np.uint16(np.floor(f / 2)):c + np.
           uint16(np.ceil(f / 2))]
122         # convolution
123         # Float32
124         if flag == 0:
125             conv = np.zeros([3, 3])
126             conv = curr_reg * filter # dot product
127             result = np.sum(conv) # sum of product
           terms
128             cache[i, j] = result # result of
           convolution
129         if flag == 1:
130             conv = np.int16(np.zeros([3, 3]))
131             # dot product
132             for l in range(0, curr_reg.shape[0]):
133                 for m in range(0, curr_reg.shape[1]):
134                     conv[l, m] = np.int16(libfun.
           fixed_point_mul(curr_reg[l, m],
           filter[l, m], fractional_bits)) #
           dot product
135             # sum of product terms
```

```
136         result = 0
137         # result_temp = 0
138         # result_org = 0
139         for p in range(0, conv.shape[0]):
140             for q in range(0, conv.shape[1]):
141                 result = np.int32(libfun.
142                     fixed_point_add(result, conv[p, q]))
143                 # result of convolution
144                 cache[i, j] = np.int32(result)
145             if flag == 2:
146                 conv = np.zeros([3, 3])
147                 conv = np.float16(curr_reg * filter) # dot
148                 # product
149                 result = np.float16(np.sum(conv)) # sum of
150                 # product terms
151                 cache[i, j] = np.float16(result) # result
152                 # of convolution
153
154 # added +1 to resultSize because Python omits last digit (
155     limit).
156 # stripping off zeros to get correct size output.
157 if flag == 0:
158     cache = np.float32(cache[1:result_size_n + 1, 1:
159         result_size_m + 1])
160 return cache
```

```
155     elif flag == 1:
156         cache = np.int32(cache[1:result_size_n + 1, 1:
157                             result_size_m + 1])
157         return cache
158     elif flag == 2:
159         cache = np.float16(cache[1:result_size_n + 1, 1:
160                                 result_size_m + 1])
160         return cache
161
162
163 """=====
164 """
164 """    Check for all conditions and pass inputs to "Convolve"
165 """
165 """=====
166 """
166
167
168 def conv(img, filt, p, s, fractional_bits, flag):
169     f = filt.shape[-1] # Filter size
170     n = img.shape[-2] # Image size
171     m = img.shape[-1]
172     result_size_n = np.uint16((n + 2 * p - f) / s + 1)
173     result_size_m = np.uint16((m + 2 * p - f) / s + 1)
174
```

```
175     # filter.shape: (2,3,3) for 2d image, (2,3,3,3) for rgb
        image
176     if len(img.shape) > 2 or len(filt.shape) > 3:
177         # check if number of ch's in filter and image match.
178         if img.shape[0] != filt.shape[1]:
179             print("img shape: ", img.shape, "filter shape: ",
                    filt.shape)
180             print("Error: Number of channels in both image and
                    filter must match.")
181             sys.exit()
182
183     # The filter has to be a square matrix
184     if len(filt.shape) > 3:
185         if filt.shape[-1] != filt.shape[-2]:
186             print("Filter shape: ", filt.shape)
187             print("Error: filter should be a square matrix")
188             sys.exit()
189     else:
190         if filt.shape[-1] != filt.shape[-2]:
191             print("Filter shape: ", filt.shape)
192             print("Error: filter should be a square matrix")
193             sys.exit()
194
195     # The filter should have odd dimensions
196     # if filter.shape[-1]%2 ==0:
```

```
197     # print("Error: Filter dimensions should be odd")
198     # sys.exit()
199
200     if len(filt.shape) > 2:
201         feature_maps = np.zeros((filt.shape[0], result_size_n,
202                                 result_size_m))
203
204         for nFilter in range(filt.shape[0]):
205             # print("Filter ", nFilter+1)
206             curr_fil = filt[nFilter, :]
207             if len(curr_fil.shape) > 2:
208                 if flag == 1:
209                     conv_map = convolve(img[0, :, :], curr_fil
210                                         [0, :, :],
211                                         f, result_size_n,
212                                         result_size_m, p, s,
213                                         fractional_bits, flag)
214                 else:
215                     conv_map = convolve(img[0, :, :], curr_fil
216                                         [0, :, :],
217                                         f, result_size_n,
218                                         result_size_m, p, s,
219                                         fractional_bits, flag)
220             # print("Filter ch0: \n", currFil[0, :, :])
221             for chNum in range(1, curr_fil.shape[0]):
```

```
219         if flag == 0:
220             conv_map = conv_map + convolve(img[
221                 chNum, :, :],
222                 curr_fil
223                 [chNum
224                 , :,
225                 :],
226                 f,
227                 result_size_n
228                 ,
229                 result_size_m
230                 , p, s
231                 ,
232                 fractional_bits
233                 , flag
234                 )
235         elif flag == 1:
236             conv_map_1 = convolve(img[chNum, :, :],
237                 curr_fil[chNum,
238                 :, :],
239                 f, result_size_n ,
240                 result_size_m , p,
241                 s,
242                 fractional_bits ,
243                 flag)
```

```
231
232         for i in range(0, conv_map_1.shape[0]):
233             for j in range(0, conv_map_1.shape
234                           [1]):
235                 conv_map[i, j] = np.int32(libfun
236                                           .fixed_point_add(conv_map[i, j]
237                                                           ], conv_map_1[i, j]))
238
239     elif flag == 2:
240         conv_map = convolve(img[0, :, :],
241                             curr_fil[0, :, :],
242                             f, result_size_n,
243                             result_size_m, p, s
244                             ,
245                             fractional_bits,
246                             flag)
247
248     conv_map = np.float16(conv_map +
249                           convolve(img[chNum, :, :],
250                                     curr_fil
251                                     [
252                                         chNum
253                                         ,
254                                         :,
255                                         :
```



```
248             # print("Filter ch", chNum, ": \n",
                currFil[chNum, :, :])
249             # print("convMap: \n", convMap)
250         else:
251             if flag == 1:
252                 conv_map = convolve(img, curr_fil,
253                                     f, result_size_n,
254                                     result_size_m, p, s,
255                                     fractional_bits, flag)
256             else:
257                 conv_map = convolve(img, curr_fil,
258                                     f, result_size_n,
259                                     result_size_m, p, s,
260                                     fractional_bits, flag)
261                 feature_maps[nFilter, :, :] = conv_map
262             if flag == 1:
263                 feature_maps[nFilter, :, :] = conv_map
264             else:
265                 feature_maps[nFilter, :, :] = conv_map
266             # feature_maps[nFilter, :, :] = conv_map
267         else:
268             if flag == 1:
269                 feature_maps = np.zeros((result_size_n,
                result_size_m))
```

```
270         conv_map = convolve(img, filt, f, result_size_n,
271                             result_size_m, p, s, fractional_bits, flag)
272     feature_maps = conv_map
273 else:
274     feature_maps = np.zeros((result_size_n,
275                             result_size_m))
276     conv_map = convolve(img, filt, f, result_size_n,
277                         result_size_m, p, s, fractional_bits, flag)
278     feature_maps = conv_map
279 # print("ConvFeature map: ", featureMaps.shape)
280 if flag == 1:
281     return np.int32(feature_maps)
282 else:
283     return feature_maps
284
285 """=====
286 """
287
288 """ Step 3: Apply ReLu activation function feature maps.
289 """
290
291 """ Rectified Linear Unit """
292 """=====
293 """
```

```
289 def bias_add(feature_maps, feature_maps_1, bias, bias_1, flag):
290     conv_bias = np.int16(np.zeros(feature_maps.shape))
291     conv_bias_temp = np.zeros(feature_maps.shape)
292     conv_bias_org = np.zeros(feature_maps.shape)
293     if feature_maps.shape[0] == bias.shape[0]:
294         # print("number of fm's is equal to bias values")
295         i = 0
296         if flag == 0 or flag == 2:
297             for i in np.arange(0, feature_maps.shape[0]):
298                 conv_bias[i, :, :] = feature_maps[i, :, :] +
                bias[i]
299         elif flag == 1:
300             for i in np.arange(0, feature_maps.shape[0]):
301                 feature_maps_temp = feature_maps[i, :, :]
302                 feature_maps_temp_1 = feature_maps_1[i, :, :]
303                 for k in range(0, feature_maps_temp.shape[0]):
304                     for l in range(0, feature_maps_temp.shape
                        [1]):
305                         conv_bias[i, k, l] = np.int16(libfun.
                            fixed_point_add(feature_maps_temp[k, l
                                ], bias[i]))
306                 conv_bias_temp[i, k, l] = np.float16(
                    libfun.fixed16_to_double(conv_bias[i,
                        k, l], 15))
```

```
307             conv_bias_org[i, k, l] = np.float16(
                feature_maps_temp_1[k, l] + bias_1[i
                ])
308     if flag == 1:
309         return np.int16(conv_bias), np.float16(conv_bias_temp),
                np.float16(conv_bias_org)
310     else:
311         return conv_bias
312
313
314 """=====
    """
315 """    Step 3:Apply ReLu activation function feature maps.
    """
316 """          Rectified Linear Unit          """
317 """=====
    """
318
319
320 # ReLu function only gives a positive number and outputs a zero
321 # every time a negative value is compared with zero
322
323
324 def relu_non_linearity(feature_maps):
```

```
325     if len(feature_maps.shape) == 2 and feature_maps.shape[0]
        == 1:
326         # preparing the output matrix for ReLu activation
            function
327         relu_out = np.zeros(feature_maps.shape)
328         for val in np.arange(0, feature_maps.shape[1]):
329             relu_out[0, val] = np.max([feature_maps[0, val],
                0])
330
331     else:
332         # preparing the output matrix for ReLu activation
            function
333         relu_out = np.zeros(feature_maps.shape)
334         # preparing for partial derivative of RELU, for back-
            propagation
335         pd_relu_out = np.zeros(feature_maps.shape)
336         for map_num in range(feature_maps.shape[0]):
337             for r in np.arange(0, feature_maps.shape[1]):
338                 for c in np.arange(0, feature_maps.shape[2]):
339                     relu_out[map_num, r, c] = np.max([
                        feature_maps[map_num, r, c], 0])
340                     # print("mapnum: ", mapNum, "element:",
                        reluOut[mapNum, r, c],
341                     # "map element: ", featureMaps[mapNum, r, c]
                        )
```

```
342         if np.max([feature_maps[map_num, r, c], 0])
           != 0:
343             pd_relu_out[map_num, r, c] = 1
344             # print("ReluOut: \n",pd_reluOut)
345     return relu_out
346
347
348 """=====
   """
349 """ Step 4: Apply Max Pooling on the feature map from ReLu.
   """
350 """=====
   """
351
352
353 def max_pooling(feature_map, f_size=2, stride=2):
354     # preparing output for pooling operation
355     if np.uint16(feature_map.shape[1] % 2 == 0):
356         max_pool_out = np.zeros((np.uint16((feature_map.shape
           [0])),
357                                 np.uint16((feature_map.shape
           [1] - f_size + 1) / stride +
           1),
358                                 np.uint16((feature_map.shape
           [-1] - f_size + 1) / stride
```

```

+ 1)))
359     else :
360         max_pool_out = np.zeros((np.uint16((feature_map.shape
           [0])),
361                                 np.uint16((feature_map.shape
           [1] - f_size + 1) / stride),
362                                 np.uint16((feature_map.shape
           [-1] - f_size + 1) / stride)
           ))
363     # partial derivative of pooling layer for back prop.
364     pd_max_pool_out = np.zeros(feature_map.shape)
365
366     for map_num in range(feature_map.shape[0]):
367         r2 = 0
368         for r in np.arange(0, feature_map.shape[1] - f_size +
           1, stride):
369             c2 = 0
370             for c in np.arange(0, feature_map.shape[-1] -
           f_size + 1, stride):
371                 max_pool_out[map_num, r2, c2] = np.max([
           feature_map[map_num, r:r + f_size, c:c +
           f_size]])
372                 # argmax get the indices of the max value in a
           matrix
```



```
373         i, j = np.unravel_index(np.argmax([feature_map[
           map_num, r:r + f_size, c:c + f_size]]), (
           f_size, f_size))
374         pd_max_pool_out[map_num, r + i, c + j] = 1
375         c2 = c2 + 1
376         r2 = r2 + 1
377     # print("Max Pooling: \n", "maxPoolOut: ", max_pool_out.
           shape)
378     return max_pool_out
379
380
381 """=====
           """
382 """      Implementing Fully Connected layer      """
383 """=====
           """
384
385
386 def fc_bias_add(in_mat, in_mat_org, bias, bias_org, flag):
387     if flag == 0 or flag == 2:
388         d_out = np.add(in_mat, bias)
389     elif flag == 1:
390         d_out = np.int16(np.zeros(in_mat.shape))
391         d_out_temp = np.float16(np.zeros(in_mat.shape))
392         d_out_org = np.float16(np.zeros(in_mat.shape))
```

```
393     if in_mat.shape[1] == bias.shape[0]:
394         for r in range(0, in_mat.shape[1]):
395             d_out[0, r] = np.int16(libfun.fixed_point_add(
396                 in_mat[0, r], bias[r]))
396             d_out_temp[0, r] = np.float16(libfun.
397                 fixed16_to_double(d_out[0, r], 15))
397             d_out_org[0, r] = np.float16(in_mat_org[0, r] +
398                 bias_org[r])
398     if flag == 1:
399         return np.int16(d_out), np.float16(d_out_temp), np.
400             float16(d_out_org)
400     else:
401         return d_out
402
403
404 def matrix_mul(in_mat, w_mat, fractional_bits, flag):
405     if flag == 1:
406         result = np.int32(np.zeros([in_mat.shape[0], w_mat.
407             shape[1]]))
407     else:
408         result = np.int16(np.zeros([in_mat.shape[0], w_mat.
409             shape[1]]))
409     if in_mat.shape[1] == w_mat.shape[0]:
410         if flag == 0 or flag == 2:
411             result = np.matmul(in_mat, w_mat)
```

```
412         elif flag == 1:
413             for col in range(0, w_mat.shape[1]):
414                 for row in range(0, w_mat.shape[0]):
415                     input_2 = libfun.fixed_point_mul(in_mat[0,
416                                                         row], w_mat[row, col], fractional_bits)
417                     result[0, col] = np.int32(libfun.
418                                                         fixed_point_add(result[0, col], input_2))
419         else:
420             print("Shapes Mismatch: in_mat: ", in_mat.shape, "w_mat
421                   : ", w_mat.shape)
422             sys.exit()
423         if flag == 1:
424             return np.int32(result)
425         else:
426             return result
427
428 def clipping_fxpt(in_mat, flag):
429     if len(in_mat.shape) > 2:
430         for i in range(in_mat.shape[0]):
431             for j in range(in_mat.shape[1]):
432                 for k in range(in_mat.shape[2]):
433                     in_mat[i, j, k] = libfun.clipping_fxpt(
434                                             in_mat[i, j, k], flag)
435     return in_mat
```

```
433     elif len(in_mat.shape) == 2:
434         for i in range(in_mat.shape[0]):
435             for j in range(in_mat.shape[1]):
436                 in_mat[i, j] = libfun.clipping_fxpt(in_mat[i, j
437                 ], flag)
438     return in_mat
439 else:
440     print("Wrong input to clip")
441     sys.exit()
442
443 def clipping_flt(in_mat, flag):
444     if len(in_mat.shape) > 2:
445         for i in range(in_mat.shape[0]):
446             for j in range(in_mat.shape[1]):
447                 for k in range(in_mat.shape[2]):
448                     if in_mat[i, j, k] > flag:
449                         in_mat[i, j, k] = flag
450                     elif in_mat[i, j, k] < (-1*flag):
451                         in_mat[i, j, k] = (-1*flag)
452     return in_mat
453 elif len(in_mat.shape) == 2:
454     for i in range(in_mat.shape[0]):
455         for j in range(in_mat.shape[1]):
456             if in_mat[i, j] > flag:
457                 in_mat[i, j] = flag
```

```
457             elif in_mat[i, j] < (-1 * flag):
458                 in_mat[i, j] = (-1 * flag)
459         return in_mat
460     else:
461         print("Wrong input to clip")
462         sys.exit()
```

Listing I.5: NumPy Functions

I.6 CNN from Scratch - Python

```
1 # This file contains all functions for
2 # Floating point format
3 # Fixed Point Format
4 # Fixed Point to Floating Point Format
5 # Logic to determine highest value in every output Feature Map
6
7 from __future__ import absolute_import
8 from __future__ import division
9 from __future__ import print_function
10 import numpy as np
11 import csv
12 import sys
13 sys.path.insert(1, '../../../../../Lib')
14 import inspect_checkpoint_csv as ic
15 sys.path.insert(1, '../../../../../Lib')
16 import functions as fn
17 sys.path.insert(1, '../../../../../Lib')
18 import libfun
19
20 path_to_demo_dataset_flt = "../../../../datasets/csv/demo/test/
    flt_32/"
21 path_to_demo_dataset_fxpt = "../../../../datasets/csv/demo/test/
    fxpt_"
```

```
22 path_to_dataset_flt = " ../../../../ datasets / csv / test / flt_32 /"
23 path_to_dataset_fxpt = " ../../../../ datasets / csv / test / fxpt_"
24 path_to_weights_flt = " ../../../../ results / weights / flt_32 /"
25 path_to_weights_fxpt = " ../../../../ results / weights / fxpt_"
26
27 #
=====
28 # -----FIXED POINT FORMAT
-----
29 #
=====

30 # TODO: Modify IC to output
31 # - fixed to float converted values
32 # - values
33
34
35 def model_fn(image_np_0, labels_y, network_weights_0,
              fully_connected_weights_0, fractional_bit, cmp, flag):
36     count = 0
37     compare_x = np.int32(cmp[0])
38     compare_8 = np.int32(cmp[1])
39     samples = image_np_0.shape[0]
40
```

```
41     w1_np_0 = np.int16(network_weights_0[0])
42     w2_np_0 = np.int16(network_weights_0[1])
43     w3_np_0 = np.int16(network_weights_0[2])
44     w4_np_0 = np.int16(network_weights_0[3])
45     w5_np_0 = np.int16(fully_connected_weights_0[0])
46     w6_np_0 = np.int16(fully_connected_weights_0[1])
47     w7_np_0 = np.int16(fully_connected_weights_0[2])
48     w8_np_0 = np.int16(fully_connected_weights_0[3])
49
50
51     for q in range(samples):
52         curr_img_0 = np.int16(image_np_0[q])
53
54         # =====NUMPY LAYER 1=====
55         # Fixed Point Convolution
56         # _fxpt -> Fixed point output, _tmp -> Fixed to float
57         # output, _org -> float16 output
58         fm_lc1_fxpt = fn.conv(curr_img_0, w1_np_0, 0, 1,
59                               fractional_bit, flag)
60         for i in range(fm_lc1_fxpt.shape[0]):
61             for j in range(fm_lc1_fxpt.shape[1]):
62                 for k in range(fm_lc1_fxpt.shape[2]):
63                     fm_lc1_fxpt[i, j, k] = libfun.clipping_fxpt(
64                         fm_lc1_fxpt[i, j, k], compare_x)
65
66         # Fixed Point ReLu
```



```
63     fm_lc1_fxpt = np.int16(fn.relu_non_linearity(
        fm_lc1_fxpt))
64     # Fixed Point Max Pool
65     fm_mp1_fxpt = np.int16(fn.max_pooling(fm_lc1_fxpt))
66
67     # =====NUMPY LAYER 2=====
68     # Fixed Point Convolution
69     fm_lc2_fxpt = fn.conv(fm_mp1_fxpt, w2_np_0, 0, 1,
        fractional_bit, flag)
70     for i in range(fm_lc2_fxpt.shape[0]):
71         for j in range(fm_lc2_fxpt.shape[1]):
72             for k in range(fm_lc2_fxpt.shape[2]):
73                 fm_lc2_fxpt[i, j, k] = libfun.clipping_fxpt
                    (fm_lc2_fxpt[i, j, k], compare_x)
74     # Fixed Point ReLu
75     fm_lc2_fxpt = np.int16(fn.relu_non_linearity(
        fm_lc2_fxpt))
76     # Fixed Point Max Pool
77     fm_mp2_fxpt = np.int16(fn.max_pooling(fm_lc2_fxpt))
78
79     # =====NUMPY LAYER 3=====
80     # Fixed Point Convolution
81     fm_lc3_fxpt = fn.conv(fm_mp2_fxpt, w3_np_0, 0, 1,
        fractional_bit, flag)
82     for i in range(fm_lc3_fxpt.shape[0]):
```

```
83         for j in range(fm_lc3_fxpt.shape[1]):
84             for k in range(fm_lc3_fxpt.shape[2]):
85                 fm_lc3_fxpt[i, j, k] = libfun.clipping_fxpt
                        (fm_lc3_fxpt[i, j, k], compare_x)
86     # Fixed Point ReLu
87     fm_lc3_fxpt = np.int16(fn.relu_non_linearity(
                        fm_lc3_fxpt))
88     # Fixed Point Max Pool
89     fm_mp3_fxpt = np.int16(fn.max_pooling(fm_lc3_fxpt))
90
91     # =====NUMPY LAYER 4=====
92     # Fixed Point Convolution
93     fm_lc4_fxpt = fn.conv(fm_mp3_fxpt, w4_np_0, 0, 1,
                        fractional_bit, flag)
94     for i in range(fm_lc4_fxpt.shape[0]):
95         for j in range(fm_lc4_fxpt.shape[1]):
96             for k in range(fm_lc4_fxpt.shape[2]):
97                 fm_lc4_fxpt[i, j, k] = libfun.clipping_fxpt
                        (fm_lc4_fxpt[i, j, k], compare_x)
98     # Fixed Point ReLu
99     fm_relu4_fxpt = np.int16(fn.relu_non_linearity(
                        fm_lc4_fxpt))
100
101     # =====NUMPY LAYER 5=====
102     # Fixed Point Fully Connected
```

```
103     fm_lc5_fxpt = fn.convert_fm_np_to_tf(fm_relu4_fxpt)
104     fm_lc5_fxpt = np.int16(np.reshape(fm_lc5_fxpt, [1,
        w5_np_0.shape[0]]))
105     # Fully connected.
106     fm_lc5_fxpt = fn.matrix_mul(fm_lc5_fxpt, w5_np_0,
        fractional_bit, flag)
107     for i in range(fm_lc5_fxpt.shape[0]):
108         for j in range(fm_lc5_fxpt.shape[1]):
109             fm_lc5_fxpt[i, j] = libfun.clipping_fxpt(
                fm_lc5_fxpt[i, j], compare_x)
110     # Fixed Point ReLU
111     fm_lc5_fxpt = np.int16(fn.relu_non_linearity(
        fm_lc5_fxpt))
112
113     # =====NUMPY LAYER 6=====
114     # Fully Connected
115     fm_lc6_fxpt = fn.matrix_mul(fm_lc5_fxpt, w6_np_0,
        fractional_bit, flag)
116     for i in range(fm_lc6_fxpt.shape[0]):
117         for j in range(fm_lc6_fxpt.shape[1]):
118             fm_lc6_fxpt[i, j] = libfun.clipping_fxpt(
                fm_lc6_fxpt[i, j], compare_x)
119     # Fixed Point ReLU
120     fm_lc6_fxpt = np.int16(fn.relu_non_linearity(
        fm_lc6_fxpt))
```

```
121
122     # =====NUMPY LAYER 7=====
123     # Fully Connected
124     fm_lc7_fxpt = fn.matrix_mul(fm_lc6_fxpt, w7_np_0,
125                                fractional_bit, flag)
126     for i in range(fm_lc7_fxpt.shape[0]):
127         for j in range(fm_lc7_fxpt.shape[1]):
128             fm_lc7_fxpt[i, j] = libfun.clipping_fxpt(
129                 fm_lc7_fxpt[i, j], compare_x)
130
131     # Fixed Point ReLU
132     fm_lc7_fxpt = np.int16(fn.relu_non_linearity(
133         fm_lc7_fxpt))
134
135     # =====NUMPY LAYER 8=====
136     # Fully Connected
137     fm_lc8_fxpt = fn.matrix_mul(fm_lc7_fxpt, w8_np_0,
138                                fractional_bit, flag)
139     for i in range(fm_lc8_fxpt.shape[0]):
140         for j in range(fm_lc8_fxpt.shape[1]):
141             fm_lc8_fxpt[i, j] = libfun.clipping_fxpt(
142                 fm_lc8_fxpt[i, j], compare_x)
143
144     # Fixed Point ReLU
145     fm_relu8_fxpt = np.int16(fn.relu_non_linearity(
146         fm_lc8_fxpt))
147
```

```
140     # numpy Predictions --> argmax is the way.
141     fm_cmp = np.zeros(fm_relu8_fxpt.shape)
142     val = np.array_equal(fm_cmp, fm_relu8_fxpt)
143     if val:
144         print("No prediction Original label: ", labels_y[q
145             ])
146         print("Output 1: ", fm_relu8_fxpt)
147     else:
148         y_pred_fxpt = np.argmax(fm_relu8_fxpt)
149         name = ""
150         if labels_y[q] == y_pred_fxpt:
151             count = count + 1
152             if y_pred_fxpt == 0:
153                 name = "car"
154             elif y_pred_fxpt == 1:
155                 name = "lane"
156             elif y_pred_fxpt == 2:
157                 name = "pedestrians"
158             elif y_pred_fxpt == 3:
159                 name = "lights"
160             print("At bit width: ", fractional_bit+1, "|
161                 Model says that input image is ", name, "| ",
162                 q, "/", samples)
163         print("Output 1: ", fm_relu8_fxpt)
164         # print("Output 2: ", fm_relu8_flt)
```

```
162         else:
163             print("*****Wrong
                    Prediction*****")
164             print("At address: Original label: ", labels_y[
                    q])
165             print("Output 1: ", fm_relu8_fxpt)
166             # print("Output 2: ", fm_relu8_flt)
167             print("
                    *****
                    ")
168             accuracy = count / (q+1)
169             print("Accuracy: ", accuracy * 100, "%")
170             accuracy = count / image_np_0.shape[0]
171             print("Accuracy: ", accuracy * 100, "%")
172             return accuracy
173
174
175 # bit_widths = [16, 15, 14, 13, 12, 11, 10, 9, 8]
176 bit_widths = [10, 9, 8]
177 # 0.3 and 0.8 range values
178 # cmp = [[9830, 26213],[4914, 13106],[2457, 6552],[1228,
            3276],[614, 1637],[306, 818],[153, 408],[76,204],[38, 101]]
179 # 0.5 and 0.8 range values.
180 cmp = [[255, 408],[127,204],[63, 101]]
181 acc = []
```

```
182 for i in range(len(bit_widths)):
183     image_csv, label_csv = ic.read_dataset_csv(0, bit_widths[i]
184         ]-1, path_to_dataset_fxpt+str(bit_widths[i])+"/test.csv")
185     conv_wts, fc_wts = ic.read_from_checkpoint(
186         path_to_weights_fxpt + str(bit_widths[i])+"/weights_0.5.
187         csv")
188     accuracy = model_fn(image_csv, label_csv, conv_wts, fc_wts,
189         bit_widths[i]-1, cmp[i], 1)
190     acc.append(accuracy)
191     print("=====", bit_widths[i], " bits
192         =====")
193
194 with open(".././././results/accuracy_vs_bitwidth.csv", 'w') as
195     f:
196     wtr = csv.writer(f, lineterminator='\n', delimiter=',')
197     wtr.writerow(bit_widths)
198     wtr.writerow(acc)
```

Listing I.6: CNN from Scratch - Python

I.7 ACCUM

```
1
2 module ACCUM #(parameter BIT_WIDTH=16) (
3     reset ,
4     clk ,
5     scan_in0 ,
6     scan_en ,
7     test_mode ,
8     scan_out0 ,
9     start_bit ,
10    in1 ,
11    out
12    );
13 localparam BIT_WIDTH7P = BIT_WIDTH + 7;
14
15 input
16     reset ,                // system reset
17     clk ;                  // system clock
18
19 input
20     scan_in0 ,             // test scan mode data input
21     scan_en ,              // test scan mode enable
22     test_mode ;           // test mode select
23
```



```
24 input start_bit;
25 input [BIT_WIDTH-1:0] in1;
26
27 //Why +7? because if 0.3 is added 1406 times in 16bit fixed
    point
28 //it results in a value that fits in 16+7 bit-width
29 output reg [BIT_WIDTH7P:0] out;
30
31 output
32     scan_out0;                // test scan mode data output
33
34 always @(posedge clk)
35 begin
36     if(reset)
37         out <= 0;
38     else
39         if (start_bit) begin
40             out <= out + {{8{in1[BIT_WIDTH-1]}}, in1[BIT_WIDTH-1:0]};
41         end else begin
42             out <= 0;
43         end
44 end
45
46 endmodule // ACCUM
```

Listing I.7: ACCUM

I.8 ADD

```
1
2 module ADD (
3     reset ,
4     clk ,
5     scan_in0 ,
6     scan_en ,
7     test_mode ,
8     scan_out0 ,
9     in1 ,
10    in2 ,
11    out
12    );
13
14 parameter BIT_WIDTH=16;
15 localparam BIT_WIDTH7P = BIT_WIDTH + 7;
16
17 input
18     reset ,                // system reset
19     clk ;                  // system clock
20
21 input
22     scan_in0 ,            // test scan mode data input
23     scan_en ,            // test scan mode enable
```

```
24     test_mode;                // test mode select
25
26     input  [BIT_WIDTH7P:0] in1;
27     input  [BIT_WIDTH7P:0] in2;
28
29     output reg [BIT_WIDTH7P:0] out;
30
31     output scan_out0;         // test scan mode data
32     output
33 always @ (posedge clk or posedge reset) begin
34     if(reset) begin
35         out <= {BIT_WIDTH7P{1'b0}};
36     end else begin
37         out <= {in1[BIT_WIDTH7P], in1[BIT_WIDTH7P-1:0]} + {in2[
38             BIT_WIDTH7P:0]};
39         //out <= {in1[BIT_WIDTH7P-1], in1[BIT_WIDTH7P-1:0]} + {in2[
40             BIT_WIDTH7P-1], in2[BIT_WIDTH7P-1:0]};
41     end
42 endmodule // ADD
```

Listing I.8: ADD

I.9 MUL

```
1
2 module MUL #(parameter BIT_WIDTH=16) (
3     reset ,
4     clk ,
5     scan_in0 ,
6     scan_en ,
7     test_mode ,
8     scan_out0 ,
9     in1 ,
10    in2 ,
11    out
12    );
13
14 input
15     reset ,                // system reset
16     clk ;                 // system clock
17
18 input
19     scan_in0 ,            // test scan mode data input
20     scan_en ,            // test scan mode enable
21     test_mode ;         // test mode select
22 input
23     [BIT_WIDTH-1:0] in1 , in2 ;
```

```
24
25 output
26     scan_out0;           // test scan mode data output
27
28 output reg [BIT_WIDTH-1:0] out;
29
30 localparam BIT_WIDTH1M = BIT_WIDTH-1;
31
32 always @(posedge clk or posedge reset)
33 begin
34     if(reset) begin
35         out <= 0;
36     end else begin
37         out <= (((BIT_WIDTH{in1 [BIT_WIDTH-1]}}, {in1 [BIT_WIDTH
38             -1:0]}}) * {{BIT_WIDTH{in2 [BIT_WIDTH-1]}}, {in2 [BIT_WIDTH
39             -1:0]}}) >> BIT_WIDTH1M;
40
41     end
42 end
43 endmodule // MUL
```

Listing I.9: MUL

I.10 RELU

```
1
2 module RELU (
3     reset ,
4     clk ,
5     scan_in0 ,
6     scan_en ,
7     test_mode ,
8     scan_out0 ,
9     val_in ,
10    compare ,
11    val_out
12 );
13
14 parameter IP_WIDTH=20;
15 parameter OP_WIDTH=16;
16
17 input
18     reset ,                // system reset
19     clk ;                  // system clock
20
21 input
22     scan_in0 ,            // test scan mode data input
23     scan_en ,            // test scan mode enable
```

```
24     test_mode;                // test mode select
25
26 input  [IP_WIDTH-1:0] val_in;
27 input  [OP_WIDTH-1:0] compare;
28 //reg   [OP_WIDTH-1:0] compare; // 20 bits, pass in a
    parameter as per the layer.
29
30 output [OP_WIDTH-1:0] val_out;
31
32 output
33     scan_out0;                // test scan mode data output
34
35 // assign val_out = val_in[IP_WIDTH-1] ? 0 : ((val_in > {{(
    IP_WIDTH - OP_WIDTH){compare[OP_WIDTH-1]}}, compare[OP_WIDTH
    -1:0]}}) ? compare : val_in);
36
37 assign val_out = val_in[IP_WIDTH-1] ? 0 : ((val_in > compare) ?
    compare : val_in);
38
39 /* always @(posedge clk)
40 begin
41     if(reset) begin
42         val_out <= 0;
43     end else begin
44         if(val_in[IP_WIDTH-1])
```

```
45     // ReLu
46     val_out <= 0;
47     else if(val_in > {{{(IP_WIDTH - OP_WIDTH){compare[OP_WIDTH
        -1]}}} ,compare[OP_WIDTH-1:0]))
48     // clipping
49     val_out <= compare;
50     else
51     val_out <= val_in;
52     end
53 end */
54
55 endmodule // RELU
```

Listing I.10: RELU

I.11 MAXPOOL

```
1
2 module MAXPOOL #(parameter BIT_WIDTH=16) (
3     reset ,
4     clk ,
5     scan_in0 ,
6     scan_en ,
7     test_mode ,
8     scan_out0 ,
9     in1 ,
10    in2 ,
11    in3 ,
12    in4 ,
13    max_out
14    );
15
16 input
17     reset ,                // system reset
18     clk ;                  // system clock
19
20 input
21     scan_in0 ,            // test scan mode data input
22     scan_en ,            // test scan mode enable
23     test_mode ;          // test mode select
```

```
24
25 input [BIT_WIDTH-1:0] in1 , in2 , in3 , in4;
26
27 output [BIT_WIDTH-1:0] max_out;
28
29 output
30     scan_out0;                // test scan mode data output
31
32 wire [BIT_WIDTH-1:0] max1, max2;
33
34 assign max1 = (in1>in2) ? in1 : in2;
35 assign max2 = (in3>in4) ? in3 : in4;
36 assign max_out = (max1>max2) ? max1 : max2;
37
38 // assign max_out = (((in1>in2) ? in1 : in2)>((in3>in4) ? in3 :
    in4)) ? ((in1>in2) ? in1 : in2) : ((in3>in4) ? in3 : in4);
39
40 /* always @(posedge clk)
41 begin
42     if(reset) begin
43         max_out <= 0;
44         max1 <= 0;
45         max2 <= 0;
46     end else begin
47         max1 <= (in1>in2) ? in1 : in2;
```

```
48     max2 <= (in3>in4) ? in3 : in4;
49     end
50 end */
51
52 endmodule // MAXPOOL
```

Listing I.11: MAXPOOL

I.12 ICNN RTL

```
1
2 module ICNN (
3     reset ,
4     clk ,
5     scan_in0 ,
6     scan_en ,
7     test_mode ,
8     scan_out0 ,
9     o_m_wb_adr ,
10    o_m_wb_sel ,
11    o_m_wb_stb ,
12    i_m_wb_ack ,
13    i_m_wb_err ,
14    o_m_wb_we ,
15    o_m_wb_cyc ,
16    o_m_wb_dat ,
17    o_s_wb_ack ,
18    o_s_wb_err ,
19    i_s_wb_adr ,
20    i_s_wb_sel ,
21    i_s_wb_we ,
22    i_s_wb_dat ,
23    o_s_wb_dat ,
```

```
24     i_m_wb_dat ,
25     i_s_wb_cyc ,
26     i_s_wb_stb
27     // ICNN_done
28     );
29
30 `include " ../include/register_addresses.vh"
31 `include " ../include/icnn_state_defs.vh"
32
33 parameter BIT_WIDTH  = 16;
34 parameter WB_DWIDTH  = 32;
35 parameter WB_SWIDTH  = 4 ;
36 parameter NUM_LAYERS = 2;
37
38 parameter MAX_IP_SIZE_R = 64;
39 parameter MAX_IP_SIZE_C = 64;
40 parameter MAX_OP_SIZE_R = 62;
41 parameter MAX_OP_SIZE_C = 62;
42
43 parameter COMPARE_3      = 9830;
44 parameter COMPARE_8      = 26213;
45
46 localparam BIT_WIDTH7P   = BIT_WIDTH + 7;
47 localparam MAX_OPI_MR    = MAX_OP_SIZE_R - 1;
48 localparam MAX_OPI_MC    = MAX_OP_SIZE_C - 1;
```

49

50

51 **input**

52 reset , // system reset

53 clk ; // system clock

54

55 **input**

56 scan_in0 , // test scan mode data input

57 scan_en , // test scan mode enable

58 test_mode ; // test mode select

59

60 **output**

61 scan_out0 ; // test scan mode data output

62

63 //

64 // wishbone master and slave ports

65 //

66 **input** [WB_DWIDTH-1:0] i_m_wb_dat ;67 **input** i_m_wb_ack ;68 **input** i_m_wb_err ;69 **input** [31:0] i_s_wb_adr ;

```
70 input      [WB_SWIDTH-1:0]  i_s_wb_sel ;
71 input      i_s_wb_we ;
72 input      [WB_DWIDTH-1:0]  i_s_wb_dat ;
73 input      i_s_wb_cyc ;
74 input      i_s_wb_stb ;
75
76 output     [WB_DWIDTH-1:0]  o_m_wb_adr ;
77 output reg  [WB_SWIDTH-1:0]  o_m_wb_sel ;
78 output reg          o_m_wb_we ;
79 output reg  [WB_DWIDTH-1:0]  o_m_wb_dat ;
80 output reg          o_m_wb_cyc ;
81 output reg          o_m_wb_stb ;
82 output     [WB_DWIDTH-1:0]  o_s_wb_dat ;
83 output          o_s_wb_ack ;
84 output          o_s_wb_err ;
85
86 wire      [WB_DWIDTH-1:0]  o_s_wb_dat ;
87 wire      [WB_DWIDTH-1:0]  o_m_wb_adr ;
88
89 reg      [WB_DWIDTH-1:0]  curr_adr ;
90 //=====ADD=====
91 reg  [BIT_WIDTH7P:0]  add_in1 ;
92 reg  [BIT_WIDTH7P:0]  add_in2 ;
93 wire [BIT_WIDTH7P:0]  add_out ;
94 //=====ACCUM=====
```



```

95 reg          fc_start_bit ;
96 reg [BIT_WIDTH-1:0]  fc_in1 ;
97 reg          start_bit_conv ;
98 reg [BIT_WIDTH-1:0]  in1_conv ;
99 wire          start_bit_mux ;
100 wire [BIT_WIDTH-1:0 ] in1_mux ;
101 wire [BIT_WIDTH7P:0]  acc_out ;
102 //=====MUL=====
103 wire [BIT_WIDTH-1:0]  im_mux ;
104 wire [BIT_WIDTH-1:0]  wt_mux ;
105 reg [BIT_WIDTH-1:0]  im_conv ;
106 reg [BIT_WIDTH-1:0]  wt_conv ;
107 reg [BIT_WIDTH-1:0]  fc_im ;
108 reg [BIT_WIDTH-1:0]  fc_wt ;
109 //output decoded in the state machines
110 wire [BIT_WIDTH-1:0]  mul_out ;
111 //=====RELU=====
112 reg [BIT_WIDTH7P:0]  fc_val_in ;
113 reg [BIT_WIDTH7P:0]  val_in_conv ;
114 wire [BIT_WIDTH7P:0]  val_in_mux ;
115 wire [BIT_WIDTH-1:0]  cmp_mux ;
116 wire [BIT_WIDTH-1:0]  relu_out ;
117 //=====MAXPOOL=====
118 reg [BIT_WIDTH-1:0]  mp1_val_1 ,
119          mp1_val_2 ,

```

```
120         mp1_val_3 ,
121         mp1_val_4 ;
122 wire [BIT_WIDTH-1:0]    mp1_out_1 ;
123
124 //=====
125 //===== Master R/W State Machine =====
126
127
128
129 //=====
130 //===== LAYER CONFIG =====
131
132 //----- Convolution -----
133 reg [7:0] conv_ip_size_r , //IP_SIZE row
134 conv_ip_size_2r , //IP_SIZE2X (2xrow)
135 conv_ip_size_c , //IP_SIZE col
136
137 conv_op_size_r , //OP_SIZE row
138 conv_op_size_c , //OP_SIZE col
139
140 conv_mp_size_r , //MP_SIZE row
141 conv_mp_size_c , //MP_SIZE col
142 conv_num_fil , //FIL_NUM
143 conv_num_ch ; //CH_NUM
144
```

```

145 reg [15:0] conv_ip_size_rc , //IP_SIZE_SQ
146     conv_op_size_rc; //OP_SIZE_SQ
147
148 //----- Fully Connected -----
149 reg [15:0] fc_wt_row_cnt , //WT Row Count
150     fc_wt_col_cnt; //WT Col Count
151
152 //---- Other Layer Config Signals ----
153 reg conv_fc_mode ,
154     mp_flag ,
155     cmp_flag ,
156     ICNN_done;
157 reg [13:0] cnfg_adr;
158 reg [3:0] config_state;
159 reg [3:0] layer_count;
160
161 //=====
162
163 //===== Read Weights State Machine ==
164 //reg [BIT_WIDTH-1:0] wt_chann [0:8]; //
165     current set of weights
166 reg [BIT_WIDTH-1:0] wt_ch0; //current set
167     of weights
168 reg [BIT_WIDTH-1:0] wt_ch1;
169 reg [BIT_WIDTH-1:0] wt_ch2;

```

```
168 reg [BIT_WIDTH-1:0] wt_ch3;
169 reg [BIT_WIDTH-1:0] wt_ch4;
170 reg [BIT_WIDTH-1:0] wt_ch5;
171 reg [BIT_WIDTH-1:0] wt_ch6;
172 reg [BIT_WIDTH-1:0] wt_ch7;
173 reg [BIT_WIDTH-1:0] wt_ch8;
174
175
176 reg [19:0] rd_wt_adr; //
    address pointer, 13:3 traverse vertical in MM,
177 //2:0 -> 16bit select, 2:1 -> 32
    bit select.
178 reg wt_rreq; //weights read
    request
179 reg [3:0] nth_wt; //
    weight state
180 //reg [5:0] wt_num_ch; //
    channel number WT
181 reg has_wts; //indicates weights are read in
    and are ready to convolve
182 //===== Read FMs State Machine =====
183 //reg [BIT_WIDTH-1:0] fm_buf1 [0:8]; //
    FM, fm -> Feature map,
184 //reg [BIT_WIDTH-1:0] fm_buf2 [0:8]; //fm
    values
```

```
185
186 reg [BIT_WIDTH-1:0]   fm_ch0;           // current set
      of weights
187 reg [BIT_WIDTH-1:0]   fm_ch1;
188 reg [BIT_WIDTH-1:0]   fm_ch2;
189 reg [BIT_WIDTH-1:0]   fm_ch3;
190 reg [BIT_WIDTH-1:0]   fm_ch4;
191 reg [BIT_WIDTH-1:0]   fm_ch5;
192 reg [BIT_WIDTH-1:0]   fm_ch6;
193 reg [BIT_WIDTH-1:0]   fm_ch7;
194 reg [BIT_WIDTH-1:0]   fm_ch8;
195
196 reg [13:0]            rd_fm_adr;         //
      main FM address pointer
197 reg [13:0]            r1_fm_ptr ,      //FM
      row1 adr ptr
198           r2_fm_ptr ,                 //FM row2 adr ptr
199           r3_fm_ptr ;                 //FM row3 adr ptr
200 reg                   fm_rreq;         //FM read request
201 reg [3:0]              nth_fm;         //feature map state
202 reg [5:0]              conv_ch_count ,  //number of input
      channels of FM
203           conv_fil_count;             // Filter number WT =
      number of output channels should be equal;
```

```
204 reg [5:0]          row_cnt,          //row count to slide
      windows vertically thru FM
205          col_cnt;                  //column count to
      slide windows horizontally thru FM
206 reg          fm_ch_cmplt,          //indicates completion of one
      complete channel of input FM
207          fm_cmplt,                //indicates completion of all
      channels of input FM
208          all_fm_cmplt;
209 reg          has_fm_1;             //indicates fm_buf1 has values
      that are ready to convolve
210 //          has_fm_2;             //indicates fm_buf2 has values
      that are ready to convolve
211 //===== ACCMUL, RELU, MAXPOOL =====
212 reg [BIT_WIDTH7P:0] conv_op[0:MAX_OPIMR][0:MAX_OPMC]; //
      stores output of convolution until second stage of
      accumulation and RELU
213 reg [4:0]          mul_cnt;          //accumul state
214 reg [5:0]          curr_ch,          //store current channel
      from FM State Machine
215          curr_row,                //store current row
216          curr_col;                //store current col
217 reg [6:0]          mp_row,          //row count for MAXPOOL
218          mp_col,                  //col count for MAXPOOL
```

```
219         mp_row_1p,           //row count + 1 for MAXPOOL
           indexing
220         mp_col_1p;          //col count + 1 for MAXPOOL
           indexing
221 reg [6:0]          mp_fil_count;           //count number
           of times mp has completed.
222 reg              acc_mul_cmplt;          //indicates
           completion of one convolution operation.
223              //dotprod of 9 wts and 9fms, and accumulate 9
           resulting values.
224 reg              has_relu_fm;           //turns on after second stage
           of accumulation
225              //2nd stage accum-> adds all channel data.
226              //perform relu
227              //send to MAXPOOL when fm_cmplt turns on
228 reg [13:0]        rd_op_adr;            //address
           pointer output -> writes to MM
229 reg              op_wreq,              //write request to memory
           after every single value is computed.
230         mp_cmplt;          //indicates completion of MAXPOOL
           operation.
231 reg [WB_DWIDTH-1:0] write_buf;         //Maxpool state writes
           result to buffer that's to be written to MM.
232 reg              selx_0, selx_1;       //selecting which part of
           16 bits of 32bits to write to.
```

```
233 reg [7:0] mp_iterator_r;
234 reg [7:0] mp_iterator_c;
235 //=====Wishbone Interface=====
236 reg [6:0] wb_adr_wt_base_ptr;
237 reg [12:0] wb_adr_fm_base_ptr;
238 reg [12:0] wb_adr_op_base_ptr;
239 reg [12:0] wb_adr_cnfg_base_ptr;
240 reg [16:0] wb_adr_wt_ptr;
241 reg [10:0] wb_adr_fm_ptr ,
242 wb_adr_op_ptr ,
243 wb_adr_cnfg_ptr; //11 bits to
traverse vertically in main_mem
244 reg [31:0] wb_rdata32;
245 wire [31:0] wb_wdata32;
246
247 reg valid_wt_base_adr ,
248 valid_fm_base_adr ,
249 valid_op_base_adr ,
250 start_bit;
251 reg start_read_1;
252 wire start_read ,
253 start_write;
254
255
256 //===== FULLY CONNECTED =====
```



```
257 // reg [15:0]          fc_fm_buf [0:1];
258 // reg [15:0]          fc_wt_buf [0:1];
259
260 reg [15:0]             fc_fm_0 ,
261                      fc_fm_1 ,
262                      fc_wt_0 ,
263                      fc_wt_1 ;
264
265 reg [31:0]             fc_op_buf ;
266 reg [4:0]              fc_state ;
267 reg [10:0]             fc_row_count ,
268                      fc_col_count ,
269                      r_fake ,
270                      c_fake ;
271
272 reg [19:0]             fc_wt_adr ;
273 reg [13:0]             fc_fm_adr ;
274 reg [13:0]             fc_op_adr ;
275 reg                    fc_wt_rreq ,
276                      fc_fm_rreq ,
277                      fc_op_wreq ;
278 reg                    fc_done ,
279                      fc_sel_0 ,
280                      fc_sel_1 ;
281 reg [2:0]              fc_count ;
```

```
282 //=====
283 wire          reset_mux;
284 reg          sw_reset;
285
286 integer          j, //to reset conv_op rows
287                 k; //to reset conv_op cols
288
289 ADD #(.BIT_WIDTH(BIT_WIDTH)) adder(.reset(reset_mux),
290     .clk(clk),
291     .scan_in0(scan_in0),
292     .scan_en(scan_en),
293     .test_mode(test_mode),
294     .scan_out0(scan_out0),
295     .in1(add_in1),
296     .in2(add_in2),
297     .out(add_out));
298
299
300 ACCUM #(.BIT_WIDTH(BIT_WIDTH)) acc_ch1(.reset(reset_mux),
301     .clk(clk),
302     .scan_in0(scan_in0),
303     .scan_en(scan_en),
304     .test_mode(test_mode),
305     .scan_out0(scan_out0),
306     .start_bit(start_bit_mux),
```

```
307         .in1 (in1_mux),
308         .out (acc_out)
309     );
310
311 MUL #(.BIT_WIDTH(BIT_WIDTH)) mul_ch1 (.reset (reset_mux),
312         .clk (clk),
313         .scan_in0 (scan_in0),
314         .scan_en (scan_en),
315         .test_mode (test_mode),
316         .scan_out0 (scan_out0),
317         .in1 (im_mux),
318         .in2 (wt_mux),
319         .out (mul_out)
320     );
321
322 RELU #(.IP_WIDTH(BIT_WIDTH*7P+1), .OP_WIDTH(BIT_WIDTH)) relu_ch1
323     (.reset (reset_mux),
324         .clk (clk),
325         .scan_in0 (scan_in0),
326         .scan_en (scan_en),
327         .test_mode (test_mode),
328         .scan_out0 (scan_out0),
329         .val_in (val_in_mux),
330         .compare (cmp_mux),
331         .val_out (relu_out)
```

```
331         );
332
333 MAXPOOL #(.BIT_WIDTH(BIT_WIDTH)) mp_1(.reset(reset_mux),
334         .clk(clk),
335         .scan_in0(scan_in0),
336         .scan_en(scan_en),
337         .test_mode(test_mode),
338         .scan_out0(scan_out0),
339         .in1(mp1_val_1),
340         .in2(mp1_val_2),
341         .in3(mp1_val_3),
342         .in4(mp1_val_4),
343         .max_out(mp1_out_1)
344     );
345
346
347 assign reset_mux      = (test_mode) ? reset : (sw_reset ||
    reset);
348 assign o_m_wb_adr    = curr_adr;
349 // Implementing MUX for ACCUM and MUL – Reusing hardware for FC
350 assign im_mux        = (conv_fc_mode) ? fc_im : im_conv;
351 assign wt_mux        = (conv_fc_mode) ? fc_wt : wt_conv;
352 assign start_bit_mux = (conv_fc_mode) ? fc_start_bit :
    start_bit_conv;
353 assign in1_mux       = (conv_fc_mode) ? fc_in1 : in1_conv;
```

```
354 assign val_in_mux      = (conv_fc_mode) ? fc_val_in : val_in_conv;
355 assign cmp_mux        = (cmp_flag)? COMPARE_8:COMPARE_3;
356
357 //=====
358 //-----  State Machine - Convolution  -----
359 //=====
360
361
362 //=====
363 //-----  Master READS and WRITES  -----
364 //=====
365 always @(posedge clk or posedge reset_mux)
366 begin
367     if(reset_mux) begin
368         //o_m_wb_adr <= 0;
369         curr_adr    <= 32'b0;
370         o_m_wb_sel <= 0;
371         o_m_wb_dat <= 0;
372         o_m_wb_we  <= 0;
373         o_m_wb_cyc <= 0;
374         o_m_wb_stb <= 0;
375     end else begin
376         if(!start_bit) begin
377             o_m_wb_cyc <= 1'b0;
378             o_m_wb_stb <= 1'b0;
```

```
379     o_m_wb_we  <= 1'b0;
380     o_m_wb_cyc <= 1'b0;
381     o_m_wb_stb <= 1'b0;
382     o_m_wb_sel <= 4'b0;
383     end else begin
384     if (!ICNN_done && !valid_wt_base_adr && !valid_fm_base_adr
        && !valid_op_base_adr) begin
385         curr_adr  <= {4'b0, wb_adr_cnfg_base_ptr, cnfg_adr
            [13:1], 2'b0};
386         o_m_wb_cyc <= 1'b1;
387         o_m_wb_stb <= 1'b1;
388         if (i_m_wb_ack) begin
389             o_m_wb_we  <= 1'b0;
390             o_m_wb_cyc <= 1'b0;
391             o_m_wb_stb <= 1'b0;
392             o_m_wb_sel <= 4'b0;
393         end
394     end else if (valid_wt_base_adr && valid_fm_base_adr &&
        valid_op_base_adr) begin
395         if (!conv_fc_mode) begin
396             if (mp_fil_count < conv_num_fil) begin
397                 if (valid_wt_base_adr && !has_wts && !acc_mul_cmplt &&
                    !has_relu_fm) begin //&& !acc_mul_cmplt not sure
398                     curr_adr  <= {4'b0, wb_adr_wt_base_ptr, rd_wt_adr
                        [19:1], 2'b0};
```

```
399         o_m_wb_cyc <= 1'b1;
400         o_m_wb_stb <= 1'b1;
401
402         // Access bus strictly when necessary, has_wts is ON
           and has_fm is low
403         // and fm_ch_cmplt goes high next cycle which turns
           has_wts low.
404         // and requests bus access to read weights but fm is
           read instead -> hence added !fm_ch_cmplt.
405     end else if (valid_fm_base_addr && has_wts && !has_fm_1
           && !fm_ch_cmplt) begin
406         curr_addr <= {4'b0, wb_addr_fm_base_ptr, rd_fm_addr
           [13:1], 2'b0};
407         o_m_wb_cyc <= 1'b1;
408         o_m_wb_stb <= 1'b1;
409
410     end else if (valid_op_base_addr && has_relu_fm &&
           op_wreq) begin
411         curr_addr <= {4'b0, wb_addr_op_base_ptr, rd_op_addr
           [13:1], 2'b0};
412         o_m_wb_cyc <= 1'b1;
413         o_m_wb_stb <= 1'b1;
414         o_m_wb_we <= 1'b1;
415         o_m_wb_dat <= write_buf;
416         o_m_wb_sel <= {{2{selx_1}}, {2{selx_0}}};
```

```
417         end else begin
418             o_m_wb_cyc <= 1'b0;
419             o_m_wb_stb <= 1'b0;
420             o_m_wb_we  <= 1'b0;
421             o_m_wb_sel <= 4'b0;
422         end
423
424         if (i_m_wb_ack) begin
425             o_m_wb_we  <= 1'b0;
426             o_m_wb_cyc <= 1'b0;
427             o_m_wb_stb <= 1'b0;
428             o_m_wb_sel <= 4'b0;
429         end
430     end
431 end else begin //conv_fc_mode is high -> FC layer
432
433     if(fc_wt_rreq && !fc_fm_rreq && !fc_op_wreq) begin //
434         && !acc_mul_cmplt not sure
435         curr_adr    <= {4'b0, wb_adr_wt_base_ptr, fc_wt_adr
436             [19:1], 2'b0};
437         o_m_wb_cyc <= 1'b1;
438         o_m_wb_stb <= 1'b1;
439     end else if (!fc_wt_rreq && fc_fm_rreq && !fc_op_wreq
440         ) begin
```



```
438         curr_addr    <= {4'b0, wb_addr_fm_base_ptr, fc_fm_addr
           [13:1], 2'b0};
439         o_m_wb_cyc <= 1'b1;
440         o_m_wb_stb <= 1'b1;
441     end else if (!fc_wt_rreq && !fc_fm_rreq && fc_op_wreq)
           begin
442         curr_addr    <= {4'b0, wb_addr_op_base_ptr, fc_op_addr
           [13:1], 2'b0};
443         o_m_wb_cyc <= 1'b1;
444         o_m_wb_stb <= 1'b1;
445         o_m_wb_we   <= 1'b1;
446         o_m_wb_dat <= fc_op_buf;
447         o_m_wb_sel <= {{2{fc_sel_1}}, {2{fc_sel_0}}};
448     end else begin
449         o_m_wb_cyc <= 1'b0;
450         o_m_wb_stb <= 1'b0;
451         o_m_wb_we   <= 1'b0;
452         o_m_wb_sel <= 4'b0;
453     end
454
455     if (i_m_wb_ack) begin
456         o_m_wb_we   <= 1'b0;
457         o_m_wb_cyc <= 1'b0;
458         o_m_wb_stb <= 1'b0;
459         o_m_wb_sel <= 4'b0;
```

```
460         end
461
462     end
463 end
464 end
465 end
466 end // always
467
468 //=====
469 //----- Layer - Configuration -----
470 //=====
471
472 always @(posedge clk or posedge reset_mux)
473 begin
474     if(reset_mux) begin
475         conv_ip_size_r    <= 8'b0;
476         conv_ip_size_2r  <= 8'b0;
477         conv_ip_size_c    <= 8'b0;
478         conv_op_size_r    <= 8'b0;
479         conv_op_size_c    <= 8'b0;
480         conv_mp_size_r    <= 8'b0;
481         conv_mp_size_c    <= 8'b0;
482
483         conv_ip_size_rc   <= 16'b0;
484         conv_op_size_rc   <= 16'b0;
```

```
485
486     conv_num_fil      <= 8'b0;
487     conv_num_ch       <= 8'b0;
488
489     fc_wt_row_cnt     <= 16'b0;
490     fc_wt_col_cnt     <= 16'b0;
491
492     conv_fc_mode      <= 1'b0;
493     mp_flag           <= 1'b0;
494     cmp_flag          <= 1'b0;
495
496     valid_wt_base_adr <= 1'b0;
497     wb_adr_wt_base_ptr <= 11'b0;
498     wb_adr_wt_ptr     <= 17'b0;
499
500     valid_fm_base_adr <= 1'b0;
501     wb_adr_fm_base_ptr <= 13'b0;
502     wb_adr_fm_ptr     <= 11'b0;
503
504     valid_op_base_adr <= 1'b0;
505     wb_adr_op_base_ptr <= 13'b0;
506     wb_adr_op_ptr     <= 11'b0;
507
508     layer_count       <= 4'b0;
509     cnfg_adr          <= 13'b0;
```

```
510         config_state         <= 4'b0;
511
512     ICNN_done                 <= 1'b0;
513
514     end else begin
515         if(!start_bit) begin
516             //reset or hold?
517         end else begin
518             case(config_state)
519                 'CNFG_IDLE: begin
520                     if(!fc_done && !valid_wt_base_adr && !valid_fm_base_adr
521                         && !valid_op_base_adr) begin
522                         conv_ip_size_r   <= 8'b0;
523                         conv_ip_size_2r <= 8'b0;
524                         conv_ip_size_c   <= 8'b0;
525                         conv_op_size_r   <= 8'b0;
526                         conv_op_size_c   <= 8'b0;
527                         conv_mp_size_r   <= 8'b0;
528                         conv_mp_size_c   <= 8'b0;
529
530                         conv_ip_size_rc <= 16'b0;
531                         conv_op_size_rc <= 16'b0;
532
533                         conv_num_fil    <= 8'b0;
534                         conv_num_ch     <= 8'b0;
```

```
534
535         fc_wt_row_cnt    <= 16'b0;
536         fc_wt_col_cnt    <= 16'b0;
537
538         conv_fc_mode     <= 1'b0;
539         mp_flag          <= 1'b0;
540         cmp_flag         <= 1'b0;
541
542         valid_wt_base_adr <= 1'b0;
543         wb_adr_wt_base_ptr <= 13'b0;
544         wb_adr_wt_ptr     <= 17'b0;
545
546         valid_fm_base_adr <= 1'b0;
547         wb_adr_fm_base_ptr <= 13'b0;
548         wb_adr_fm_ptr     <= 11'b0;
549
550         valid_op_base_adr <= 1'b0;
551         wb_adr_op_base_ptr <= 13'b0;
552         wb_adr_op_ptr     <= 11'b0;
553
554         // layer_count    <= 4'b0;
555         cnfg_adr          <= {wb_adr_cnfg_ptr, {3'b0}}; // set
                    this the first time only
556         config_state <= 'CNFG_START_BIT;
557
```

```
558         ICNN_done     <= 0; //continues to read FM if start
           bit is on.
559
560     end else if (!ICNN_done && valid_wt_base_adr &&
           valid_fm_base_adr && valid_op_base_adr) begin
561         if (((!conv_fc_mode) && (mp_fil_count == conv_num_fil)
           ) || (conv_fc_mode && fc_done)) begin
562             //Clear/reset all config params after completion of
           conv/fc layer.
563             //other than config base pointer.
564             conv_ip_size_r   <= 8'b0;
565             conv_ip_size_2r  <= 8'b0;
566             conv_ip_size_c   <= 8'b0;
567             conv_op_size_r   <= 8'b0;
568             conv_op_size_c   <= 8'b0;
569             conv_mp_size_r   <= 8'b0;
570             conv_mp_size_c   <= 8'b0;
571
572             conv_ip_size_rc  <= 16'b0;
573             conv_op_size_rc  <= 16'b0;
574
575             conv_num_fil     <= 8'b0;
576             conv_num_ch      <= 8'b0;
577
578             fc_wt_row_cnt    <= 16'b0;
```

```
579         fc_wt_col_cnt    <= 16'b0;
580
581         conv_fc_mode     <= 1'b0;
582         mp_flag          <= 1'b0;
583         cmp_flag         <= 1'b0;
584
585         valid_wt_base_adr <= 1'b0;
586         wb_adr_wt_base_ptr <= 13'b0;
587         wb_adr_wt_ptr     <= 17'b0;
588
589         valid_fm_base_adr <= 1'b0;
590         wb_adr_fm_base_ptr <= 13'b0;
591         wb_adr_fm_ptr     <= 11'b0;
592
593         valid_op_base_adr <= 1'b0;
594         wb_adr_op_base_ptr <= 13'b0;
595         wb_adr_op_ptr     <= 11'b0;
596
597         if(layer_count < NUM_LAYERS-1) begin
598             layer_count    <= layer_count +1;
599             config_state   <= 'CNFG_START_BIT;
600         end else if(layer_count == NUM_LAYERS-1) begin
601             layer_count    <= 0;
602             ICNN_done      <= 1;
603         end
```

```
604         end
605     end
606 end
607
608 'CNFG_START_BIT: begin
609     if(i_m_wb_ack) begin
610         {mp_flag , cmp_flag , conv_fc_mode} <= i_m_wb_dat
611             [2:0];    //3 bits
612         {conv_num_fil , conv_num_ch}         <= i_m_wb_dat
613             [31:16]; //8+8 bits
614         conv_ip_size_2r                       <= i_m_wb_dat
615             [15:8]; //8 bits
616         cnfg_adr      <= cnfg_adr + 2;          //read 32 bits
617             -> +2
618         config_state <= 'CNFG_IP_SIZE;
619     end
620 end
621
622 'CNFG_IP_SIZE: begin
623     if(i_m_wb_ack) begin
624         if(conv_fc_mode) begin
625             {fc_wt_row_cnt , fc_wt_col_cnt} <= i_m_wb_dat;    //
626                 16+16 bits
627             cnfg_adr      <= cnfg_adr + 2;          //read 32 bits
628                 -> +2
```



```
623         config_state    <= 'CNFG_WT_BASE_ADR;
624     end else begin
625         conv_ip_size_r <= i_m_wb_dat[23:16];
626         conv_ip_size_c <= i_m_wb_dat[7:0];    //8+8 skip 8+8
627         cnfg_adr        <= cnfg_adr + 2;    //read 32 bits
           -> +2
628         // {conv_op_size_r , conv_op_size_c} <= i_m_wb_dat
           [32:16];
629         config_state    <= 'CNFG_MP_SIZE;
630     end
631 end
632 end
633
634 'CNFG_MP_SIZE: begin
635     if(i_m_wb_ack) begin
636         {conv_op_size_r , conv_op_size_c} <= i_m_wb_dat
           [31:16];
637         {conv_mp_size_r , conv_mp_size_c} <= i_m_wb_dat[15:0];
638         cnfg_adr        <= cnfg_adr + 2;
639         config_state    <= 'CNFG_RxC_SIZE;
640     end
641 end
642
643 'CNFG_RxC_SIZE: begin
644     if(i_m_wb_ack) begin
```

```
645         { conv_ip_size_rc , conv_op_size_rc } <= i_m_wb_dat;
646         cnfg_adr         <= cnfg_adr + 2;
647         config_state     <= 'CNFG_WT_BASE_ADR;
648     end
649 end
650
651 'CNFG_WT_BASE_ADR: begin
652     if(i_m_wb_ack) begin
653         { wb_adr_wt_base_ptr , wb_adr_wt_ptr } <= i_m_wb_dat
654             [23:0];
655         cnfg_adr         <= cnfg_adr + 2;
656         config_state     <= 'CNFG_FM_BASE_ADR;
657     end
658 end
659
660 'CNFG_FM_BASE_ADR: begin
661     if(i_m_wb_ack) begin
662         { wb_adr_fm_base_ptr , wb_adr_fm_ptr } <= i_m_wb_dat
663             [23:0];
664         cnfg_adr         <= cnfg_adr + 2;
665         config_state     <= 'CNFG_OP_BASE_ADR;
666     end
667 end
668
669 'CNFG_OP_BASE_ADR: begin
```

```
668         if(i_m_wb_ack) begin
669             {wb_adr_op_base_ptr , wb_adr_op_ptr} <= i_m_wb_dat
670                 [23:0];
671             cnfg_adr          <= cnfg_adr + 2;
672             valid_op_base_adr <= 1;
673             valid_fm_base_adr <= 1;
674             valid_wt_base_adr <= 1;
675             config_state     <= 'CNFG_IDLE;
676         end
677     end
678     default: begin
679         config_state <= 'CNFG_IDLE;
680     end
681 endcase
682 end
683 end
684 end // always
685
686
687 //=====
688 //----- Fully Connected -----
689 //=====
690 always @(posedge clk or posedge reset_mux)
691 begin
```

```
692     if (reset_mux) begin
693         fc_im         <= 0;
694         fc_wt         <= 0;
695         fc_start_bit <= 0;
696         fc_in1        <= 0;
697         fc_val_in     <= 0;
698
699         fc_fm_0       <= 0;
700         fc_fm_1       <= 0;
701         fc_wt_0       <= 0;
702         fc_wt_1       <= 0;
703
704         fc_state      <= 0;
705         fc_wt_adr     <= 0;
706         fc_fm_adr     <= 0;
707         fc_op_adr     <= 0;
708         fc_row_count  <= 0;
709         fc_col_count  <= 0;
710         r_fake        <= 0;
711         c_fake        <= 0;
712         fc_done       <= 0;
713         fc_wt_rreq    <= 0;
714         fc_fm_rreq    <= 0;
715         fc_op_wreq    <= 0;
716         fc_op_buf     <= 0;
```

```
717     fc_sel_0     <= 0;
718     fc_sel_1     <= 0;
719
720     fc_count <= 0;
721 end else begin
722     case( fc_state )
723         'IDLE_FC: begin
724             if (!fc_done && valid_wt_base_adr && valid_fm_base_adr
725                 && valid_op_base_adr && conv_fc_mode) begin
726                 //enters here the first time it's reading weights.
727
728                 // Transition
729                 fc_fm_0     <= 0;
730                 fc_fm_0     <= 0;
731                 fc_wt_0     <= 0;
732                 fc_wt_0     <= 0;
733
734                 // Original
735                 // fc_wt_buf[0] <= 0;
736                 // fc_wt_buf[1] <= 0;
737                 // fc_fm_buf[0] <= 0;
738                 // fc_fm_buf[1] <= 0;
739
```

```
740         //base address set at the beginning and is not
           reset
741         //everytime new set of weights are read.
742         fc_wt_adr    <= {wb_adr_wt_ptr , {3'b0}};
743         fc_fm_adr    <= {wb_adr_fm_ptr , {3'b0}};
744         fc_op_adr    <= {wb_adr_op_ptr , {3'b0}};
745         fc_wt_rreq   <= 1;
746         fc_fm_rreq   <= 0;
747         fc_op_wreq   <= 0;
748         fc_sel_0     <= 0;
749         fc_sel_1     <= 0;
750         fc_row_count <= 0;
751         fc_col_count <= 0;
752         r_fake       <= 0;
753         c_fake       <= 0;
754         fc_in1       <= 0;
755         if(ICNN_done) begin
756             //Assuming ICNN_done turns on after completion of
           last layer.
757             fc_count <= 0;
758         end
759
760         if(fc_count == 0) begin
761             fc_state    <= 'CONV_FC_WT0;
762         end else begin
```

```
763         fc_state      <= 'FC_WT0;
764     end
765 end
766 end else if (fc_done && !valid_wt_base_adr && !
       valid_fm_base_adr && !valid_op_base_adr ) begin
767     fc_done <= 0;
768     fc_wt_adr <= 0;
769     fc_fm_adr <= 0;
770     fc_row_count <= 0;
771     fc_col_count <= 0;
772     fc_count <= fc_count +1;
773     if (ICNN_done) begin
774         // Assuming ICNN_done turns on after completion of
           last layer.
775         fc_count <= 0;
776     end
777 end
778 end
779
780 //=====
781 //----- Layer 4 -----
782 //=====
783 'CONV_FC_WT0: begin
784     fc_wt_rreq <= 1;
785     if (i_m_wb_ack) begin
```

```
786         if (!fc_wt_adr[0]) begin
787             // Transition
788             // fc_wt_0 <= i_m_wb_dat[15:0]; //w0
789             fc_wt    <= i_m_wb_dat[15:0];
790         end else begin
791             // Transition
792             // fc_wt_0 <= i_m_wb_dat[31:16]; //w0
793             fc_wt    <= i_m_wb_dat[31:16];
794         end
795
796         fc_wt_rreq <= 0;
797         fc_fm_rreq <= 1;
798         // fc_wt_adr <= fc_wt_adr + (fc_wt_row_count+1)*
799             fc_wt_col_cnt+fc_wt_col_count;
800     end
801 end
802
803 'CONV_FC_FM0: begin
804     if (i_m_wb_ack) begin
805         if (!fc_fm_adr[0]) begin
806             // fc_fm_0 <= i_m_wb_dat[15:0]; //fm0
807             fc_fm    <= i_m_wb_dat[15:0];
808         end else begin
809             // fc_fm_0 <= i_m_wb_dat[31:16]; //fm0
```



```
810         fc_im      <= i_m_wb_dat[31:16];
811     end
812
813     fc_fm_rreq <= 0;
814     //fc_fm_adr <= fc_fm_adr + 2;
815     fc_state <= 'CONV_FC_MUL0;
816 end
817 end
818
819 'CONV_FC_MUL0: begin
820     //NOP
821     fc_state <= 'CONV_MUL_OUT;
822 end
823
824 'CONV_MUL_OUT: begin
825     fc_start_bit <= 1'b1;
826     fc_in1       <= mul_out;
827     fc_wt       <= 0;
828     fc_im       <= 0;
829     fc_state    <= 'CONV_FC_ACC;
830 end
831
832 'CONV_FC_ACC: begin
833     fc_in1      <= 0;
834     if(fc_row_count < fc_wt_row_cnt-1) begin
```

```
835         //fc_in1           <= mul_out;
836         //read weights and FM again
837         fc_row_count     <= fc_row_count + 1;
838         if(r_fake < conv_num_fil-1) begin
839             r_fake <= r_fake + 1;
840         end else begin
841             r_fake <= 0;
842             c_fake <= c_fake + 1;
843         end
844         fc_state         <= 'NEXT_ADDR;
845
846     end else begin
847         if(fc_col_count < fc_wt_col_cnt) begin
848             //fc_start_bit <= 1'b0;
849             //fc_in1     <= mul_out;
850             //on completing row count -> ReLu
851             fc_state <= 'CONV_ACC_OUT;
852         end else begin
853             fc_done <= 1;
854             fc_start_bit <= 1'b0;
855             fc_state <= 'IDLE_FC;
856         end
857     end
858 end
859
```

```
860     'CONV_ACC_OUT: begin
861         fc_start_bit <= 1'b0;
862         fc_state      <= 'CONV_RELU_OUT;
863     end
864
865     'CONV_RELU_OUT: begin
866         fc_val_in  <= acc_out;
867         fc_in1     <= 0;
868         fc_state   <= 'WREQ;
869     end
870
871     'NEXT_ADDR: begin
872         fc_wt_addr <= {wb_addr_wt_ptr, {3'b0}} + fc_row_count*
            fc_wt_col_cnt + fc_col_count;
873         fc_fm_addr <= {wb_addr_fm_ptr, {3'b0}} + conv_num_ch*
            conv_ip_size_2r*r_fake + c_fake;
874         // fc_in1 <= 0;
875         fc_state   <= 'CONV_FC_WT0;
876     end
877
878     //=====
879     //----- Layers 5 through 7 -----
880     //=====
881     'NEXT_WT0_ADDR: begin
882         fc_wt_rreq <= 1;
```

```
883         fc_wt_adr   <= {wb_adr_wt_ptr, {3'b0}} + fc_row_count*
           fc_wt_col_cnt + fc_col_count;
884         fc_state    <= 'FC_WT0;
885     end
886
887     'FC_WT0: begin
888         fc_in1 <= 0;
889         fc_wt_rreq <= 1;
890         if (i_m_wb_ack) begin
891             if (!fc_wt_adr[0]) begin
892                 // Transition
893                 fc_wt_0 <= i_m_wb_dat[15:0]; //w0
894                 //fc_wt    <= i_m_wb_dat[15:0];
895             end else begin
896                 // Transition
897                 fc_wt_0 <= i_m_wb_dat[31:16]; //w0
898                 //fc_wt    <= i_m_wb_dat[31:16];
899             end
900
901             fc_wt_rreq    <= 0;
902             //fc_fm_rreq <= 1;
903             fc_row_count <= fc_row_count + 1;
904
905             //fc_wt_adr   <= (fc_wt_row_count+1)*fc_wt_col_cnt+
           fc_wt_col_count;
```

```
906         fc_state      <= 'NEXT_WT1_ADDR;
907     end
908 end
909
910 'NEXT_WT1_ADDR: begin
911     fc_wt_rreq <= 1;
912     fc_wt_adr  <= {wb_adr_wt_ptr, {3'b0}} + fc_row_count*
          fc_wt_col_cnt + fc_col_count;
913     fc_state   <= 'FC_WT1;
914 end
915
916 'FC_WT1: begin
917     if (i_m_wb_ack) begin
918         if (!fc_wt_adr[0]) begin
919             // Transition
920             fc_wt_1 <= i_m_wb_dat[15:0]; //w1
921             //fc_wt <= i_m_wb_dat[15:0];
922         end else begin
923             // Transition
924             fc_wt_1 <= i_m_wb_dat[31:16]; //w1
925             //fc_wt <= i_m_wb_dat[31:16];
926         end
927
928     fc_wt_rreq <= 0;
929     fc_fm_rreq <= 1;
```

```
930         fc_row_count <= fc_row_count + 1;
931
932         // fc_wt_adr  <= (fc_wt_row_count+1)*fc_wt_col_cnt+
           fc_wt_col_count;
933         fc_state      <= 'FC_FM0;
934     end
935 end
936
937 'FC_FM0: begin
938     //address pointer is incremented by two or one every
           time based on number
939     //of values read in at once (based of parity bit.)
           parity check is done
940     //in every state unlike in FM read State Machine
           because the values are
941     //read continuously and even & odd indexes can occur in
           any state .
942     if (i_m_wb_ack) begin
943         if (!fc_fm_adr[0]) begin
944             //Transition
945             fc_fm_0  <= i_m_wb_dat[15:0]; //fm0
946             fc_fm_1  <= i_m_wb_dat[31:16]; //fm1
947
948             fc_fm_adr      <= fc_fm_adr + 2;
949             fc_fm_rreq     <= 0;
```

```
950         fc_wt_rreq    <= 0;
951         fc_state      <= 'FC_MUL0;
952     end else begin
953         // Transition
954         fc_fm_0        <= i_m_wb_dat[31:16]; //fm0
955
956         fc_fm_adr      <= fc_fm_adr + 1;
957         fc_state      <= 'FC_FM1;
958     end
959 end
960 end
961
962 'FC_FM1: begin
963     //address pointer is incremented by two or one every
964     //time based on number
965     //of values read in at once (based of parity bit.)
966     //parity check is done
967     //in every state unlike in FM read State Machine
968     //because the values are
969     //read continuously and even & odd indexes can occur in
970     //any state .
971     if (i_m_wb_ack) begin
972         if (!fc_fm_adr[0]) begin
973             // Transition
974             fc_fm_1    <= i_m_wb_dat[15:0]; //fm1
```

```
971
972         fc_fm_adr      <= fc_fm_adr + 1;
973         fc_fm_rreq     <= 0;
974         fc_wt_rreq     <= 0;
975         fc_state       <= 'FC_MUL0;
976     end
977 end
978 end
979
980 'FC_MUL0: begin
981     fc_im      <= fc_fm_0;
982     fc_wt      <= fc_wt_0;
983     fc_state   <= 'FC_MUL1;
984 end
985
986 'FC_MUL1: begin
987     fc_im      <= fc_fm_1;
988     fc_wt      <= fc_wt_1;
989     fc_state   <= 'MUL_OUT;
990 end
991
992 'MUL_OUT: begin
993     fc_start_bit <= 1'b1;
994     // Clear first set of values after collecting mul_out
995     fc_in1      <= mul_out;
```



```
996         fc_fm_0      <= 0;
997         fc_wt_0      <= 0;
998         fc_state     <= 'FC_ACC;
999     end
1000
1001     'FC_ACC: begin
1002         // Since processing two values at once, divide by 2
1003         if(fc_row_count < fc_wt_row_cnt-1) begin
1004             // Clear second set of values after collecting 2nd
1005                 mul_out
1006             fc_in1     <= mul_out;
1007             fc_fm_1    <= 0;
1008             fc_wt_1    <= 0;
1009             fc_im      <= 0;
1010             fc_wt      <= 0;
1011             // read weights and FM again
1012             fc_wt_adr   <= {wb_adr_wt_ptr, {3'b0}} + fc_row_count
1013                 *fc_wt_col_cnt + fc_col_count;
1014             //fc_row_count <= fc_row_count + 1;
1015             fc_state    <= 'FC_WT0;
1016
1017         end else begin
1018             if(fc_col_count < fc_wt_col_cnt) begin
1019                 // fc_start_bit <= 1'b0;
1020                 fc_in1     <= mul_out;
```

```
1019         fc_fm_1  <= 0;
1020         fc_wt_1  <= 0;
1021         fc_im    <= 0;
1022         fc_wt    <= 0;
1023         //on completing row count -> ReLu
1024         fc_state <= 'ACC_OUT;
1025     end else begin
1026         fc_in1   <= 0;
1027         fc_start_bit <= 1'b0;
1028         fc_done   <= 1;
1029         fc_state <= 'IDLE_FC;
1030     end
1031 end
1032 end
1033
1034 'ACC_OUT: begin
1035     fc_in1      <= 0;
1036     fc_start_bit <= 1'b0;
1037     fc_state    <= 'RELU_OUT;
1038 end
1039
1040 'RELU_OUT: begin
1041     fc_val_in   <= acc_out;
1042     fc_state    <= 'WREQ;
1043 end
```

```
1044
1045     'WREQ: begin
1046         // write back to memory
1047         if (!fc_op_wreq) begin
1048             if (!fc_op_adr[0]) begin
1049                 fc_op_buf[15:0] <= relu_out;
1050                 fc_op_adr <= fc_op_adr + 1;
1051                 fc_row_count <= 0;
1052                 fc_col_count <= fc_col_count + 1;
1053
1054                 // reset FM pointer
1055                 fc_fm_adr <= {wb_adr_fm_ptr, {3'b0}};
1056                 // reset WT pointer
1057                 //fc_wt_adr <= {wb_adr_wt_ptr, {3'b0}};
1058                 //WT adr ptr remains constant
1059                 //read weights and FM again
1060                 fc_op_wreq <= 0;
1061                 fc_sel_0 <= 1;
1062                 if (fc_count == 0) begin
1063                     c_fake <= 0;
1064                     r_fake <= 0;
1065                     fc_state <= 'NEXT_ADDR;
1066                 end else begin
1067                     fc_state <= 'NEXT_WT0_ADDR;
1068                 end
```

```
1069     end else begin
1070         fc_op_buf[31:16] <= relu_out;
1071         fc_sel_1    <= 1;
1072         fc_op_wreq <= 1;
1073     end
1074 end
1075
1076     if (i_m_wb_ack) begin
1077         fc_row_count <= 0;
1078         fc_col_count <= fc_col_count + 1;
1079         fc_op_adr    <= fc_op_adr + 1;
1080         // reset FM pointer
1081         fc_fm_adr    <= {wb_adr_fm_ptr, {3'b0}};
1082         // reset WT pointer
1083         // fc_wt_adr <= {wb_adr_wt_ptr, {3'b0}};
1084         // WT adr ptr remains constant
1085         // read weights and FM again
1086         fc_op_wreq   <= 0;
1087         fc_sel_0     <= 0;
1088         fc_sel_1     <= 0;
1089         if(fc_count == 0) begin
1090             c_fake    <= 0;
1091             r_fake    <= 0;
1092             fc_state  <= 'NEXT_ADDR;
1093         end else begin
```

```
1094         fc_state    <= 'NEXT_WT0_ADDR;
1095     end
1096 end
1097 end
1098
1099     default: begin
1100         fc_state <= 'IDLE_FC;
1101     end
1102
1103 endcase
1104 end
1105 end
1106
1107
1108
1109 //=====
1110 //----- Read Weights -----
1111 //=====
1112
1113 always @(posedge clk or posedge reset_mux)
1114 begin
1115     if(reset_mux) begin
1116         nth_wt    <= 0;
1117         has_wts   <= 0;
1118         rd_wt_adr <= 20'b0;
```

```
1119     wt_rreq     <= 1'b0;
1120
1121     wt_ch0      <= 16'b0;
1122     wt_ch1      <= 16'b0;
1123     wt_ch2      <= 16'b0;
1124     wt_ch3      <= 16'b0;
1125     wt_ch4      <= 16'b0;
1126     wt_ch5      <= 16'b0;
1127     wt_ch6      <= 16'b0;
1128     wt_ch7      <= 16'b0;
1129     wt_ch8      <= 16'b0;
1130
1131     end else begin
1132         if(conv_fc_mode || !valid_wt_base_adr) begin
1133             nth_wt     <= 0;
1134             has_wts     <= 0;
1135             rd_wt_adr   <= 20'b0;
1136             wt_rreq     <= 1'b0;
1137
1138             wt_ch0      <= 16'b0;
1139             wt_ch1      <= 16'b0;
1140             wt_ch2      <= 16'b0;
1141             wt_ch3      <= 16'b0;
1142             wt_ch4      <= 16'b0;
1143             wt_ch5      <= 16'b0;
```

```
1144     wt_ch6     <= 16'b0;
1145     wt_ch7     <= 16'b0;
1146     wt_ch8     <= 16'b0;
1147
1148     end
1149     else if (!has_relu_fm && valid_wt_base_adr) begin
1150     case (nth_wt)
1151
1152         'IDLE_WT: begin
1153             if (valid_wt_base_adr) begin
1154                 if (!has_wts) begin
1155                     // enters here the first time it's reading weights.
1156
1157                     wt_ch0     <= 16'b0;
1158                     wt_ch1     <= 16'b0;
1159                     wt_ch2     <= 16'b0;
1160                     wt_ch3     <= 16'b0;
1161                     wt_ch4     <= 16'b0;
1162                     wt_ch5     <= 16'b0;
1163                     wt_ch6     <= 16'b0;
1164                     wt_ch7     <= 16'b0;
1165                     wt_ch8     <= 16'b0;
1166
1167                     // base address set at the beginning and is not
1168                     reset
```

```
1168         // everytime new set of weights are read.
1169         rd_wt_adr <= {wb_adr_wt_ptr, {3'b0}};
1170         nth_wt     <= 'READ_W0;
1171     end else begin // has weights
1172         if (fm_ch_cmplt && !has_fm_1) begin //&&
1173             acc_mul_cmplt
1174             has_wts <= 1'b0;
1175             nth_wt <= 'READ_W0;
1176         end
1177     end
1178 end
1179
1180 'READ_W0: begin
1181     // address pointer is incremented by two or one every
1182     // time based on number
1183     // of values read in at once (based of parity bit.)
1184     // parity check is done
1185     // in every state unlike in FM read State Machine
1186     // because the values are
1187     // read continuously and even & odd indices can occur in
1188     // any state.
1189     if (i_m_wb_ack) begin
1190         if (!rd_wt_adr[0]) begin
1191             // transition
```



```
1188         wt_ch0  <= i_m_wb_dat[15:0];  //w0
1189         wt_ch1  <= i_m_wb_dat[31:16]; //w1
1190
1191         rd_wt_adr <= rd_wt_adr + 2;
1192         // >= 6 because of +2 in above line , overflow
1193         // can occur only if byte_cnt_wt >= 6
1194         nth_wt    <= 'READ_W2;
1195     end else begin
1196         // transition
1197         wt_ch0    <= i_m_wb_dat[31:16]; //w0
1198         rd_wt_adr <= rd_wt_adr + 1;
1199         nth_wt    <= 'READ_W1;
1200     end
1201 end
1202 end
1203
1204 'READ_W1: begin
1205     if(i_m_wb_ack) begin
1206         if(!rd_wt_adr[0]) begin
1207             // transition
1208             wt_ch1    <= i_m_wb_dat[15:0];  //w1
1209             wt_ch2    <= i_m_wb_dat[31:16]; //w2
1210             rd_wt_adr <= rd_wt_adr + 2;
1211             nth_wt    <= 'READ_W3;
1212         end else begin
```

```
1213         // Transition
1214         wt_ch1    <= i_m_wb_dat[31:16]; //w1
1215         rd_wt_adr <= rd_wt_adr + 1;
1216         nth_wt    <= 'READ_W2;
1217     end
1218 end
1219 end
1220
1221 'READ_W2: begin
1222     if (i_m_wb_ack) begin
1223         if (!rd_wt_adr[0]) begin
1224             // Transition
1225             wt_ch2    <= i_m_wb_dat[15:0]; //w2
1226             wt_ch3    <= i_m_wb_dat[31:16]; //w3
1227             rd_wt_adr <= rd_wt_adr + 2;
1228             nth_wt    <= 'READ_W4;
1229         end else begin
1230             // Transition
1231             wt_ch2    <= i_m_wb_dat[31:16]; //w2
1232             rd_wt_adr <= rd_wt_adr + 1;
1233             nth_wt    <= 'READ_W3;
1234         end
1235     end
1236 end
1237
```

```
1238     'READ_W3: begin
1239         if (i_m_wb_ack) begin
1240             if (!rd_wt_adr[0]) begin
1241                 // Transition
1242                 wt_ch3    <= i_m_wb_dat[15:0]; //w3
1243                 wt_ch4    <= i_m_wb_dat[31:16]; //w4
1244                 rd_wt_adr <= rd_wt_adr + 2;
1245                 nth_wt    <= 'READ_W5;
1246             end else begin
1247                 // Transition
1248                 wt_ch3    <= i_m_wb_dat[31:16]; //w3
1249                 rd_wt_adr <= rd_wt_adr + 1;
1250                 nth_wt    <= 'READ_W4;
1251             end
1252         end
1253     end
1254
1255     'READ_W4: begin
1256         if (i_m_wb_ack) begin
1257             if (!rd_wt_adr[0]) begin
1258                 // Transition
1259                 wt_ch4    <= i_m_wb_dat[15:0]; //w4
1260                 wt_ch5    <= i_m_wb_dat[31:16]; //w5
1261                 rd_wt_adr <= rd_wt_adr + 2;
1262                 nth_wt    <= 'READ_W6;
```

```
1263         end else begin
1264             // Transition
1265             wt_ch4    <= i_m_wb_dat[31:16]; //w4
1266             rd_wt_adr <= rd_wt_adr + 1;
1267             nth_wt    <= 'READ_W5;
1268         end
1269     end
1270 end
1271
1272 'READ_W5: begin
1273     if(i_m_wb_ack) begin
1274         if(!rd_wt_adr[0]) begin
1275             // Transition
1276             wt_ch5    <= i_m_wb_dat[15:0]; //w5
1277             wt_ch6    <= i_m_wb_dat[31:16]; //w6
1278             rd_wt_adr <= rd_wt_adr + 2;
1279             nth_wt    <= 'READ_W7;
1280         end else begin
1281             // Transition
1282             wt_ch5    <= i_m_wb_dat[31:16]; //w5
1283             rd_wt_adr <= rd_wt_adr + 1;
1284             nth_wt    <= 'READ_W6;
1285         end
1286     end
1287 end
```

```
1288
1289     'READ_W6: begin
1290         if (i_m_wb_ack) begin
1291             if (!rd_wt_adr[0]) begin
1292                 // Transition
1293                 wt_ch6    <= i_m_wb_dat[15:0]; //w6
1294                 wt_ch7    <= i_m_wb_dat[31:16]; //w7
1295                 rd_wt_adr <= rd_wt_adr + 2;
1296                 nth_wt    <= 'READ_W8;
1297             end else begin
1298                 // Transition
1299                 wt_ch6    <= i_m_wb_dat[31:16]; //w6
1300                 rd_wt_adr <= rd_wt_adr + 1;
1301                 nth_wt    <= 'READ_W5;
1302             end
1303         end
1304     end
1305
1306     'READ_W7: begin
1307         if (i_m_wb_ack) begin
1308             if (!rd_wt_adr[0]) begin
1309                 if (!has_wts) begin
1310                     // Transition
1311                     wt_ch7    <= i_m_wb_dat[15:0]; //w7
1312                     wt_ch8    <= i_m_wb_dat[31:16]; //w8
```

```
1313         rd_wt_adr <= rd_wt_adr + 2;
1314         has_wts    <= 1'b1;
1315         nth_wt     <= 'IDLE_WT;
1316         // wt_rreq <= 1'b0;
1317     end
1318 end else begin
1319     // Transition
1320     wt_ch7    <= i_m_wb_dat[31:16]; //w7
1321     rd_wt_adr <= rd_wt_adr + 1;
1322     nth_wt    <= 'READ_W8;
1323 end
1324 //nth_wt <= 'IDLE_WT;
1325 // if state machine needs to idle around go to Idle
        state only.
1326 end
1327 end
1328
1329 'READ_W8: begin
1330     if(i_m_wb_ack) begin
1331         //turning has wts on one cycle before
1332         //to reduce one clock cycle latency.
1333         has_wts <= 1'b1;
1334     if(!rd_wt_adr[0]) begin
1335         if(!has_wts) begin
1336             // Transition
```

```
1337         wt_ch8    <= i_m_wb_dat[15:0];    //w8
1338         rd_wt_adr <= rd_wt_adr + 1;
1339     end
1340 end else begin
1341     if(!has_wts) begin
1342         // Transition
1343         wt_ch8    <= i_m_wb_dat[31:16]; //w8
1344         rd_wt_adr <= rd_wt_adr + 1;
1345     end
1346 end
1347     nth_wt <= 'IDLE_WT;
1348 end
1349 end
1350
1351 default: begin
1352     nth_wt <= 'IDLE_WT;
1353 end
1354 endcase
1355 end
1356 end
1357 end //read weights
1358
1359 //=====
1360 //----- Read Feature maps -----
1361 //=====
```

```
1362 always @(posedge clk or posedge reset_mux)
1363 begin
1364     if (reset_mux ) begin
1365         nth_fm          <= 4'b0;
1366         row_cnt         <= 6'b0;
1367         col_cnt         <= 6'b0;
1368         rd_fm_adr       <= 14'b0;
1369         r1_fm_ptr       <= 14'b0;
1370         r2_fm_ptr       <= 14'b0;
1371         r3_fm_ptr       <= 14'b0;
1372         conv_ch_count   <= 6'b0;
1373         fm_rreq         <= 1'b0;
1374         has_fm_l        <= 1'b0;
1375         fm_ch_cmplt     <= 1'b0;
1376         fm_cmplt        <= 1'b0;
1377         all_fm_cmplt    <= 1'b0;
1378         conv_fil_count  <= 6'b0;
1379         fm_ch0          <= 16'b0;
1380         fm_ch1          <= 16'b0;
1381         fm_ch2          <= 16'b0;
1382         fm_ch3          <= 16'b0;
1383         fm_ch4          <= 16'b0;
1384         fm_ch5          <= 16'b0;
1385         fm_ch6          <= 16'b0;
1386         fm_ch7          <= 16'b0;
```



```
1387     fm_ch8           <= 16'b0;
1388 end else begin
1389     if(conv_fc_mode || !valid_fm_base_adr) begin
1390         nth_fm         <= 4'b0;
1391         row_cnt        <= 6'b0;
1392         col_cnt        <= 6'b0;
1393         rd_fm_adr      <= 14'b0;
1394         r1_fm_ptr      <= 14'b0;
1395         r2_fm_ptr      <= 14'b0;
1396         r3_fm_ptr      <= 14'b0;
1397         conv_ch_count  <= 6'b0;
1398         fm_rreq        <= 1'b0;
1399         has_fm_1       <= 1'b0;
1400         fm_ch_cmplt    <= 1'b0;
1401         fm_cmplt       <= 1'b0;
1402         all_fm_cmplt   <= 1'b0;
1403         conv_fil_count <= 6'b0;
1404         fm_ch0         <= 16'b0;
1405         fm_ch1         <= 16'b0;
1406         fm_ch2         <= 16'b0;
1407         fm_ch3         <= 16'b0;
1408         fm_ch4         <= 16'b0;
1409         fm_ch5         <= 16'b0;
1410         fm_ch6         <= 16'b0;
1411         fm_ch7         <= 16'b0;
```

```
1412     fm_ch8           <= 16'b0;
1413
1414     end
1415     else if(!conv_fc_mode && valid_fm_base_adr) begin
1416     case(nth_fm)
1417         'IDLE_FM: begin
1418             if(mp_fil_count != conv_num_fil) begin
1419                 if(valid_fm_base_adr) begin //&& !has_relu_fm
1420
1421                     //should read FM values only when it has valid
1422                     weights and
1423                     // it doesn't have new FM values
1424                     if (has_wts && !has_fm_1) begin
1425                         rd_fm_adr     <= {wb_adr_fm_ptr , {3'b0}} ;
1426                         r1_fm_ptr     <= {wb_adr_fm_ptr , {3'b0}} ;
1427                         r2_fm_ptr     <= {wb_adr_fm_ptr , {3'b0}} +
1428                                     conv_ip_size_r ;
1429                         r3_fm_ptr     <= {wb_adr_fm_ptr , {3'b0}} +
1430                                     conv_ip_size_2r;
1431                         has_fm_1     <= 1'b0;
1432                         fm_ch_cmplt  <= 0;
1433                         fm_cmplt    <= 0;
1434                         all_fm_cmplt <= 0; //later change.
1435                         nth_fm       <= 'READ_FM0;
1436                     end
1437                 end
1438             end
1439         end
1440     end
1441 end
```

```
1433     end
1434     end
1435 end
1436 'READ_FM0: begin
1437     if (fm_ch_cmplt)
1438         fm_ch_cmplt <= 0;
1439     else begin
1440         if (has_wts) begin
1441             if (!has_relu_fm) begin
1442                 if (i_m_wb_ack) begin
1443                     if (!rd_fm_adr[0]) begin
1444                         // Transition
1445                         fm_ch0    <= i_m_wb_dat[15:0]; //FM0
1446                         fm_ch1    <= i_m_wb_dat[31:16]; //FM1
1447                         rd_fm_adr <= rd_fm_adr +2;
1448                         nth_fm    <= 'READ_FM2;
1449                     end else begin
1450                         // Transition
1451                         fm_ch0    <= i_m_wb_dat[31:16]; //FM0
1452                         rd_fm_adr <= rd_fm_adr + 1;
1453                         nth_fm    <= 'READ_FM1;
1454                     end
1455                 end
1456             end
1457         end
```

```
1458     end
1459 end
1460 'READ_FM1: begin
1461     if (i_m_wb_ack) begin
1462         if (!rd_fm_adr[0]) begin
1463             // Transition
1464             fm_ch1  <= i_m_wb_dat[15:0]; //FM1
1465             fm_ch2  <= i_m_wb_dat[31:16]; //FM2
1466
1467             //comes to this state ONLY if ONE value was read
1468             //in previous state. since three values will read
1469             //from one row by the end of this state the address
1470             //pointer should point to next row for the next
1471             read.
1472             rd_fm_adr <= r2_fm_ptr;
1473             nth_fm    <= 'READ_FM3;
1474         end
1475     end
1476 'READ_FM2: begin
1477     if (i_m_wb_ack) begin
1478         //rd_fm_adr[0] should be 0 even
1479         //if in prev state it was even index,
1480         //you moved two indices at once, even
1481         //num+2 is even, so read the even
```

```
1482         //indexed value from 32 bit input
1483         // data.
1484         if (!rd_fm_adr[0]) begin
1485             // Transition
1486             fm_ch2    <= i_m_wb_dat[15:0];    //FM2
1487             rd_fm_adr <= r2_fm_ptr; //Address pointer for next
                value to be read from row2
1488             nth_fm    <= 'READ_FM3;
1489             //can be implemneted in 3 states but requires more
                counters and complicates ->
1490             //hence implemented in 9 states.
1491         end
1492     end
1493 end
1494 'READ_FM3: begin
1495     if (i_m_wb_ack) begin
1496         if (!rd_fm_adr[0]) begin
1497             // Transition
1498             fm_ch3    <= i_m_wb_dat[15:0];    //FM3
1499             fm_ch4    <= i_m_wb_dat[31:16];    //FM4
1500             rd_fm_adr <= rd_fm_adr +2;
1501             nth_fm    <= 'READ_FM5;
1502         end else begin
1503             // Transition
1504             fm_ch3    <= i_m_wb_dat[31:16];    //FM3
```

```
1505         rd_fm_adr <= rd_fm_adr + 1;
1506         nth_fm     <= 'READ_FM4;
1507     end
1508 end
1509 end
1510 'READ_FM4: begin
1511     if (i_m_wb_ack) begin
1512         if (!rd_fm_adr[0]) begin
1513             // Transition
1514             fm_ch4 <= i_m_wb_dat[15:0]; //FM4
1515             fm_ch5 <= i_m_wb_dat[31:16]; //FM5
1516             //comes to this state ONLY if ONE value was read
1517             //in previous state. since three values will read
1518             //from one row by the end of this state the address
1519             //pointer should point to next row for the next
1520             read.
1521             rd_fm_adr <= r3_fm_ptr;
1522             nth_fm     <= 'READ_FM6;
1523         end
1524     end
1525 'READ_FM5: begin
1526     if (i_m_wb_ack) begin
1527         if (!rd_fm_adr[0]) begin
1528             // Transition
```

```
1529         fm_ch5     <= i_m_wb_dat[15:0];    //FM5
1530         rd_fm_adr <= r3_fm_ptr; //Address pointer for next
           value to be read from row2
1531         nth_fm     <= 'READ_FM6;
1532         //can be implemneted in 3 states but requires more
           counters and complicates ->
1533         //hence implemented in 9 states.
1534     end
1535 end
1536 end
1537 'READ_FM6: begin
1538     if (i_m_wb_ack) begin
1539         if (!rd_fm_adr[0]) begin
1540             // Transition
1541             fm_ch6     <= i_m_wb_dat[15:0];    //FM6
1542             fm_ch7     <= i_m_wb_dat[31:16];    //FM7
1543             rd_fm_adr <= rd_fm_adr +2;
1544             nth_fm     <= 'READ_FM8;
1545         end else begin
1546             // Transition
1547             fm_ch6     <= i_m_wb_dat[31:16];    //FM6
1548             rd_fm_adr <= rd_fm_adr + 1;
1549             nth_fm     <= 'READ_FM7;
1550         end
1551     end
```

```
1552     end
1553     'READ_FM7: begin
1554         if (i_m_wb_ack) begin
1555             if (!rd_fm_adr[0]) begin
1556                 // Transition
1557                 fm_ch7   <= i_m_wb_dat[15:0]; //FM7
1558                 fm_ch8   <= i_m_wb_dat[31:16]; //FM8
1559                 has_fm_1 <= 1'b1;
1560
1561                 //comes to this state ONLY if ONE value was read
1562                 //in previous state. since three values will read
1563                 //from one row by the end of this state the address
1564                 //pointer should point to next row for the next
1565                 read.
1566
1567                 r3_fm_ptr <= r3_fm_ptr + conv_ip_size_r;
1568                 //row count and col count: IPSIZE -2-1 -> -2
1569                 convoltion
1570                 //and 1 cuz one iteration is complete before
1571                 reaching this point.
1572                 nth_fm     <= 'START_ACCMUL;
1573             end
1574         end
1575     end
1576     'READ_FM8: begin
```



```
1574     if (i_m_wb_ack) begin
1575         if (!rd_fm_adr[0]) begin
1576             // Transition
1577             fm_ch8     <= i_m_wb_dat[15:0];    //FM8
1578             has_fm_1   <= 1'b1;
1579             r3_fm_ptr  <= r3_fm_ptr + conv_ip_size_r;
1580             nth_fm     <= 'START_ACCMUL;
1581             //can be implemented in 3 states but requires
1582             //more counters and complicates ->
1583             //hence implemented in 9 states.
1584         end
1585     end
1586 end
1587 'START_ACCMUL: begin
1588     if(acc_mul_cmplt) begin
1589         if (row_cnt < conv_op_size_r-1) begin
1590             has_fm_1 <= 0;
1591             row_cnt  <= row_cnt + 1;
1592             rd_fm_adr <= r3_fm_ptr; //Address pointer for next
1593                                     value to be read from row2
1594             // Transition
1595             fm_ch0    <= fm_ch3;
1596             fm_ch1    <= fm_ch4;
1597             fm_ch2    <= fm_ch5;
```

```
1598         fm_ch3     <= fm_ch6;
1599         fm_ch4     <= fm_ch7;
1600         fm_ch5     <= fm_ch8;
1601
1602         nth_fm      <= 'READ_FM6;
1603
1604     end else begin
1605         if (col_cnt < conv_op_size_c - 1) begin
1606             row_cnt <= 0;
1607             has_fm_1 <= 0;
1608             col_cnt <= col_cnt + 1;
1609
1610             //REV 3 -> transition from params to reg vals
1611             rd_fm_adr <= {wb_adr_fm_ptr, {3'b0}} +
                conv_ch_count * conv_ip_size_rc + col_cnt + 1;
1612             r1_fm_ptr <= {wb_adr_fm_ptr, {3'b0}} +
                conv_ch_count * conv_ip_size_rc + col_cnt + 1;
1613             r2_fm_ptr <= {wb_adr_fm_ptr, {3'b0}} +
                conv_ch_count * conv_ip_size_rc +
                conv_ip_size_r + col_cnt + 1;
1614             r3_fm_ptr <= {wb_adr_fm_ptr, {3'b0}} +
                conv_ch_count * conv_ip_size_rc +
                conv_ip_size_2r + col_cnt + 1;
1615
1616             nth_fm    <= 'READ_FM0;
```

```
1617         end else begin
1618             if (conv_ch_count < conv_num_ch-1) begin
1619                 col_cnt         <= 0;
1620                 row_cnt         <= 0;
1621                 has_fm_1        <= 0;
1622                 conv_ch_count <= conv_ch_count +1;
1623                 fm_ch_cmplt     <= 1'b1;
1624
1625                 //REV 3 -> transition from params to reg vals
1626                 rd_fm_adr       <= {wb_adr_fm_ptr, {3'b0}} + (
1627                     conv_ch_count+1) * conv_ip_size_rc;
1628                 r1_fm_ptr       <= {wb_adr_fm_ptr, {3'b0}} + (
1629                     conv_ch_count+1) * conv_ip_size_rc;
1630                 r2_fm_ptr       <= {wb_adr_fm_ptr, {3'b0}} + (
1631                     conv_ch_count+1) * conv_ip_size_rc +
1632                     conv_ip_size_r;
1633                 r3_fm_ptr       <= {wb_adr_fm_ptr, {3'b0}} + (
1634                     conv_ch_count+1) * conv_ip_size_rc +
1635                     conv_ip_size_2r;
1636
1637                 nth_fm          <= 'READ_FM0;
1638
1639             end else begin
1640                 if (conv_fil_count < conv_num_fil-1) begin
1641                     col_cnt         <= 0;
```

```
1636         row_cnt         <= 0;
1637         conv_ch_count    <= 0;
1638         has_fm_1         <= 0;
1639         fm_ch_cmplt      <= 1'b1;
1640         fm_cmplt         <= 1'b1;
1641         conv_fil_count   <= conv_fil_count + 1;
1642         nth_fm           <= 'IDLE_FM;
1643     end else begin
1644         col_cnt         <= 0;
1645         row_cnt         <= 0;
1646         has_fm_1         <= 0;
1647         conv_ch_count    <= 0;
1648         fm_ch_cmplt      <= 1'b1;
1649         fm_cmplt         <= 1'b1;
1650         conv_fil_count   <= 0;
1651         all_fm_cmplt     <= 1'b1;
1652         nth_fm           <= 'IDLE_FM;
1653     end
1654 end
1655 end
1656 end
1657 end
1658 end
1659 default: begin
1660     nth_fm <= 'IDLE_FM;
```

```
1661         end
1662     endcase
1663 end
1664 end
1665 end //read feature maps
1666
1667
1668 //=====
1669 //----- ACC-MUL State Machine -----
1670 //=====
1671
1672 always @(posedge clk or posedge reset_mux)
1673 begin
1674     if(reset_mux) begin
1675         im_conv         <= 0;
1676         wt_conv         <= 0;
1677         in1_conv        <= 0;
1678         mul_cnt         <= 0;
1679         start_bit_conv  <= 0;
1680         acc_mul_cmplt   <= 0;
1681         val_in_conv     <= 0;
1682         has_relu_fm     <= 0;
1683         curr_row        <= 0;
1684         curr_col        <= 0;
1685         mp_row          <= 0;
```

```
1686     mp_row_1p      <= 0;
1687     mp_col         <= 0;
1688     mp_col_1p     <= 0;
1689     op_wreq       <= 0;
1690     mp_cmplt      <= 0;
1691     add_in1       <= 0;
1692     add_in2       <= 0;
1693     curr_ch       <= 0;
1694     mp1_val_1     <= 0;
1695     mp1_val_2     <= 0;
1696     mp1_val_3     <= 0;
1697     mp1_val_4     <= 0;
1698     rd_op_adr     <= 0;
1699     selx_0        <= 0;
1700     selx_1        <= 0;
1701     write_buf     <= 0;
1702     mp_fil_count  <= 0;
1703     mp_iterator_c <= 0;
1704     mp_iterator_r <= 0;
1705
1706     for (j=0; j < MAX_OP_SIZE_R; j=j+1) begin
1707         for(k=0; k< MAX_OP_SIZE_R; k=k+1) begin
1708             conv_op[j][k] <= 0;
1709         end
1710     end
```

```
1711     end else begin
1712         if(conv_fc_mode || !valid_op_base_adr) begin
1713             im_conv          <= 0;
1714             wt_conv          <= 0;
1715             in1_conv         <= 0;
1716             mp_row           <= 0;
1717             mp_col           <= 0;
1718             mp_row_1p        <= 0;
1719             mp_col_1p        <= 0;
1720             start_bit_conv   <= 0;
1721             selx_0           <= 0;
1722             selx_1           <= 0;
1723             write_buf        <= 0;
1724             mp_cmplt         <= 0;
1725             mp_fil_count     <= 0;
1726             rd_op_adr        <= 0;
1727             mp_iterator_c    <= 0;
1728             mp_iterator_r    <= 0;
1729
1730             for (j=0; j < MAX_OP_SIZE_R; j=j+1) begin
1731                 for(k=0; k< MAX_OP_SIZE_R; k=k+1) begin
1732                     conv_op[j][k] <= 0;
1733                 end
1734             end
1735         end
```

```
1736     else if(valid_op_base_adr) begin
1737     case(mul_cnt)
1738         'ACCMUL_IDLE: begin
1739             im_conv      <= 0;
1740             wt_conv      <= 0;
1741             in1_conv     <= 0;
1742             mp_row       <= 0;
1743             mp_col       <= 0;
1744             mp_row_1p    <= 0;
1745             mp_col_1p    <= 0;
1746             start_bit_conv <= 0;
1747             selx_0       <= 0;
1748             selx_1       <= 0;
1749             write_buf    <= 0;
1750             mp_cmplt     <= 0;
1751             mp_iterator_c <= 0;
1752             mp_iterator_r <= 0;
1753
1754             //TODO: when to reset conv_op[][]?
1755             //reset conv_op after every filter operation.
1756             if((mp_fil_count == conv_num_fil)&& !valid_wt_base_adr)
1757                 begin
1758                     mp_fil_count <= 0;
1759                     for (j=0; j < MAX_OP_SIZE_R; j=j+1) begin
1760                         for(k=0; k< MAX_OP_SIZE_R; k=k+1) begin
```



```
1760             conv_op[j][k] <= 0;
1761         end
1762     end
1763 end
1764 if (valid_op_base_adr && (mp_fil_count < conv_num_fil))
1765     begin
1766         if (!valid_fm_base_adr && !valid_wt_base_adr)
1767             rd_op_adr      <= {wb_adr_op_ptr, {3'b0}}; // +
1768                 fil_num *OP_SIZE_SQ;
1769         else if (has_wts && has_fm_1) begin
1770
1771             im_conv      <= fm_ch0;
1772             wt_conv      <= wt_ch0;
1773             acc_mul_cmplt <= 1'b0;
1774             mul_cnt      <= 'MUL1;
1775         end
1776     end
1777 end
1778
1779 'MUL1: begin
1780     wt_conv  <= wt_ch1;
1781     im_conv  <= fm_ch1;
1782     start_bit_conv <= 1'b1;
1783     in1_conv  <= mul_out;
1784     mul_cnt  <= 'MUL2;
```

```
1782     end
1783
1784     'MUL2: begin
1785         in1_conv <= mul_out;
1786         wt_conv  <= wt_ch2;
1787         im_conv  <= fm_ch2;
1788         mul_cnt  <= 'MUL3;
1789     end
1790
1791     'MUL3: begin
1792         in1_conv <= mul_out;
1793         im_conv  <= fm_ch3;
1794         wt_conv  <= wt_ch3;
1795         mul_cnt  <= 'MUL4;
1796     end
1797
1798     'MUL4: begin
1799         in1_conv <= mul_out;
1800         im_conv  <= fm_ch4;
1801         wt_conv  <= wt_ch4;
1802         mul_cnt  <= 'MUL5;
1803     end
1804
1805     'MUL5: begin
1806         in1_conv <= mul_out;
```

```
1807         im_conv  <= fm_ch5;
1808         wt_conv  <= wt_ch5;
1809         mul_cnt  <= 'MUL6;
1810     end
1811
1812     'MUL6: begin
1813         inl_conv <= mul_out;
1814         im_conv  <= fm_ch6;
1815         wt_conv  <= wt_ch6;
1816         mul_cnt  <= 'MUL7;
1817     end
1818
1819     'MUL7: begin
1820         inl_conv <= mul_out;
1821         im_conv  <= fm_ch7;
1822         wt_conv  <= wt_ch7;
1823         mul_cnt  <= 'MUL8;
1824     end
1825
1826     'MUL8: begin
1827         inl_conv <= mul_out;
1828         im_conv  <= fm_ch8;
1829         wt_conv  <= wt_ch8;
1830         mul_cnt  <= 'ACC7;
1831     end
```

```
1832
1833     'ACC7: begin
1834         in1_conv <= mul_out;
1835         mul_cnt <= 'ACC8;
1836     end
1837
1838     'ACC8: begin
1839         in1_conv <= mul_out;
1840         mul_cnt <= 'CMPLT;
1841     end
1842
1843     'CMPLT: begin
1844         start_bit_conv <= 1'b0;
1845         acc_mul_cmplt <= 1'b1;
1846         if(acc_mul_cmplt && has_fm_1) begin
1847             //accumulating all channel convolved results
1848             //bring in your adder
1849             //acc_out is 23 bits -> add_in1 is 23
1850             add_in1 <= acc_out;
1851             //conv_op is 24 bits , add_in2 is 24 bits
1852             add_in2 <= conv_op[row_cnt][col_cnt];
1853             //collect the current r, c, ch values as they change
1854             //in FM read state machine next clock cycle.
1855             curr_row <= row_cnt;
1856             curr_col <= col_cnt;
```

```
1857         curr_ch  <= conv_ch_count;
1858         mul_cnt  <= 'ADD;
1859     end
1860 end
1861
1862 'ADD: begin
1863     if (!has_fm_1) begin
1864         if (curr_ch == conv_num_ch-1) begin
1865             //determining if all input channels are
1866             //convolved and accumulated.
1867             if (fm_ch_cmplt && fm_cmplt) begin
1868                 has_relu_fm <= 1;
1869             end
1870         end
1871     end
1872     mul_cnt <= 'ADD1;
1873 end
1874
1875 'ADD1: begin
1876     if (curr_ch == conv_num_ch-1) begin
1877         conv_op[curr_row][curr_col] <= add_out;
1878         acc_mul_cmplt <= 1'b0;
1879         mul_cnt <= 'RELU;
1880     end else begin
1881         conv_op[curr_row][curr_col] <= add_out;
```

```
1882         acc_mul_cmplt    <= 1'b0;
1883         mul_cnt           <= 'ACCMUL_IDLE;
1884     end
1885 end
1886
1887 'RELU: begin
1888     val_in_conv <= conv_op[curr_row][curr_col];
1889     mul_cnt     <= 'COLLECT_RELU;
1890 end
1891
1892 'COLLECT_RELU: begin
1893
1894     //figure the math for number of bits for
1895     //concatenating 0's before assignment of relu out
1896     conv_op[curr_row][curr_col] <= relu_out;
1897
1898     //what's the next state?
1899     if(has_relu_fm && !mp_flag) begin
1900         mul_cnt <= 'WRITE_RELU;
1901     end else if(has_relu_fm && mp_flag) begin
1902         mp_row_1p <= mp_row + 1;
1903         mp_col_1p <= mp_col + 1;
1904
1905         if(conv_op_size_c[0]) begin
1906             mp_iterator_c <= conv_op_size_c - 3;
```

```
1907         end else begin
1908             mp_iterator_c <= conv_op_size_c - 2;
1909         end
1910
1911         if(conv_op_size_r[0]) begin
1912             mp_iterator_r <= conv_op_size_r - 3;
1913         end else begin
1914             mp_iterator_r <= conv_op_size_r - 2;
1915         end
1916
1917         mul_cnt <= 'MAXPOOL_1;
1918     end else begin
1919         acc_mul_cmplt <= 1'b0;
1920         mul_cnt      <= 'ACCMUL_IDLE;
1921     end
1922 end
1923
1924 'WRITE_RELU: begin
1925     // Write ReLu values to memory
1926     if (!mp_cmplt && !op_wreq) begin
1927         // if(!op_wreq) begin
1928         //rd_op_adr is set to base adr initially
1929         //Write data to buffer
1930         if(!rd_op_adr[0]) begin
1931             //even number indexed value resides in
```

```
1932         //lower significant bits of 32 bit block in MM
1933         selx_0      <= 1'b1;    //4'b0011;
1934         write_buf[15:0] <= conv_op[mp_row][mp_col];
1935         if ((mp_col < conv_op_size_c-1)) begin
1936             //if the current value is NOT the LAST value then
1937             //wait for next 16 bits to be filled before
1938                 raising wreq.
1939             mp_col      <= mp_col + 1;
1940             rd_op_adr <= rd_op_adr + 1; //16 bit access
1941                 address ptr.
1942         end else begin
1943             if(mp_row < conv_op_size_r-1) begin
1944                 mp_col      <= 0;
1945                 mp_row      <= mp_row + 1;
1946                 rd_op_adr <= rd_op_adr + 1; //16 bit access
1947                     address ptr.
1948             end else begin
1949                 //if the current value is the LAST value then
1950                 //raise the wreq.
1951                 write_buf[15:0] <= conv_op[mp_row][mp_col];
1952                 op_wreq          <= 1'b1;
1953             end
1954         end
1955     end else begin
```



```
1953         //odd number indexed value resides in MSB 16 bits
           of 32 bit block in MM
1954         //once MSB is written it means the buffer is full
           for current mem access or
1955         //the even indexed value was previously written.
1956         //In either case raise wreq.
1957         selx_1      <= 1'b1; //4'b1100;
1958         write_buf[31:16] <= conv_op[mp_row][mp_col];
1959         op_wreq     <= 1'b1;
1960     end
1961 end else if(!mp_cmplt && op_wreq) begin
1962     if(i_m_wb_ack) begin
1963         op_wreq     <= 1'b0;
1964         selx_0      <= 0;
1965         selx_1      <= 0;
1966         //enter here if the write is serviced ,
1967         //and set index & address pointers for next value.
1968         rd_op_adr  <= rd_op_adr + 1; //16 bit access
           address ptr.
1969         //OP_SIZE2M -> rotates half number of times as it
           is reading four val's at once
1970         if(mp_col < conv_op_size_c-1) begin
1971             mp_col   <= mp_col + 1;
1972         end else begin
1973             mp_col   <= 0;
```

```
1974         if(mp_row < conv_op_size_r-1) begin //OP_SIZE2M
1975             mp_row             <= mp_row + 1;
1976         end else begin
1977             mp_row             <= 0;
1978             mp_row_1p         <= 0;
1979             mp_cmplt         <= 1;
1980             mp_fil_count     <= mp_fil_count + 1;
1981             for (j=0; j < MAX_OP_SIZE_R; j=j+1) begin
1982                 for(k=0; k< MAX_OP_SIZE_R; k=k+1) begin
1983                     conv_op[j][k] <= 0;
1984                 end
1985             end
1986             has_relu_fm     <= 0;
1987             mul_cnt         <= 'ACCMUL_IDLE;
1988         end
1989     end
1990 end
1991 end
1992 end
1993
1994 'MAXPOOL_1: begin
1995     mp1_val_1 <= conv_op[mp_row][mp_col];           // r0 c0
1996     mul_cnt <= 'MAXPOOL_2;
1997 end
1998
```



```
2024      //lower significant bits of 32 bit block in MM
2025      selx_0      <= 1'b1;  //4'b0011;
2026      write_buf[15:0]  <= mp1_out_1;
2027      // if ((mp_col < conv_op_size_c -2)) begin
2028      if ((mp_col < mp_iterator_c)) begin
2029          //if the current value is NOT the LAST value then
2030          //wait for next 16 bits to be filled before
                raising wreq.
2031      mp_col      <= mp_col + 2;
2032      mp_col_1p   <= mp_col_1p +2;
2033      rd_op_adr   <= rd_op_adr + 1; //16 bit access
                address ptr.
2034      mul_cnt     <= 'MAXPOOL_1;
2035      end else begin
2036          if(mp_row < mp_iterator_r) begin
2037              mp_col      <= 0;
2038              mp_col_1p   <= 1;
2039              mp_row      <= mp_row + 2;
2040              mp_row_1p   <= mp_row_1p + 2;
2041              rd_op_adr   <= rd_op_adr + 1; //16 bit access
                address ptr.
2042              mul_cnt     <= 'MAXPOOL_1;
2043          end else begin
2044              //if the current value is the LAST value then
2045              //raise the wreq.
```

```
2046         write_buf[15:0]  <= mp1_out_1;
2047         op_wreq           <= 1'b1;
2048     end
2049 end
2050 end else begin
2051     //odd number indexed value resides in MSB 16 bits
2052     //of 32 bit block in MM
2053     //once MSB is written it means the buffer is full
2054     //for current mem access or
2055     //the even indexed value was previously written.
2056     //In either case raise wreq.
2057     selx_1      <= 1'b1; //4'b1100;
2058     write_buf[31:16] <= mp1_out_1;
2059     op_wreq     <= 1'b1;
2060 end
2061 end else if (!mp_cmplt && op_wreq) begin
2062     if (i_m_wb_ack) begin
2063         op_wreq    <= 1'b0;
2064         selx_0     <= 0;
2065         selx_1     <= 0;
2066         //enter here if the write is serviced ,
2067         //and set index & address pointers for next value.
2068         rd_op_adr <= rd_op_adr + 1; //16 bit access
2069         address ptr.
```

```
2067         //OP_SIZE2M -> rotates half number of times as it
           is reading four val's at once
2068     if(mp_col < mp_iterator_c) begin
2069         mp_col     <= mp_col + 2;
2070         mp_col_1p <= mp_col_1p + 2;
2071         mul_cnt   <= 'MAXPOOL_1;
2072     end else begin
2073         mp_col     <= 0;
2074         mp_col_1p <= 1;
2075         if(mp_row < mp_iterator_r) begin //OP_SIZE2M
2076             mp_row     <= mp_row + 2;
2077             mp_row_1p  <= mp_row_1p + 2;
2078             mul_cnt    <= 'MAXPOOL_1;
2079         end else begin
2080             mp_row     <= 0;
2081             mp_row_1p  <= 0;
2082             mp_cmplt   <= 1;
2083             mp_fil_count <= mp_fil_count + 1;
2084             //Reset conv_op after every filter operation.
2085             for (j=0; j < MAX_OP_SIZE_R; j=j+1) begin
2086                 for(k=0; k< MAX_OP_SIZE_R; k=k+1) begin
2087                     conv_op[j][k] <= 0;
2088                 end
2089             end
2090             has_relu_fm <= 0;
```

```
2091             mul_cnt      <= 'ACCMUL_IDLE;
2092         end
2093     end
2094 end
2095 end
2096 end
2097
2098     default:
2099         mul_cnt <= 'ACCMUL_IDLE;
2100     endcase
2101 end
2102 end
2103 end //ACC MUL
2104
2105
2106 //
2107 // *****
2108 //           Wishbone Interface
2109 // *****
2110
2111 assign start_write = i_s_wb_stb && i_s_wb_we && !start_read_1;
2112 assign start_read  = i_s_wb_stb && !i_s_wb_we && !o_s_wb_ack;
```

```
2112 always @( posedge reset_mux or posedge clk )
2113     if (reset_mux)
2114         start_read_1 <= 1'b0;
2115     else
2116         start_read_1 <= start_read;
2117
2118 assign o_s_wb_err = 1'd0;
2119 assign o_s_wb_ack = i_s_wb_stb && ( start_write || start_read_1
    );
2120
2121 generate
2122 if (WB_DWIDTH == 128)
2123     begin : wb128
2124         assign wb_wdata32 = i_s_wb_adr[3:2] == 2'd3 ? i_s_wb_dat
    [127:96] :
2125
    i_s_wb_adr[3:2] == 2'd2 ? i_s_wb_dat[
    95:64] :
2126
    i_s_wb_adr[3:2] == 2'd1 ? i_s_wb_dat[
    63:32] :
2127
    i_s_wb_dat[
    31: 0] ;
2128
2129     assign o_s_wb_dat = {4{ wb_rdata32 }};
2130     end
2131 else
```



```
2132     begin : wb32
2133     assign wb_wdata32 = i_s_wb_dat;
2134     assign o_s_wb_dat = wb_rdata32;
2135     end
2136 endgenerate
2137
2138
2139 //
                *****
2140 //     Register writes
2141 //
                *****
2142
2143 always @(posedge clk or posedge reset_mux)
2144 begin
2145     if (reset_mux == 1'b1) begin
2146         sw_reset <= 1'b0;
2147         wb_adr_cnfg_ptr    <= 11'b0;
2148         wb_adr_cnfg_base_ptr <= 13'b0;
2149         start_bit    <= 1'b0;
2150     end else begin
2151         if ( start_write ) begin
2152             case ( i_s_wb_adr[15:0] )
```

```
2153     ICNN_SRESET:      sw_reset                <=
                        wb_wdata32 [0];        // 16'h0000
2154     ICNN_LAYER_CONFIG: { start_bit , wb_adr_cnfg_base_ptr ,
                        wb_adr_cnfg_ptr } <= wb_wdata32 [24:0]; // , mp_flag ,
                        conv_fc_mode ,
2155         endcase
2156     end
2157 end
2158 end
2159
2160
2161 //
                *****
2162 //     Register reads
2163 //
                *****
2164
2165 always @(posedge clk or posedge reset_mux)
2166 begin
2167     if (reset_mux == 1'b1) begin
2168         wb_rdata32 <= 32'h00C0FFEE;
2169     end else begin
2170         if ( start_read ) begin
```

```
2171     case ( i_s_wb_adr[15:0] )
2172         ICNN_SRESET:      wb_rdata32 <= {31'b0, sw_reset };
2173         ICNN_WT_BASE_ADR: wb_rdata32 <= {8'b0,
                wb_adr_wt_base_ptr };
2174         ICNN_FM_BASE_ADR: wb_rdata32 <= {8'b0,
                wb_adr_fm_base_ptr };
2175         ICNN_OP_BASE_ADR: wb_rdata32 <= {8'b0,
                wb_adr_fm_base_ptr };
2176         ICNN_WT_VALADR   : wb_rdata32 <= {31'b0,
                valid_wt_base_adr };
2177         ICNN_FM_VALADR   : wb_rdata32 <= {31'b0,
                valid_fm_base_adr };
2178         ICNN_OP_VALADR   : wb_rdata32 <= {31'b0,
                valid_op_base_adr };
2179         ICNN_WT_CUR_PTR   : wb_rdata32 <= {19'b0, rd_wt_adr
                [13:0]};
2180         ICNN_FM_CUR_PTR   : wb_rdata32 <= {19'b0, rd_fm_adr
                [13:0]};
2181         ICNN_OP_CUR_PTR   : wb_rdata32 <= {19'b0, rd_op_adr
                [13:0]};
2182         ICNN_FM_CUR_CH    : wb_rdata32 <= {26'b0, conv_ch_count };
2183         ICNN_CUR_FIL      : wb_rdata32 <= {26'b0, conv_fil_count };
2184         ICNN_WT_STATUS    : wb_rdata32 <= {31'b0, has_wts };
2185         ICNN_FM_STATUS    : wb_rdata32 <= {31'b0, has_fm_1 };
2186         ICNN_RELU_STATUS  : wb_rdata32 <= {31'b0, has_relu_fm };
```

```
2187     ICNN_MP_STATUS    : wb_rdata32 <= {31'b0, mp_cmplt };
2188     ICNN_CUR_ADR      : wb_rdata32 <= curr_adr ;
2189     ICNN_COMPLETE     : wb_rdata32 <= {26'b0, mp_fil_count };
2190     ICNN_FC_DONE      : wb_rdata32 <= {31'b0, fc_done };
2191     ICNN_LAYER_CMPLT : wb_rdata32 <= {28'b0, layer_count };
2192     ICNN_DONE         : wb_rdata32 <= {31'b0, ICNN_done };
2193         default :      wb_rdata32 <= 32'h00C0FFEE;
2194
2195         endcase
2196     end
2197 end
2198 end
2199
2200
2201 endmodule // ICNN
```

Listing I.12: ICNN

I.13 ACNN RTL

```
1
2 module ACNN (
3     reset ,
4     clk ,
5     scan_in0 ,
6     scan_en ,
7     test_mode ,
8     scan_out0 ,
9     o_m_wb_adr ,
10    o_m_wb_sel ,
11    o_m_wb_stb ,
12    i_m_wb_ack ,
13    i_m_wb_err ,
14    o_m_wb_we ,
15    o_m_wb_cyc ,
16    o_m_wb_dat ,
17    o_s_wb_ack ,
18    o_s_wb_err ,
19    i_s_wb_adr ,
20    i_s_wb_sel ,
21    i_s_wb_we ,
22    i_s_wb_dat ,
23    o_s_wb_dat ,
```

```
24     i_m_wb_dat ,
25     i_s_wb_cyc ,
26     i_s_wb_stb
27     );
28
29 `include " ../include/register_addresses.vh"
30 `include " ../include/icnn_state_defs.vh"
31
32 parameter BIT_WIDTH  = 16;
33 parameter WB_DWIDTH  = 32;
34 parameter WB_SWIDTH  = 4 ;
35 parameter NUM_LAYERS = 2;
36
37 parameter MAX_IP_SIZE_R = 13;
38 parameter MAX_IP_SIZE_C = 99;
39 parameter MAX_OP_SIZE_R = 11;
40 parameter MAX_OP_SIZE_C = 97;
41
42 parameter COMPARE_3      = 9830;
43 parameter COMPARE_8      = 26213;
44
45 localparam BIT_WIDTH7P   = BIT_WIDTH + 7;
46 localparam MAX_OPI_MR    = MAX_OP_SIZE_R - 1;
47 localparam MAX_OPI_MC    = MAX_OP_SIZE_C - 1;
48
```

49

50 **input**

51 reset , // system reset

52 clk ; // system clock

53

54 **input**

55 scan_in0 , // test scan mode data input

56 scan_en , // test scan mode enable

57 test_mode ; // test mode select

58

59 **output**

60 scan_out0 ; // test scan mode data output

61

62 //

63 // wishbone master and slave ports

64 //

65 **input** [WB_DWIDTH-1:0] i_m_wb_dat ;66 **input** i_m_wb_ack ;67 **input** i_m_wb_err ;68 **input** [31:0] i_s_wb_adr ;69 **input** [WB_SWIDTH-1:0] i_s_wb_sel ;

```
70 input          i_s_wb_we;
71 input          [WB_DWIDTH-1:0] i_s_wb_dat;
72 input          i_s_wb_cyc;
73 input          i_s_wb_stb;
74
75 output         [WB_DWIDTH-1:0] o_m_wb_adr;
76 output reg     [WB_SWIDTH-1:0] o_m_wb_sel;
77 output reg     o_m_wb_we;
78 output reg     [WB_DWIDTH-1:0] o_m_wb_dat;
79 output reg     o_m_wb_cyc;
80 output reg     o_m_wb_stb;
81 output         [WB_DWIDTH-1:0] o_s_wb_dat;
82 output         o_s_wb_ack;
83 output         o_s_wb_err;
84
85 wire           [WB_DWIDTH-1:0] o_s_wb_dat ;
86 wire           [WB_DWIDTH-1:0] o_m_wb_adr ;
87
88 reg            [WB_DWIDTH-1:0] curr_adr;
89 //=====ADD=====
90 reg [BIT_WIDTH7P:0] add_in1;
91 reg [BIT_WIDTH7P:0] add_in2;
92 wire [BIT_WIDTH7P:0] add_out;
93 //=====ACCUM=====
94 reg            fc_start_bit;
```



```
95 reg [BIT_WIDTH-1:0] fc_in1;
96 reg start_bit_conv;
97 reg [BIT_WIDTH-1:0] in1_conv;
98 wire start_bit_mux;
99 wire [BIT_WIDTH-1:0] in1_mux;
100 wire [BIT_WIDTH7P:0] acc_out;
101 //=====MUL=====
102 wire [BIT_WIDTH-1:0] im_mux;
103 wire [BIT_WIDTH-1:0] wt_mux;
104 reg [BIT_WIDTH-1:0] im_conv;
105 reg [BIT_WIDTH-1:0] wt_conv;
106 reg [BIT_WIDTH-1:0] fc_im;
107 reg [BIT_WIDTH-1:0] fc_wt;
108 //output decoded in the state machines
109 wire [BIT_WIDTH-1:0] mul_out;
110 //=====RELU=====
111 reg [BIT_WIDTH7P:0] fc_val_in;
112 reg [BIT_WIDTH7P:0] val_in_conv;
113 wire [BIT_WIDTH7P:0] val_in_mux;
114 wire [BIT_WIDTH-1:0] cmp_mux;
115 wire [BIT_WIDTH-1:0] relu_out;
116 //=====MAXPOOL=====
117 reg [BIT_WIDTH-1:0] mp1_val_1,
118 mp1_val_2,
119 mp1_val_3,
```

```
120             mp1_val_4;
121 wire [BIT_WIDTH-1:0]    mp1_out_1;
122
123 //=====
124 //===== Master R/W State Machine =====
125
126
127
128 //=====
129 //===== LAYER CONFIG =====
130
131 //----- Convolution -----
132 reg [7:0] conv_ip_size_r, //IP_SIZE row
133 conv_ip_size_2r, //IP_SIZE2X (2xrow)
134 conv_ip_size_c, //IP_SIZE col
135
136 conv_op_size_r, //OP_SIZE row
137 conv_op_size_c, //OP_SIZE col
138
139 conv_mp_size_r, //MP_SIZE row
140 conv_mp_size_c, //MP_SIZE col
141 conv_num_fil, //FIL_NUM
142 conv_num_ch; //CH_NUM
143
144 reg [15:0] conv_ip_size_rc, //IP_SIZE_SQ
```

```

145     conv_op_size_rc; //OP_SIZE_SQ
146
147 //----- Fully Connected -----
148 reg [15:0] fc_wt_row_cnt, //WT Row Count
149     fc_wt_col_cnt; //WT Col Count
150
151 //---- Other Layer Config Signals ----
152 reg conv_fc_mode,
153     mp_flag,
154     cmp_flag,
155     ACNN_done;
156 reg [13:0] cnfg_adr;
157 reg [3:0] config_state;
158 reg [3:0] layer_count;
159
160 //=====
161
162 //===== Read Weights State Machine ==
163 //reg [BIT_WIDTH-1:0] wt_chann [0:8]; //
164     current set of weights
164 reg [BIT_WIDTH-1:0] wt_ch0; //current set
165     of weights
165 reg [BIT_WIDTH-1:0] wt_ch1;
166 reg [BIT_WIDTH-1:0] wt_ch2;
167 reg [BIT_WIDTH-1:0] wt_ch3;

```

```
168 reg [BIT_WIDTH-1:0] wt_ch4;
169 reg [BIT_WIDTH-1:0] wt_ch5;
170 reg [BIT_WIDTH-1:0] wt_ch6;
171 reg [BIT_WIDTH-1:0] wt_ch7;
172 reg [BIT_WIDTH-1:0] wt_ch8;
173
174
175 reg [23:0] rd_wt_adr; //
    address pointer, 13:3 traverse vertical in MM,
176 //2:0 -> 16bit select, 2:1 -> 32
    bit select.
177 reg wt_rreq; //weights read
    request
178 reg [3:0] nth_wt; //
    weight state
179 //reg [5:0] wt_num_ch; //
    channel number WT
180 reg has_wts; //indicates weights are read in
    and are ready to convolve
181 //===== Read FMs State Machine =====
182 //reg [BIT_WIDTH-1:0] fm_buf1 [0:8]; //
    FM, fm -> Feature map,
183 //reg [BIT_WIDTH-1:0] fm_buf2 [0:8]; //fm
    values
184
```

```

185 reg [BIT_WIDTH-1:0]  fm_ch0;           // current set
      of weights
186 reg [BIT_WIDTH-1:0]  fm_ch1;
187 reg [BIT_WIDTH-1:0]  fm_ch2;
188 reg [BIT_WIDTH-1:0]  fm_ch3;
189 reg [BIT_WIDTH-1:0]  fm_ch4;
190 reg [BIT_WIDTH-1:0]  fm_ch5;
191 reg [BIT_WIDTH-1:0]  fm_ch6;
192 reg [BIT_WIDTH-1:0]  fm_ch7;
193 reg [BIT_WIDTH-1:0]  fm_ch8;
194
195 reg [13:0]            rd_fm_adr;       //
      main FM address pointer
196 reg [13:0]            r1_fm_ptr ,     //FM
      row1 adr ptr
197           r2_fm_ptr ,                 //FM row2 adr ptr
198           r3_fm_ptr;                  //FM row3 adr ptr
199 reg           fm_rreq;                 //FM read request
200 reg [3:0]           nth_fm;           //feature map state
201 reg [5:0]           conv_ch_count ,    //number of input
      channels of FM
202           conv_fil_count;             // Filter number WT =
      number of output channels should be equal;
203 reg [7:0]           row_cnt ,         //row count to slide
      windows vertically thru FM

```

```
204         col_cnt;                //column count to
           slide windows horizontally thru FM
205 reg     fm_ch_cmplt,           //indicates completion of one
           complete channel of input FM
206         fm_cmplt,              //indicates completion of all
           channels of input FM
207         all_fm_cmplt;
208 reg     has_fm_1;              //indicates fm_buf1 has values
           that are ready to convolve
209 //     has_fm_2;              //indicates fm_buf2 has values
           that are ready to convolve
210 //===== ACCMUL, RELU, MAXPOOL =====
211 reg [BIT_WIDTH7P:0] conv_op[0:MAX_OPIMR][0:MAX_OPIMC]; //
           stores output of convolution until second stage of
           accumulation and RELU
212 reg [4:0]     mul_cnt;                //acumul state
213 reg [7:0]     curr_ch,                //store current channel
           from FM State Machine
214         curr_row,                  //store current row
215         curr_col;                  //store current col
216 reg [7:0]     mp_row,                //row count for MAXPOOL
217         mp_col,                    //col count for MAXPOOL
218         mp_row_1p,                 //row count + 1 for MAXPOOL
           indexing
```

```
219         mp_col_1p;           //col count + 1 for MAXPOOL
           indexing
220 reg [6:0]          mp_fil_count;           //count number
           of times mp has completed.
221 reg              acc_mul_cmplt;           //indicates
           completion of one convolution operation.
222              //dotprod of 9 wts and 9fms, and accumulate 9
           resulting values.
223 reg              has_relu_fm;           //turns on after second stage
           of accumulation
224              //2nd stage accum-> adds all channel data.
225              //perform relu
226              //send to MAXPOOL when fm_cmplt turns on
227 reg [13:0]        rd_op_adr;           //address
           pointer output -> writes to MM
228 reg              op_wreq,           //write request to memory
           after every single value is computed.
229         mp_cmplt;           //indicates completion of MAXPOOL
           operation.
230 reg [WB_DWIDTH-1:0] write_buf;           //Maxpool state writes
           result to buffer that's to be written to MM.
231 reg              selx_0, selx_1;           //selecting which part of
           16 bits of 32bits to write to.
232 reg [7:0] mp_iterator_r;
233 reg [7:0] mp_iterator_c;
```

```
234 //=====Wishbone Interface=====
235 reg [2:0] wb_adr_wt_base_ptr;
236 reg [12:0] wb_adr_fm_base_ptr;
237 reg [12:0] wb_adr_op_base_ptr;
238 reg [12:0] wb_adr_cnfg_base_ptr;
239 reg [20:0] wb_adr_wt_ptr;
240 reg [10:0] wb_adr_fm_ptr ,
241 wb_adr_op_ptr ,
242 wb_adr_cnfg_ptr; //11 bits to
traverse vertically in main_mem
243 reg [31:0] wb_rdata32;
244 wire [31:0] wb_wdata32;
245
246 reg valid_wt_base_adr ,
247 valid_fm_base_adr ,
248 valid_op_base_adr ,
249 start_bit;
250 reg start_read_1;
251 wire start_read ,
252 start_write;
253
254
255 //===== FULLY CONNECTED =====
256 // reg [15:0] fc_fm_buf[0:1];
257 // reg [15:0] fc_wt_buf[0:1];
```



```
258
259 reg [15:0]          fc_fm_0 ,
260          fc_fm_1 ,
261          fc_wt_0 ,
262          fc_wt_1 ;
263
264 reg [31:0]          fc_op_buf ;
265 reg [4:0]           fc_state ;
266 reg [10:0]          fc_row_count ,
267          fc_col_count ,
268          r_fake ,
269          c_fake ;
270
271 reg [23:0]          fc_wt_adr ;
272 reg [13:0]          fc_fm_adr ;
273 reg [13:0]          fc_op_adr ;
274 reg          fc_wt_rreq ,
275          fc_fm_rreq ,
276          fc_op_wreq ;
277 reg          fc_done ,
278          fc_sel_0 ,
279          fc_sel_1 ;
280 reg [2:0]          fc_count ;
281 // =====
282 wire          reset_mux ;
```

```
283 reg                                sw_reset;
284
285 integer                              j, //to reset conv_op rows
286                                     k; //to reset conv_op cols
287
288 ADD #(.BIT_WIDTH(BIT_WIDTH)) adder(.reset(reset_mux),
289     .clk(clk),
290     .scan_in0(scan_in0),
291     .scan_en(scan_en),
292     .test_mode(test_mode),
293     .scan_out0(scan_out0),
294     .in1(add_in1),
295     .in2(add_in2),
296     .out(add_out));
297
298
299 ACCUM #(.BIT_WIDTH(BIT_WIDTH)) acc_ch1(.reset(reset_mux),
300     .clk(clk),
301     .scan_in0(scan_in0),
302     .scan_en(scan_en),
303     .test_mode(test_mode),
304     .scan_out0(scan_out0),
305     .start_bit(start_bit_mux),
306     .in1(in1_mux),
307     .out(acc_out)
```

```
308         );
309
310 MUL #(.BIT_WIDTH(BIT_WIDTH)) mul_ch1 (.reset(reset_mux),
311         .clk(clk),
312         .scan_in0(scan_in0),
313         .scan_en(scan_en),
314         .test_mode(test_mode),
315         .scan_out0(scan_out0),
316         .in1(im_mux),
317         .in2(wt_mux),
318         .out(mul_out)
319     );
320
321 RELU #(.IP_WIDTH(BIT_WIDTH*7P+1), .OP_WIDTH(BIT_WIDTH)) relu_ch1
322     (.reset(reset_mux),
323         .clk(clk),
324         .scan_in0(scan_in0),
325         .scan_en(scan_en),
326         .test_mode(test_mode),
327         .scan_out0(scan_out0),
328         .val_in(val_in_mux),
329         .compare(cmp_mux),
330         .val_out(relu_out)
331     );
```

```
332 MAXPOOL #(.BIT_WIDTH(BIT_WIDTH)) mp_1(.reset(reset_mux),
333     .clk(clk),
334     .scan_in0(scan_in0),
335     .scan_en(scan_en),
336     .test_mode(test_mode),
337     .scan_out0(scan_out0),
338     .in1(mp1_val_1),
339     .in2(mp1_val_2),
340     .in3(mp1_val_3),
341     .in4(mp1_val_4),
342     .max_out(mp1_out_1)
343 );
344
345
346 assign reset_mux      = (test_mode) ? reset : (sw_reset ||
    reset);
347 assign o_m_wb_adr    = curr_adr;
348 // Implementing MUX for ACCUM and MUL - Reusing hardware for FC
349 assign im_mux        = (conv_fc_mode) ? fc_im:im_conv;
350 assign wt_mux        = (conv_fc_mode) ? fc_wt:wt_conv;
351 assign start_bit_mux = (conv_fc_mode) ? fc_start_bit:
    start_bit_conv;
352 assign in1_mux       = (conv_fc_mode) ? fc_in1 : in1_conv;
353 assign val_in_mux    = (conv_fc_mode) ? fc_val_in:val_in_conv;
354 assign cmp_mux       = (cmp_flag)? COMPARE_8:COMPARE_3;
```

```
355
356 //=====
357 //----- State Machine - Convolution -----
358 //=====
359
360
361 //=====
362 //----- Master READS and WRITES -----
363 //=====
364 always @(posedge clk or posedge reset_mux)
365 begin
366     if(reset_mux) begin
367         //o_m_wb_adr <= 0;
368         curr_adr    <= 32'b0;
369         o_m_wb_sel <= 0;
370         o_m_wb_dat <= 0;
371         o_m_wb_we  <= 0;
372         o_m_wb_cyc <= 0;
373         o_m_wb_stb <= 0;
374     end else begin
375         if(!start_bit) begin
376             o_m_wb_cyc <= 1'b0;
377             o_m_wb_stb <= 1'b0;
378             o_m_wb_we  <= 1'b0;
379             o_m_wb_cyc <= 1'b0;
```

```
380     o_m_wb_stb <= 1'b0;
381     o_m_wb_sel <= 4'b0;
382     end else begin
383     if (!ACNN_done && !valid_wt_base_adr && !valid_fm_base_adr
        && !valid_op_base_adr) begin
384     curr_adr    <= {4'b0, wb_adr_cnfg_base_ptr, cnfg_adr
        [13:1], 2'b0};
385     o_m_wb_cyc <= 1'b1;
386     o_m_wb_stb <= 1'b1;
387     if (i_m_wb_ack) begin
388     o_m_wb_we  <= 1'b0;
389     o_m_wb_cyc <= 1'b0;
390     o_m_wb_stb <= 1'b0;
391     o_m_wb_sel <= 4'b0;
392     end
393     end else if (valid_wt_base_adr && valid_fm_base_adr &&
        valid_op_base_adr) begin
394     if (!conv_fc_mode) begin
395     if (mp_fil_count < conv_num_fil) begin
396     if (valid_wt_base_adr && !has_wts && !acc_mul_cmplt &&
        !has_relu_fm) begin //&& !acc_mul_cmplt not sure
397     curr_adr    <= {4'b0, wb_adr_wt_base_ptr, rd_wt_adr
        [23:1], 2'b0};
398     o_m_wb_cyc <= 1'b1;
399     o_m_wb_stb <= 1'b1;
```

```
400
401     // Access bus strictly when necessary , has_wts is ON
         and has_fm is low
402     // and fm_ch_cmplt goes high next cycle which turns
         has_wts low.
403     // and requests bus access to read weights but fm is
         read instead -> hence added !fm_ch_cmplt.
404     end else if (valid_fm_base_addr && has_wts && !has_fm_1
         && !fm_ch_cmplt) begin
405         curr_addr    <= {4'b0, wb_addr_fm_base_ptr, rd_fm_addr
         [13:1], 2'b0};
406         o_m_wb_cyc <= 1'b1;
407         o_m_wb_stb <= 1'b1;
408
409     end else if (valid_op_base_addr && has_relu_fm &&
         op_wreq) begin
410         curr_addr    <= {4'b0, wb_addr_op_base_ptr, rd_op_addr
         [13:1], 2'b0};
411         o_m_wb_cyc <= 1'b1;
412         o_m_wb_stb <= 1'b1;
413         o_m_wb_we   <= 1'b1;
414         o_m_wb_dat <= write_buf;
415         o_m_wb_sel <= {{2{selx_1}}, {2{selx_0}}};
416     end else begin
417         o_m_wb_cyc <= 1'b0;
```

```
418         o_m_wb_stb <= 1'b0;
419         o_m_wb_we  <= 1'b0;
420         o_m_wb_sel <= 4'b0;
421     end
422
423     if (i_m_wb_ack) begin
424         o_m_wb_we  <= 1'b0;
425         o_m_wb_cyc <= 1'b0;
426         o_m_wb_stb <= 1'b0;
427         o_m_wb_sel <= 4'b0;
428     end
429 end
430 end else begin //conv_fc_mode is high -> FC layer
431
432     if(fc_wt_rreq && !fc_fm_rreq && !fc_op_wreq) begin //
433         && !acc_mul_cmplt not sure
434         curr_adr    <= {4'b0, wb_adr_wt_base_ptr, fc_wt_adr
435             [23:1], 2'b0};
436         o_m_wb_cyc <= 1'b1;
437         o_m_wb_stb <= 1'b1;
438     end else if (!fc_wt_rreq && fc_fm_rreq && !fc_op_wreq
439         ) begin
440         curr_adr    <= {4'b0, wb_adr_fm_base_ptr, fc_fm_adr
441             [13:1], 2'b0};
442         o_m_wb_cyc <= 1'b1;
```



```
439         o_m_wb_stb <= 1'b1;
440     end else if (!fc_wt_rreq && !fc_fm_rreq && fc_op_wreq)
         begin
441         curr_adr    <= {4'b0, wb_adr_op_base_ptr, fc_op_adr
                [13:1], 2'b0};
442         o_m_wb_cyc <= 1'b1;
443         o_m_wb_stb <= 1'b1;
444         o_m_wb_we  <= 1'b1;
445         o_m_wb_dat <= fc_op_buf;
446         o_m_wb_sel <= {{2{fc_sel_1}}, {2{fc_sel_0}}};
447     end else begin
448         o_m_wb_cyc <= 1'b0;
449         o_m_wb_stb <= 1'b0;
450         o_m_wb_we  <= 1'b0;
451         o_m_wb_sel <= 4'b0;
452     end
453
454     if (i_m_wb_ack) begin
455         o_m_wb_we  <= 1'b0;
456         o_m_wb_cyc <= 1'b0;
457         o_m_wb_stb <= 1'b0;
458         o_m_wb_sel <= 4'b0;
459     end
460
461 end
```

```
462     end
463     end
464 end
465 end // always
466
467 //=====
468 //----- Layer - Configuration -----
469 //=====
470
471 always @(posedge clk or posedge reset_mux)
472 begin
473     if(reset_mux) begin
474         conv_ip_size_r    <= 8'b0;
475         conv_ip_size_2r  <= 8'b0;
476         conv_ip_size_c    <= 8'b0;
477         conv_op_size_r    <= 8'b0;
478         conv_op_size_c    <= 8'b0;
479         conv_mp_size_r    <= 8'b0;
480         conv_mp_size_c    <= 8'b0;
481
482         conv_ip_size_rc   <= 16'b0;
483         conv_op_size_rc   <= 16'b0;
484
485         conv_num_fil      <= 8'b0;
486         conv_num_ch       <= 8'b0;
```

```
487
488     fc_wt_row_cnt      <= 16'b0;
489     fc_wt_col_cnt      <= 16'b0;
490
491     conv_fc_mode       <= 1'b0;
492     mp_flag            <= 1'b0;
493     cmp_flag           <= 1'b0;
494
495     valid_wt_base_adr  <= 1'b0;
496     wb_adr_wt_base_ptr <= 0;
497     wb_adr_wt_ptr      <= 0;
498
499     valid_fm_base_adr  <= 1'b0;
500     wb_adr_fm_base_ptr <= 13'b0;
501     wb_adr_fm_ptr      <= 11'b0;
502
503     valid_op_base_adr  <= 1'b0;
504     wb_adr_op_base_ptr <= 13'b0;
505     wb_adr_op_ptr      <= 11'b0;
506
507     layer_count        <= 4'b0;
508     cnfg_adr           <= 13'b0;
509     config_state       <= 4'b0;
510
511     ACNN_done          <= 1'b0;
```

```
512
513     end else begin
514         if (!start_bit) begin
515             //reset or hold?
516         end else begin
517             case (config_state)
518                 'CNFG_IDLE: begin
519                     if (!fc_done && !valid_wt_base_adr && !valid_fm_base_adr
520                         && !valid_op_base_adr) begin
521                         conv_ip_size_r  <= 8'b0;
522                         conv_ip_size_2r <= 8'b0;
523                         conv_ip_size_c  <= 8'b0;
524                         conv_op_size_r  <= 8'b0;
525                         conv_op_size_c  <= 8'b0;
526                         conv_mp_size_r  <= 8'b0;
527                         conv_mp_size_c  <= 8'b0;
528
529                         conv_ip_size_rc <= 16'b0;
530                         conv_op_size_rc <= 16'b0;
531
532                         conv_num_fil   <= 8'b0;
533                         conv_num_ch    <= 8'b0;
534
535                         fc_wt_row_cnt  <= 16'b0;
536                         fc_wt_col_cnt  <= 16'b0;
```

```
536
537     conv_fc_mode    <= 1'b0;
538     mp_flag        <= 1'b0;
539     cmp_flag       <= 1'b0;
540
541     valid_wt_base_adr <= 1'b0;
542     wb_adr_wt_base_ptr <= 0;
543     wb_adr_wt_ptr     <= 0;
544
545     valid_fm_base_adr <= 1'b0;
546     wb_adr_fm_base_ptr <= 13'b0;
547     wb_adr_fm_ptr     <= 11'b0;
548
549     valid_op_base_adr <= 1'b0;
550     wb_adr_op_base_ptr <= 13'b0;
551     wb_adr_op_ptr     <= 11'b0;
552
553     // layer_count <= 4'b0;
554     cnfg_adr        <= {wb_adr_cnfg_ptr, {3'b0}}; // set
           this the first time only
555     config_state <= 'CNFG_START_BIT;
556
557     ACNN_done     <= 0; // continues to read FM if start
           bit is on.
558
```

```
559     end else if (!ACNN_done && valid_wt_base_adr &&
        valid_fm_base_adr && valid_op_base_adr) begin
560     if (((!conv_fc_mode) && (mp_fil_count == conv_num_fil)
        ) || (conv_fc_mode && fc_done)) begin
561     // Clear/reset all config params after completion of
        conv/fc layer.
562     // other than config base pointer.
563     conv_ip_size_r   <= 8'b0;
564     conv_ip_size_2r <= 8'b0;
565     conv_ip_size_c   <= 8'b0;
566     conv_op_size_r   <= 8'b0;
567     conv_op_size_c   <= 8'b0;
568     conv_mp_size_r   <= 8'b0;
569     conv_mp_size_c   <= 8'b0;
570
571     conv_ip_size_rc  <= 16'b0;
572     conv_op_size_rc  <= 16'b0;
573
574     conv_num_fil     <= 8'b0;
575     conv_num_ch      <= 8'b0;
576
577     fc_wt_row_cnt    <= 16'b0;
578     fc_wt_col_cnt    <= 16'b0;
579
580     conv_fc_mode     <= 1'b0;
```

```
581         mp_flag          <= 1'b0;
582         cmp_flag         <= 1'b0;
583
584         valid_wt_base_adr <= 1'b0;
585         wb_adr_wt_base_ptr <= 0;
586         wb_adr_wt_ptr     <= 0;
587
588         valid_fm_base_adr <= 1'b0;
589         wb_adr_fm_base_ptr <= 13'b0;
590         wb_adr_fm_ptr     <= 11'b0;
591
592         valid_op_base_adr <= 1'b0;
593         wb_adr_op_base_ptr <= 13'b0;
594         wb_adr_op_ptr     <= 11'b0;
595
596         if(layer_count < NUM_LAYERS-1) begin
597             layer_count    <= layer_count +1;
598             config_state   <= 'CNFG_START_BIT;
599         end else if(layer_count == NUM_LAYERS-1) begin
600             layer_count    <= 0;
601             ACNN_done      <= 1;
602         end
603     end
604 end
605 end
```

```
606
607     'CNFG_START_BIT: begin
608         if(i_m_wb_ack) begin
609             {mp_flag , cmp_flag , conv_fc_mode} <= i_m_wb_dat
610                 [2:0]; //3 bits
611             {conv_num_fil , conv_num_ch} <= i_m_wb_dat
612                 [31:16]; //8+8 bits
613             conv_ip_size_2r <= i_m_wb_dat
614                 [15:8]; //8 bits
615             cnfg_adr <= cnfg_adr + 2; //read 32 bits
616                 -> +2
617             config_state <= 'CNFG_IP_SIZE;
618         end
619     end
620
621     'CNFG_IP_SIZE: begin
622         if(i_m_wb_ack) begin
623             if(conv_fc_mode) begin
624                 {fc_wt_row_cnt , fc_wt_col_cnt} <= i_m_wb_dat; //
625                     16+16 bits
626                 cnfg_adr <= cnfg_adr + 2; //read 32 bits
627                     -> +2
628                 config_state <= 'CNFG_WT_BASE_ADR;
629             end else begin
630                 conv_ip_size_r <= i_m_wb_dat[23:16];
```



```
625         conv_ip_size_c <= i_m_wb_dat[7:0];    //8+8 skip 8+8
626         cnfg_adr        <= cnfg_adr + 2;      //read 32 bits
           -> +2
627         //{ conv_op_size_r , conv_op_size_c }<= i_m_wb_dat
           [32:16];
628         config_state    <= 'CNFG_MP_SIZE;
629     end
630 end
631 end
632
633 'CNFG_MP_SIZE: begin
634     if(i_m_wb_ack) begin
635         { conv_op_size_r , conv_op_size_c } <= i_m_wb_dat
           [31:16];
636         { conv_mp_size_r , conv_mp_size_c } <= i_m_wb_dat[15:0];
637         cnfg_adr        <= cnfg_adr + 2;
638         config_state    <= 'CNFG_RxC_SIZE;
639     end
640 end
641
642 'CNFG_RxC_SIZE: begin
643     if(i_m_wb_ack) begin
644         { conv_ip_size_rc , conv_op_size_rc } <= i_m_wb_dat;
645         cnfg_adr        <= cnfg_adr + 2;
646         config_state    <= 'CNFG_WT_BASE_ADR;
```

```
647         end
648     end
649
650     'CNFG_WT_BASE_ADR: begin
651         if (i_m_wb_ack) begin
652             {wb_adr_wt_base_ptr, wb_adr_wt_ptr} <= i_m_wb_dat
653                 [23:0];
654             cnfg_adr          <= cnfg_adr + 2;
655             config_state     <= 'CNFG_FM_BASE_ADR;
656         end
657     end
658
659     'CNFG_FM_BASE_ADR: begin
660         if (i_m_wb_ack) begin
661             {wb_adr_fm_base_ptr, wb_adr_fm_ptr} <= i_m_wb_dat
662                 [23:0];
663             cnfg_adr          <= cnfg_adr + 2;
664             config_state     <= 'CNFG_OP_BASE_ADR;
665         end
666     end
667
668     'CNFG_OP_BASE_ADR: begin
669         if (i_m_wb_ack) begin
670             {wb_adr_op_base_ptr, wb_adr_op_ptr} <= i_m_wb_dat
671                 [23:0];
```

```
669         cnfg_adr             <= cnfg_adr + 2;
670         valid_op_base_adr <= 1;
671         valid_fm_base_adr <= 1;
672         valid_wt_base_adr <= 1;
673         config_state        <= 'CNFG_IDLE;
674     end
675 end
676
677     default: begin
678         config_state <= 'CNFG_IDLE;
679     end
680 endcase
681 end
682 end
683 end // always
684
685
686 //=====
687 //----- Fully Connected -----
688 //=====
689 always @(posedge clk or posedge reset_mux)
690 begin
691     if(reset_mux) begin
692         fc_im         <= 0;
693         fc_wt         <= 0;
```

```
694     fc_start_bit <= 0;
695     fc_in1        <= 0;
696     fc_val_in     <= 0;
697
698     fc_fm_0       <= 0;
699     fc_fm_1       <= 0;
700     fc_wt_0       <= 0;
701     fc_wt_1       <= 0;
702
703     fc_state      <= 0;
704     fc_wt_adr     <= 0;
705     fc_fm_adr     <= 0;
706     fc_op_adr     <= 0;
707     fc_row_count  <= 0;
708     fc_col_count  <= 0;
709     r_fake        <= 0;
710     c_fake        <= 0;
711     fc_done       <= 0;
712     fc_wt_rreq    <= 0;
713     fc_fm_rreq    <= 0;
714     fc_op_wreq    <= 0;
715     fc_op_buf     <= 0;
716     fc_sel_0      <= 0;
717     fc_sel_1      <= 0;
718
```

```
719     fc_count <= 0;
720 end else begin
721     case(fc_state)
722         'IDLE_FC: begin
723             if(!fc_done && valid_wt_base_adr && valid_fm_base_adr
724                 && valid_op_base_adr && conv_fc_mode) begin
725                 //enters here the first time it's reading weights.
726
727                 //Transition
728                 fc_fm_0      <= 0;
729                 fc_fm_0      <= 0;
730                 fc_wt_0      <= 0;
731                 fc_wt_0      <= 0;
732
733                 //base address set at the beginning and is not
734                 //everytime new set of weights are read.
735                 fc_wt_adr     <= {wb_adr_wt_ptr, {3'b0}};
736                 fc_fm_adr     <= {wb_adr_fm_ptr, {3'b0}};
737                 fc_op_adr     <= {wb_adr_op_ptr, {3'b0}};
738                 fc_wt_rreq    <= 1;
739                 fc_fm_rreq    <= 0;
740                 fc_op_wreq    <= 0;
741                 fc_sel_0      <= 0;
```

```
742         fc_sel_1      <= 0;
743         fc_row_count <= 0;
744         fc_col_count <= 0;
745         r_fake       <= 0;
746         c_fake       <= 0;
747         fc_in1       <= 0;
748         if(ACNN_done) begin
749             // Assuming ACNN_done turns on after completion of
750                 last layer.
751             fc_count <= 0;
752             end
753             if(fc_count == 0) begin
754                 fc_state <= 'CONV_FC_WT0;
755             end else begin
756                 fc_state <= 'FC_WT0;
757             end
758         end
759     end else if( fc_done && !valid_wt_base_adr && !
760         valid_fm_base_adr && !valid_op_base_adr ) begin
761         fc_done <= 0;
762         fc_wt_adr <= 0;
763         fc_fm_adr <= 0;
764         fc_row_count <= 0;
765         fc_col_count <= 0;
```

```
765         fc_count <= fc_count +1;
766         if(ACNN_done) begin
767             // Assuming ACNN_done turns on after completion of
768                 last layer.
769             fc_count <= 0;
770         end
771     end
772
773     //=====
774     //----- Layer 5 -----
775     //=====
776     `CONV_FC_WT0: begin
777         fc_wt_rreq <= 1;
778         if (i_m_wb_ack) begin
779             if(!fc_wt_adr[0]) begin
780                 // Transition
781                 //fc_wt_0 <= i_m_wb_dat[15:0]; //w0
782                 fc_wt    <= i_m_wb_dat[15:0];
783             end else begin
784                 // Transition
785                 //fc_wt_0 <= i_m_wb_dat[31:16]; //w0
786                 fc_wt    <= i_m_wb_dat[31:16];
787             end
788         end
```

```
789         fc_wt_rreq <= 0;
790         fc_fm_rreq <= 1;
791         // fc_wt_adr  <= fc_wt_adr + (fc_wt_row_count+1)*
           fc_wt_col_cnt+fc_wt_col_count;
792         fc_state  <= 'CONV_FC_FM0;
793     end
794 end
795
796 'CONV_FC_FM0: begin
797     if (i_m_wb_ack) begin
798         if (!fc_fm_adr[0]) begin
799             // fc_fm_0  <= i_m_wb_dat[15:0];    //fm0
800             fc_im      <= i_m_wb_dat[15:0];
801         end else begin
802             // fc_fm_0  <= i_m_wb_dat[31:16]; //fm0
803             fc_im      <= i_m_wb_dat[31:16];
804         end
805
806         fc_fm_rreq <= 0;
807         // fc_fm_adr <= fc_fm_adr + 2;
808         fc_state <= 'CONV_FC_MUL0;
809     end
810 end
811
812 'CONV_FC_MUL0: begin
```



```
813         //NOP
814         fc_state  <= 'CONV_MUL_OUT;
815     end
816
817     'CONV_MUL_OUT: begin
818         fc_start_bit <= 1'b1;
819         fc_in1       <= mul_out;
820         fc_wt        <= 0;
821         fc_im        <= 0;
822         fc_state     <= 'CONV_FC_ACC;
823     end
824
825     'CONV_FC_ACC: begin
826         fc_in1      <= 0;
827         if(fc_row_count < fc_wt_row_cnt-1) begin
828             //fc_in1          <= mul_out;
829             //read weights and FM again
830             fc_row_count  <= fc_row_count + 1;
831             if(r_fake < conv_num_fil-1) begin
832                 r_fake <= r_fake + 1;
833             end else begin
834                 r_fake <= 0;
835                 c_fake <= c_fake + 1;
836             end
837         fc_state      <= 'NEXT_ADDR;
```

```
838
839     end else begin
840         if(fc_col_count < fc_wt_col_cnt) begin
841             // fc_start_bit <= 1'b0;
842             // fc_in1   <= mul_out;
843             // on completing row count -> ReLu
844             fc_state <= 'CONV_ACC_OUT;
845         end else begin
846             fc_done  <= 1;
847             fc_start_bit <= 1'b0;
848             fc_state <= 'IDLE_FC;
849         end
850     end
851 end
852
853 'CONV_ACC_OUT: begin
854     fc_start_bit <= 1'b0;
855     fc_state     <= 'CONV_RELU_OUT;
856 end
857
858 'CONV_RELU_OUT: begin
859     fc_val_in  <= acc_out;
860     fc_in1    <= 0;
861     fc_state  <= 'WREQ;
862 end
```

```
863
864     'NEXT_ADDR: begin
865         fc_wt_adr    <= {wb_adr_wt_ptr, {3'b0}} + fc_row_count*
            fc_wt_col_cnt + fc_col_count;
866         fc_fm_adr    <= {wb_adr_fm_ptr, {3'b0}} + conv_num_ch*
            conv_ip_size_2r*r_fake + c_fake;
867         // fc_in1     <= 0;
868         fc_state     <= 'CONV_FC_WT0;
869     end
870
871     //=====
872     //----- Layers 6 through 7 -----
873     //=====
874     'NEXT_WT0_ADDR: begin
875         fc_wt_rreq <= 1;
876         fc_wt_adr  <= {wb_adr_wt_ptr, {3'b0}} + fc_row_count*
            fc_wt_col_cnt + fc_col_count;
877         fc_state   <= 'FC_WT0;
878     end
879
880     'FC_WT0: begin
881         fc_in1 <= 0;
882         fc_wt_rreq <= 1;
883         if (i_m_wb_ack) begin
884             if (!fc_wt_adr[0]) begin
```

```
885         // Transition
886         fc_wt_0 <= i_m_wb_dat[15:0]; //w0
887         // fc_wt    <= i_m_wb_dat[15:0];
888     end else begin
889         // Transition
890         fc_wt_0 <= i_m_wb_dat[31:16]; //w0
891         // fc_wt    <= i_m_wb_dat[31:16];
892     end
893
894     fc_wt_rreq    <= 0;
895     // fc_fm_rreq <= 1;
896     fc_row_count <= fc_row_count + 1;
897
898     // fc_wt_adr  <= (fc_wt_row_count+1)*fc_wt_col_cnt+
899                 fc_wt_col_count;
900     fc_state     <= 'NEXT_WT1_ADDR;
901 end
902
903 'NEXT_WT1_ADDR: begin
904     fc_wt_rreq <= 1;
905     fc_wt_adr  <= {wb_adr_wt_ptr, {3'b0}} + fc_row_count*
906                 fc_wt_col_cnt + fc_col_count;
907     fc_state   <= 'FC_WT1;
908 end
```

```
908
909     'FC_WT1: begin
910         if (i_m_wb_ack) begin
911             if (!fc_wt_adr[0]) begin
912                 // Transition
913                 fc_wt_1 <= i_m_wb_dat[15:0]; //w1
914             end else begin
915                 // Transition
916                 fc_wt_1 <= i_m_wb_dat[31:16]; //w1
917             end
918
919             fc_wt_rreq <= 0;
920             fc_fm_rreq <= 1;
921             fc_row_count <= fc_row_count + 1;
922             fc_state <= 'FC_FM0;
923         end
924     end
925
926     'FC_FM0: begin
927         //address pointer is incremented by two or one every
928         //time based on number
929         //of values read in at once (based of parity bit.)
930         //parity check is done
931         //in every state unlike in FM read State Machine
932         //because the values are
```

```
930      //read continuously and even & odd indexes can occur in
          any state .
931      if (i_m_wb_ack) begin
932          if (!fc_fm_adr[0]) begin
933              // Transition
934              fc_fm_0    <= i_m_wb_dat[15:0]; //fm0
935              fc_fm_1    <= i_m_wb_dat[31:16]; //fm1
936              fc_fm_adr  <= fc_fm_adr + 2;
937              fc_fm_rreq <= 0;
938              fc_wt_rreq <= 0;
939              fc_state   <= 'FC_MUL0;
940          end else begin
941              // Transition
942              fc_fm_0    <= i_m_wb_dat[31:16]; //fm0
943              fc_fm_adr  <= fc_fm_adr + 1;
944              fc_state   <= 'FC_FM1;
945          end
946      end
947  end
948
949  'FC_FM1: begin
950      //address pointer is incremented by two or one every
          time based on number
951      //of values read in at once (based of parity bit.)
          parity check is done
```

```
952      //in every state unlike in FM read State Machine
          because the values are
953      //read continuously and even & odd indexes can occur in
          any state .
954      if (i_m_wb_ack) begin
955          if (!fc_fm_adr[0]) begin
956              // Transition
957              fc_fm_1    <= i_m_wb_dat[15:0];    //fm1
958              fc_fm_adr  <= fc_fm_adr + 1;
959              fc_fm_rreq <= 0;
960              fc_wt_rreq <= 0;
961              fc_state   <= 'FC_MUL0;
962          end
963      end
964  end
965
966  'FC_MUL0: begin
967      fc_im    <= fc_fm_0;
968      fc_wt    <= fc_wt_0;
969      fc_state <= 'FC_MUL1;
970  end
971
972  'FC_MUL1: begin
973      fc_im    <= fc_fm_1;
974      fc_wt    <= fc_wt_1;
```

```
975         fc_state  <= 'MUL_OUT';
976     end
977
978     'MUL_OUT: begin
979         fc_start_bit <= 1'b1;
980         // Clear first set of values after collecting mul_out
981         fc_in1      <= mul_out;
982         fc_fm_0     <= 0;
983         fc_wt_0     <= 0;
984         fc_state    <= 'FC_ACC;
985     end
986
987     'FC_ACC: begin
988         // Since processing two values at once, divide by 2
989         if(fc_row_count < fc_wt_row_cnt-1) begin
990             // Clear second set of values after collecting 2nd
991                 mul_out
992             fc_in1    <= mul_out;
993             fc_fm_1  <= 0;
994             fc_wt_1  <= 0;
995             fc_im    <= 0;
996             fc_wt    <= 0;
997             // read weights and FM again
998             fc_wt_adr <= {wb_adr_wt_ptr, {3'b0}} + fc_row_count
999                 *fc_wt_col_cnt + fc_col_count;
```



```
998         //fc_row_count <= fc_row_count + 1;
999         fc_state     <= 'FC_WT0;
1000
1001     end else begin
1002         if(fc_col_count < fc_wt_col_cnt) begin
1003             //fc_start_bit <= 1'b0;
1004             fc_in1     <= mul_out;
1005             fc_fm_1    <= 0;
1006             fc_wt_1    <= 0;
1007             fc_im      <= 0;
1008             fc_wt      <= 0;
1009             //on completing row count -> ReLu
1010             fc_state <= 'ACC_OUT;
1011         end else begin
1012             fc_in1     <= 0;
1013             fc_start_bit <= 1'b0;
1014             fc_done     <= 1;
1015             fc_state     <= 'IDLE_FC;
1016         end
1017     end
1018 end
1019
1020 'ACC_OUT: begin
1021     fc_in1     <= 0;
1022     fc_start_bit <= 1'b0;
```

```
1023         fc_state      <= 'RELU_OUT;
1024     end
1025
1026     'RELU_OUT: begin
1027         fc_val_in      <= acc_out;
1028         fc_state      <= 'WREQ;
1029     end
1030
1031     'WREQ: begin
1032         // write back to memory
1033         if (!fc_op_wreq) begin
1034             if (!fc_op_adr[0]) begin
1035                 fc_op_buf[15:0] <= relu_out;
1036
1037                 if (layer_count == 4'd7 && fc_col_count ==
1038                     fc_wt_col_cnt - 1) begin
1039                     fc_sel_0 <= 1;
1040                     fc_op_wreq <= 1;
1041                 end else begin
1042                     fc_op_wreq <= 0;
1043                     fc_sel_0 <= 1;
1044                     fc_op_adr <= fc_op_adr + 1;
1045                     fc_row_count <= 0;
1046                     fc_col_count <= fc_col_count + 1;
1047                     // reset FM pointer
```

```
1047         fc_fm_adr    <= {wb_adr_fm_ptr, {3'b0}};
1048         //reset WT pointer
1049         //fc_wt_adr   <= {wb_adr_wt_ptr, {3'b0}};
1050         //WT adr ptr remains constant
1051         //read weights and FM again
1052
1053         if(fc_count == 0) begin
1054             c_fake    <= 0;
1055             r_fake    <= 0;
1056             fc_state  <= 'NEXT_ADDR;
1057         end else begin
1058             fc_state  <= 'NEXT_WT0_ADDR;
1059         end
1060     end
1061
1062     end else begin
1063         fc_op_buf[31:16] <= relu_out;
1064         fc_sel_1    <= 1;
1065         fc_op_wreq <= 1;
1066     end
1067 end
1068
1069 if (i_m_wb_ack) begin
1070     fc_op_wreq    <= 0;
1071     fc_sel_0     <= 0;
```

```
1072         fc_sel_1      <= 0;
1073         if(layer_count == 4'd7 && fc_col_count ==
           fc_wt_col_cnt-1) begin
1074             fc_row_count <= 0;
1075             fc_col_count <= 0;
1076             fc_done      <= 1;
1077             fc_state <= 'IDLE_FC;
1078         end else begin
1079             fc_row_count <= 0;
1080             fc_col_count <= fc_col_count + 1;
1081             fc_op_adr     <= fc_op_adr + 1;
1082             //reset FM pointer
1083             fc_fm_adr     <= {wb_adr_fm_ptr, {3'b0}};
1084             //reset WT pointer
1085             //fc_wt_adr <= {wb_adr_wt_ptr, {3'b0}};
1086             //WT adr ptr remains constant
1087             //read weights and FM again
1088             //fc_op_wreq  <= 0;
1089             //fc_sel_0    <= 0;
1090             //fc_sel_1    <= 0;
1091             if(fc_count == 0) begin
1092                 c_fake    <= 0;
1093                 r_fake    <= 0;
1094                 fc_state <= 'NEXT_ADDR;
1095             end else begin
```

```
1096             fc_state <= 'NEXT_WT0_ADDR;
1097         end
1098     end
1099 end
1100 end
1101
1102     default: begin
1103         fc_state <= 'IDLE_FC;
1104     end
1105
1106 endcase
1107 end
1108 end
1109
1110
1111
1112 //=====
1113 //----- Read Weights -----
1114 //=====
1115
1116 always @(posedge clk or posedge reset_mux)
1117 begin
1118     if(reset_mux) begin
1119         nth_wt     <= 0;
1120         has_wts    <= 0;
```

```
1121     rd_wt_adr  <= 20'b0;
1122     wt_rreq    <= 1'b0;
1123
1124     wt_ch0     <= 16'b0;
1125     wt_ch1     <= 16'b0;
1126     wt_ch2     <= 16'b0;
1127     wt_ch3     <= 16'b0;
1128     wt_ch4     <= 16'b0;
1129     wt_ch5     <= 16'b0;
1130     wt_ch6     <= 16'b0;
1131     wt_ch7     <= 16'b0;
1132     wt_ch8     <= 16'b0;
1133
1134     end else begin
1135         if (conv_fc_mode || !valid_wt_base_adr) begin
1136             nth_wt    <= 0;
1137             has_wts   <= 0;
1138             rd_wt_adr <= 20'b0;
1139             wt_rreq   <= 1'b0;
1140             wt_ch0    <= 16'b0;
1141             wt_ch1    <= 16'b0;
1142             wt_ch2    <= 16'b0;
1143             wt_ch3    <= 16'b0;
1144             wt_ch4    <= 16'b0;
1145             wt_ch5    <= 16'b0;
```

```
1146     wt_ch6     <= 16'b0;
1147     wt_ch7     <= 16'b0;
1148     wt_ch8     <= 16'b0;
1149
1150     end
1151     else if (!has_relu_fm && valid_wt_base_adr) begin
1152     case (nth_wt)
1153
1154         'IDLE_WT: begin
1155             if (valid_wt_base_adr) begin
1156                 if (!has_wts) begin
1157                     // enters here the first time it's reading weights.
1158                     wt_ch0     <= 16'b0;
1159                     wt_ch1     <= 16'b0;
1160                     wt_ch2     <= 16'b0;
1161                     wt_ch3     <= 16'b0;
1162                     wt_ch4     <= 16'b0;
1163                     wt_ch5     <= 16'b0;
1164                     wt_ch6     <= 16'b0;
1165                     wt_ch7     <= 16'b0;
1166                     wt_ch8     <= 16'b0;
1167
1168                     // base address set at the beginning and is not
1169                     reset
1170
1171                     // everytime new set of weights are read.
1172                     rd_wt_adr <= {wb_adr_wt_ptr, {3'b0}};
```

```
1170         nth_wt     <= 'READ_W0;
1171     end else begin //has weights
1172         if (fm_ch_cmplt && !has_fm_1) begin //&&
1173             acc_mul_cmplt
1174             has_wts <= 1'b0;
1175             nth_wt  <= 'READ_W0;
1176         end
1177     end
1178 end
1179
1180 'READ_W0: begin
1181     //address pointer is incremented by two or one every
1182     //time based on number
1183     //of values read in at once (based of parity bit.)
1184     //parity check is done
1185     //in every state unlike in FM read State Machine
1186     //because the values are
1187     //read continuously and even & odd indices can occur in
1188     //any state .
1189     if (i_m_wb_ack) begin
1190         if (!rd_wt_adr[0]) begin
1191             //transition
1192             wt_ch0 <= i_m_wb_dat[15:0]; //w0
1193             wt_ch1 <= i_m_wb_dat[31:16]; //w1
```



```
1190         rd_wt_adr <= rd_wt_adr + 2;
1191         // >= 6 because of +2 in above line , overflow
1192         // can occur only if byte_cnt_wt >= 6
1193         nth_wt     <= 'READ_W2;
1194     end else begin
1195         // transition
1196         wt_ch0     <= i_m_wb_dat[31:16]; //w0
1197         rd_wt_adr <= rd_wt_adr + 1;
1198         nth_wt     <= 'READ_W1;
1199     end
1200 end
1201 end
1202
1203 'READ_W1: begin
1204     if (i_m_wb_ack) begin
1205         if (!rd_wt_adr[0]) begin
1206             // transition
1207             wt_ch1     <= i_m_wb_dat[15:0]; //w1
1208             wt_ch2     <= i_m_wb_dat[31:16]; //w2
1209             rd_wt_adr <= rd_wt_adr + 2;
1210             nth_wt     <= 'READ_W3;
1211         end else begin
1212             // Transition
1213             wt_ch1     <= i_m_wb_dat[31:16]; //w1
1214             rd_wt_adr <= rd_wt_adr + 1;
```

```
1215         nth_wt     <= 'READ_W2;
1216     end
1217 end
1218 end
1219
1220 'READ_W2: begin
1221     if (i_m_wb_ack) begin
1222         if (!rd_wt_adr[0]) begin
1223             // Transition
1224             wt_ch2     <= i_m_wb_dat[15:0]; //w2
1225             wt_ch3     <= i_m_wb_dat[31:16]; //w3
1226             rd_wt_adr <= rd_wt_adr + 2;
1227             nth_wt     <= 'READ_W4;
1228         end else begin
1229             // Transition
1230             wt_ch2     <= i_m_wb_dat[31:16]; //w2
1231             rd_wt_adr <= rd_wt_adr + 1;
1232             nth_wt     <= 'READ_W3;
1233         end
1234     end
1235 end
1236
1237 'READ_W3: begin
1238     if (i_m_wb_ack) begin
1239         if (!rd_wt_adr[0]) begin
```

```
1240         // Transition
1241         wt_ch3    <= i_m_wb_dat[15:0]; //w3
1242         wt_ch4    <= i_m_wb_dat[31:16]; //w4
1243         rd_wt_adr <= rd_wt_adr + 2;
1244         nth_wt    <= 'READ_W5;
1245     end else begin
1246         // Transition
1247         wt_ch3    <= i_m_wb_dat[31:16]; //w3
1248         rd_wt_adr <= rd_wt_adr + 1;
1249         nth_wt    <= 'READ_W4;
1250     end
1251 end
1252 end
1253
1254 'READ_W4: begin
1255     if(i_m_wb_ack) begin
1256         if(!rd_wt_adr[0]) begin
1257             // Transition
1258             wt_ch4 <= i_m_wb_dat[15:0]; //w4
1259             wt_ch5 <= i_m_wb_dat[31:16]; //w5
1260             rd_wt_adr <= rd_wt_adr + 2;
1261             nth_wt    <= 'READ_W6;
1262         end else begin
1263             // Transition
1264             wt_ch4    <= i_m_wb_dat[31:16]; //w4
```

```
1265         rd_wt_adr <= rd_wt_adr + 1;
1266         nth_wt     <= 'READ_W5;
1267     end
1268 end
1269 end
1270
1271 'READ_W5: begin
1272     if (i_m_wb_ack) begin
1273         if (!rd_wt_adr[0]) begin
1274             // Transition
1275             wt_ch5     <= i_m_wb_dat[15:0]; //w5
1276             wt_ch6     <= i_m_wb_dat[31:16]; //w6
1277             rd_wt_adr <= rd_wt_adr + 2;
1278             nth_wt     <= 'READ_W7;
1279         end else begin
1280             // Transition
1281             wt_ch5     <= i_m_wb_dat[31:16]; //w5
1282             rd_wt_adr <= rd_wt_adr + 1;
1283             nth_wt     <= 'READ_W6;
1284         end
1285     end
1286 end
1287
1288 'READ_W6: begin
1289     if (i_m_wb_ack) begin
```

```
1290         if (!rd_wt_adr[0]) begin
1291             // Transition
1292             wt_ch6    <= i_m_wb_dat[15:0]; //w6
1293             wt_ch7    <= i_m_wb_dat[31:16]; //w7
1294             rd_wt_adr <= rd_wt_adr + 2;
1295             nth_wt    <= 'READ_W8;
1296         end else begin
1297             // Transition
1298             wt_ch6    <= i_m_wb_dat[31:16]; //w6
1299             rd_wt_adr <= rd_wt_adr + 1;
1300             nth_wt    <= 'READ_W5;
1301         end
1302     end
1303 end
1304
1305 'READ_W7: begin
1306     if (i_m_wb_ack) begin
1307         if (!rd_wt_adr[0]) begin
1308             if (!has_wts) begin
1309                 // Transition
1310                 wt_ch7    <= i_m_wb_dat[15:0]; //w7
1311                 wt_ch8    <= i_m_wb_dat[31:16]; //w8
1312                 rd_wt_adr <= rd_wt_adr + 2;
1313                 has_wts   <= 1'b1;
1314                 nth_wt    <= 'IDLE_WT;
```

```
1315         end
1316     end else begin
1317         // Transition
1318         wt_ch7    <= i_m_wb_dat[31:16]; //w7
1319         rd_wt_adr <= rd_wt_adr + 1;
1320         nth_wt    <= 'READ_W8;
1321     end
1322     //nth_wt    <= 'IDLE_WT;
1323     //if state machine needs to idle around go to Idle
        state only.
1324 end
1325 end
1326
1327 'READ_W8: begin
1328     if(i_m_wb_ack) begin
1329         //turning has wts on one cycle before
1330         //to reduce one clock cycle latency.
1331         has_wts <= 1'b1;
1332         if(!rd_wt_adr[0]) begin
1333             if(!has_wts) begin
1334                 // Transition
1335                 wt_ch8    <= i_m_wb_dat[15:0]; //w8
1336                 rd_wt_adr <= rd_wt_adr + 1;
1337             end
1338         end else begin
```

```
1339         if (!has_wts) begin
1340             // Transition
1341             wt_ch8  <= i_m_wb_dat[31:16]; //w8
1342             rd_wt_adr <= rd_wt_adr + 1;
1343         end
1344     end
1345     nth_wt <= 'IDLE_WT;
1346 end
1347 end
1348
1349     default: begin
1350         nth_wt <= 'IDLE_WT;
1351     end
1352 endcase
1353 end
1354 end
1355 end //read weights
1356
1357 //=====
1358 //----- Read Feature maps -----
1359 //=====
1360 always @(posedge clk or posedge reset_mux)
1361 begin
1362     if(reset_mux ) begin
1363         nth_fm          <= 4'b0;
```

```
1364     row_cnt          <= 6'b0;
1365     col_cnt          <= 6'b0;
1366     rd_fm_adr        <= 14'b0;
1367     r1_fm_ptr        <= 14'b0;
1368     r2_fm_ptr        <= 14'b0;
1369     r3_fm_ptr        <= 14'b0;
1370     conv_ch_count    <= 6'b0;
1371     fm_rreq          <= 1'b0;
1372     has_fm_1         <= 1'b0;
1373     fm_ch_cmplt      <= 1'b0;
1374     fm_cmplt         <= 1'b0;
1375     all_fm_cmplt     <= 1'b0;
1376     conv_fil_count   <= 6'b0;
1377     fm_ch0           <= 16'b0;
1378     fm_ch1           <= 16'b0;
1379     fm_ch2           <= 16'b0;
1380     fm_ch3           <= 16'b0;
1381     fm_ch4           <= 16'b0;
1382     fm_ch5           <= 16'b0;
1383     fm_ch6           <= 16'b0;
1384     fm_ch7           <= 16'b0;
1385     fm_ch8           <= 16'b0;
1386
1387     end else begin
1388         if(conv_fc_mode || !valid_fm_base_adr) begin
```



```
1389     nth_fm           <= 4'b0;
1390     row_cnt          <= 6'b0;
1391     col_cnt          <= 6'b0;
1392     rd_fm_adr        <= 14'b0;
1393     r1_fm_ptr        <= 14'b0;
1394     r2_fm_ptr        <= 14'b0;
1395     r3_fm_ptr        <= 14'b0;
1396     conv_ch_count    <= 6'b0;
1397     fm_rreq          <= 1'b0;
1398     has_fm_1         <= 1'b0;
1399     fm_ch_cmplt      <= 1'b0;
1400     fm_cmplt         <= 1'b0;
1401     all_fm_cmplt     <= 1'b0;
1402     conv_fil_count   <= 6'b0;
1403     fm_ch0           <= 16'b0;
1404     fm_ch1           <= 16'b0;
1405     fm_ch2           <= 16'b0;
1406     fm_ch3           <= 16'b0;
1407     fm_ch4           <= 16'b0;
1408     fm_ch5           <= 16'b0;
1409     fm_ch6           <= 16'b0;
1410     fm_ch7           <= 16'b0;
1411     fm_ch8           <= 16'b0;
1412
1413     end
```

```
1414     else if (!conv_fc_mode && valid_fm_base_adr) begin
1415     case (nth_fm)
1416     'IDLE_FM: begin
1417         if (mp_fil_count != conv_num_fil) begin
1418             if (valid_fm_base_adr) begin //&& !has_relu_fm
1419
1420                 //should read FM values only when it has valid
1421                 weights and
1422                 // it doesn't have new FM values
1423                 if (has_wts && !has_fm_1) begin
1424                     rd_fm_adr    <= {wb_adr_fm_ptr, {3'b0}} ;
1425                     r1_fm_ptr    <= {wb_adr_fm_ptr, {3'b0}} ;
1426                     r2_fm_ptr    <= {wb_adr_fm_ptr, {3'b0}} +
1427                                 conv_ip_size_c ;
1428                     r3_fm_ptr    <= {wb_adr_fm_ptr, {3'b0}} +
1429                                 conv_ip_size_2r ;
1430                     has_fm_1    <= 1'b0;
1431                     fm_ch_cmplt <= 0;
1432                     fm_cmplt   <= 0;
1433                     all_fm_cmplt <= 0; //later change.
1434                     nth_fm      <= 'READ_FM0;
1435
1436                 end
1437             end
1438         end
1439     end
1440 end
```

```
1435     'READ_FM0: begin
1436         if (fm_ch_cmplt)
1437             fm_ch_cmplt <= 0;
1438         else begin
1439             if (has_wts) begin
1440                 if (!has_relu_fm) begin
1441                     if (i_m_wb_ack) begin
1442                         if (!rd_fm_adr[0]) begin
1443                             // Transition
1444                             fm_ch0 <= i_m_wb_dat[15:0]; //FM0
1445                             fm_ch1 <= i_m_wb_dat[31:16]; //FM1
1446                             rd_fm_adr <= rd_fm_adr +2;
1447                             nth_fm <= 'READ_FM2;
1448                         end else begin
1449                             // Transition
1450                             fm_ch0 <= i_m_wb_dat[31:16]; //FM0
1451                             rd_fm_adr <= rd_fm_adr + 1;
1452                             nth_fm <= 'READ_FM1;
1453                         end
1454                     end
1455                 end
1456             end
1457         end
1458     end
1459     'READ_FM1: begin
```

```
1460         if (i_m_wb_ack) begin
1461             if (!rd_fm_adr[0]) begin
1462                 // Transition
1463                 fm_ch1  <= i_m_wb_dat[15:0]; //FM1
1464                 fm_ch2  <= i_m_wb_dat[31:16]; //FM2
1465                 //comes to this state ONLY if ONE value was read
1466                 //in previous state. since three values will read
1467                 //from one row by the end of this state the address
1468                 //pointer should point to next row for the next
1469                 read.
1470                 rd_fm_adr <= r2_fm_ptr;
1471                 nth_fm    <= 'READ_FM3;
1472             end
1473         end
1474     'READ_FM2: begin
1475         if (i_m_wb_ack) begin
1476             //rd_fm_adr[0] should be 0 even
1477             //if in prev state it was even index,
1478             //you moved two indices at once, even
1479             //num+2 is even, so read the even
1480             //indexed value from 32 bit input
1481             //data.
1482             if (!rd_fm_adr[0]) begin
1483                 // Transition
```

```
1484         fm_ch2  <= i_m_wb_dat[15:0];    //FM2
1485         rd_fm_adr <= r2_fm_ptr; //Address pointer for next
           value to be read from row2
1486         nth_fm   <= 'READ_FM3;
1487         //can be implemneted in 3 states but requires more
           counters and complicates ->
1488         //hence implemented in 9 states.
1489     end
1490 end
1491 end
1492 'READ_FM3: begin
1493     if (i_m_wb_ack) begin
1494         if (!rd_fm_adr[0]) begin
1495             // Transition
1496             fm_ch3  <= i_m_wb_dat[15:0]; //FM3
1497             fm_ch4  <= i_m_wb_dat[31:16]; //FM4
1498             rd_fm_adr <= rd_fm_adr +2;
1499             nth_fm   <= 'READ_FM5;
1500         end else begin
1501             // Transition
1502             fm_ch3   <= i_m_wb_dat[31:16]; //FM3
1503             rd_fm_adr <= rd_fm_adr + 1;
1504             nth_fm   <= 'READ_FM4;
1505         end
1506     end
```

```
1507     end
1508     'READ_FM4: begin
1509         if (i_m_wb_ack) begin
1510             if (!rd_fm_adr[0]) begin
1511                 // Transition
1512                 fm_ch4  <= i_m_wb_dat[15:0]; //FM4
1513                 fm_ch5  <= i_m_wb_dat[31:16]; //FM5
1514                 //comes to this state ONLY if ONE value was read
1515                 //in previous state. since three values will read
1516                 //from one row by the end of this state the address
1517                 //pointer should point to next row for the next
1518                 read.
1519                 rd_fm_adr <= r3_fm_ptr;
1520                 nth_fm    <= 'READ_FM6;
1521             end
1522         end
1523     'READ_FM5: begin
1524         if (i_m_wb_ack) begin
1525             if (!rd_fm_adr[0]) begin
1526                 // Transition
1527                 fm_ch5    <= i_m_wb_dat[15:0]; //FM5
1528                 rd_fm_adr <= r3_fm_ptr; // Address pointer for next
1529                 value to be read from row2
1530                 nth_fm    <= 'READ_FM6;
```

```
1530          //can be implemneted in 3 states but requires more
           //counters and complicates ->
1531          //hence implemented in 9 states.
1532      end
1533  end
1534 end
1535 'READ_FM6: begin
1536     if (i_m_wb_ack) begin
1537         if (!rd_fm_adr[0]) begin
1538             // Transition
1539             fm_ch6    <= i_m_wb_dat[15:0]; //FM6
1540             fm_ch7    <= i_m_wb_dat[31:16]; //FM7
1541             rd_fm_adr <= rd_fm_adr +2;
1542             nth_fm    <= 'READ_FM8;
1543         end else begin
1544             // Transition
1545             fm_ch6    <= i_m_wb_dat[31:16]; //FM6
1546             rd_fm_adr <= rd_fm_adr + 1;
1547             nth_fm    <= 'READ_FM7;
1548         end
1549     end
1550 end
1551 'READ_FM7: begin
1552     if (i_m_wb_ack) begin
1553         if (!rd_fm_adr[0]) begin
```

```
1554         // Transition
1555         fm_ch7  <= i_m_wb_dat[15:0];  //FM7
1556         fm_ch8  <= i_m_wb_dat[31:16]; //FM8
1557         has_fm_1 <= 1'b1;
1558
1559         //comes to this state ONLY if ONE value was read
1560         //in previous state. since three values will read
1561         //from one row by the end of this state the address
1562         //pointer should point to next row for the next
1563         read.
1564
1565         r3_fm_ptr <= r3_fm_ptr + conv_ip_size_c;
1566         //row count and col count: IPSIZE -2-1 -> -2
1567         convoltion
1568         //and 1 cuz one iteration is complete before
1569         reaching this point.
1570         nth_fm    <= 'START_ACCMUL;
1571     end
1572 end
1573 end
1574 'READ_FM8: begin
1575     if (i_m_wb_ack) begin
1576         if (!rd_fm_adr[0]) begin
1577             // Transition
1578             fm_ch8    <= i_m_wb_dat[15:0];  //FM8
```



```
1576         has_fm_1  <= 1'b1;
1577         r3_fm_ptr <= r3_fm_ptr + conv_ip_size_c;
1578         nth_fm    <= 'START_ACCMUL;
1579         //can be implemented in 3 states but requires
1580         //more counters and complicates ->
1581         //hence implemented in 9 states.
1582     end
1583 end
1584 end
1585 'START_ACCMUL: begin
1586     if(acc_mul_cmplt) begin
1587         if (row_cnt < conv_op_size_r-1) begin
1588             has_fm_1  <= 0;
1589             row_cnt   <= row_cnt + 1;
1590             rd_fm_adr <= r3_fm_ptr; //Address pointer for next
1591                                     value to be read from row2
1592             //Transition
1593             fm_ch0    <= fm_ch3;
1594             fm_ch1    <= fm_ch4;
1595             fm_ch2    <= fm_ch5;
1596             fm_ch3    <= fm_ch6;
1597             fm_ch4    <= fm_ch7;
1598             fm_ch5    <= fm_ch8;
1599
```

```
1600         nth_fm      <= 'READ_FM6;
1601
1602     end else begin
1603         if (col_cnt < conv_op_size_c-1) begin
1604             row_cnt  <= 0;
1605             has_fm_1 <= 0;
1606             col_cnt  <= col_cnt +1;
1607
1608             //REV 3 -> transition from params to reg vals
1609             rd_fm_adr <= {wb_adr_fm_ptr, {3'b0}} +
                conv_ch_count * conv_ip_size_rc + col_cnt +1;
1610             r1_fm_ptr <= {wb_adr_fm_ptr, {3'b0}} +
                conv_ch_count * conv_ip_size_rc + col_cnt +1;
1611             r2_fm_ptr <= {wb_adr_fm_ptr, {3'b0}} +
                conv_ch_count * conv_ip_size_rc +
                conv_ip_size_c + col_cnt +1;
1612             r3_fm_ptr <= {wb_adr_fm_ptr, {3'b0}} +
                conv_ch_count * conv_ip_size_rc +
                conv_ip_size_2r+ col_cnt +1;
1613
1614         nth_fm      <= 'READ_FM0;
1615     end else begin
1616         if (conv_ch_count < conv_num_ch-1) begin
1617             col_cnt  <= 0;
1618             row_cnt  <= 0;
```

```
1619         has_fm_1      <= 0;
1620         conv_ch_count <= conv_ch_count + 1;
1621         fm_ch_cmplt   <= 1'b1;
1622
1623         //REV 3 -> transition from params to reg vals
1624         rd_fm_adr     <= {wb_adr_fm_ptr, {3'b0}} + (
                conv_ch_count+1) * conv_ip_size_rc;
1625         r1_fm_ptr     <= {wb_adr_fm_ptr, {3'b0}} + (
                conv_ch_count+1) * conv_ip_size_rc;
1626         r2_fm_ptr     <= {wb_adr_fm_ptr, {3'b0}} + (
                conv_ch_count+1) * conv_ip_size_rc +
                conv_ip_size_c;
1627         r3_fm_ptr     <= {wb_adr_fm_ptr, {3'b0}} + (
                conv_ch_count+1) * conv_ip_size_rc +
                conv_ip_size_2r;
1628
1629         nth_fm        <= 'READ_FM0;
1630
1631         end else begin
1632             if(conv_fil_count < conv_num_fil-1) begin
1633                 col_cnt      <= 0;
1634                 row_cnt      <= 0;
1635                 conv_ch_count <= 0;
1636                 has_fm_1     <= 0;
1637                 fm_ch_cmplt  <= 1'b1;
```

```
1638         fm_cmplt         <= 1'b1;
1639         conv_fil_count <= conv_fil_count + 1;
1640         nth_fm           <= 'IDLE_FM;
1641     end else begin
1642         col_cnt         <= 0;
1643         row_cnt         <= 0;
1644         has_fm_1        <= 0;
1645         conv_ch_count   <= 0;
1646         fm_ch_cmplt     <= 1'b1;
1647         fm_cmplt        <= 1'b1;
1648         conv_fil_count  <= 0;
1649         all_fm_cmplt    <= 1'b1;
1650         nth_fm          <= 'IDLE_FM;
1651     end
1652 end
1653 end
1654 end
1655 end
1656 end
1657 default: begin
1658     nth_fm <= 'IDLE_FM;
1659 end
1660 endcase
1661 end
1662 end
```

```
1663 end //read feature maps
1664
1665
1666 //=====
1667 //----- ACC-MUL State Machine -----
1668 //=====
1669
1670 always @(posedge clk or posedge reset_mux)
1671 begin
1672     if(reset_mux) begin
1673         im_conv         <= 0;
1674         wt_conv         <= 0;
1675         in1_conv        <= 0;
1676         mul_cnt         <= 0;
1677         start_bit_conv  <= 0;
1678         acc_mul_cmplt   <= 0;
1679         val_in_conv     <= 0;
1680         has_relu_fm     <= 0;
1681         curr_row        <= 0;
1682         curr_col        <= 0;
1683         mp_row          <= 0;
1684         mp_row_1p       <= 0;
1685         mp_col          <= 0;
1686         mp_col_1p       <= 0;
1687         op_wreq         <= 0;
```

```
1688     mp_cmlpt         <= 0;
1689     add_in1           <= 0;
1690     add_in2           <= 0;
1691     curr_ch           <= 0;
1692     mp1_val_1         <= 0;
1693     mp1_val_2         <= 0;
1694     mp1_val_3         <= 0;
1695     mp1_val_4         <= 0;
1696     rd_op_adr         <= 0;
1697     selx_0            <= 0;
1698     selx_1            <= 0;
1699     write_buf         <= 0;
1700     mp_fil_count      <= 0;
1701
1702     mp_iterator_c      <= 0;
1703     mp_iterator_r      <= 0;
1704
1705     for (j=0; j < MAX_OP_SIZE_R; j=j+1) begin
1706         for(k=0; k< MAX_OP_SIZE_C; k=k+1) begin
1707             conv_op[j][k] <= 0;
1708         end
1709     end
1710 end else begin
1711     if(conv_fc_mode || !valid_op_base_adr) begin
1712         im_conv         <= 0;
```

```
1713     wt_conv      <= 0;
1714     in1_conv     <= 0;
1715     mp_row       <= 0;
1716     mp_col       <= 0;
1717     mp_row_1p    <= 0;
1718     mp_col_1p    <= 0;
1719     start_bit_conv <= 0;
1720     selx_0       <= 0;
1721     selx_1       <= 0;
1722     write_buf    <= 0;
1723     mp_cmplt     <= 0;
1724     mp_fil_count <= 0;
1725     rd_op_adr    <= 0;
1726
1727     mp_iterator_c <= 0;
1728     mp_iterator_r <= 0;
1729
1730     for (j=0; j < MAX_OP_SIZE_R; j=j+1) begin
1731         for(k=0; k< MAX_OP_SIZE_C; k=k+1) begin
1732             conv_op[j][k] <= 0;
1733         end
1734     end
1735 end
1736 else if(valid_op_base_adr) begin
1737     case(mul_cnt)
```

```
1738     'ACCMUL_IDLE: begin
1739         im_conv         <= 0;
1740         wt_conv         <= 0;
1741         in1_conv        <= 0;
1742         mp_row          <= 0;
1743         mp_col          <= 0;
1744         mp_row_1p       <= 0;
1745         mp_col_1p       <= 0;
1746         start_bit_conv  <= 0;
1747         selx_0          <= 0;
1748         selx_1          <= 0;
1749         write_buf       <= 0;
1750         mp_cmplt        <= 0;
1751
1752         mp_iterator_c   <= 0;
1753         mp_iterator_r   <= 0;
1754
1755         //TODO: when to reset conv_op[][]?
1756         //reset conv_op after every filter operation.
1757         if((mp_fil_count == conv_num_fil)&& !valid_wt_base_adr)
1758             begin
1759                 mp_fil_count <= 0;
1760                 for (j=0; j < MAX_OP_SIZE_R; j=j+1) begin
1761                     for (k=0; k < MAX_OP_SIZE_C; k=k+1) begin
1762                         conv_op[j][k] <= 0;
```



```
1762         end
1763     end
1764 end
1765 if (valid_op_base_adr && (mp_fil_count < conv_num_fil))
    begin
1766     if (!valid_fm_base_adr && !valid_wt_base_adr)
1767         rd_op_adr      <= {wb_adr_op_ptr, {3'b0}}; // +
            fil_num *OP_SIZE_SQ;
1768     else if (has_wts && has_fm_1) begin
1769         im_conv        <= fm_ch0;
1770         wt_conv        <= wt_ch0;
1771         acc_mul_cmplt  <= 1'b0;
1772         mul_cnt        <= 'MUL1;
1773     end
1774 end
1775 end
1776
1777 'MUL1: begin
1778     wt_conv    <= wt_ch1;
1779     im_conv    <= fm_ch1;
1780     start_bit_conv <= 1'b1;
1781     in1_conv   <= mul_out;
1782     mul_cnt    <= 'MUL2;
1783 end
1784
```

```
1785     'MUL2: begin
1786         in1_conv <= mul_out;
1787         wt_conv  <= wt_ch2;
1788         im_conv  <= fm_ch2;
1789         mul_cnt  <= 'MUL3;
1790     end
1791
1792     'MUL3: begin
1793         in1_conv <= mul_out;
1794         im_conv  <= fm_ch3;
1795         wt_conv  <= wt_ch3;
1796         mul_cnt  <= 'MUL4;
1797     end
1798
1799     'MUL4: begin
1800         in1_conv <= mul_out;
1801         im_conv  <= fm_ch4;
1802         wt_conv  <= wt_ch4;
1803         mul_cnt  <= 'MUL5;
1804     end
1805
1806     'MUL5: begin
1807         in1_conv <= mul_out;
1808         im_conv  <= fm_ch5;
1809         wt_conv  <= wt_ch5;
```

```
1810     mul_cnt  <= 'MUL6;
1811 end
1812
1813 'MUL6: begin
1814     in1_conv <= mul_out;
1815     im_conv  <= fm_ch6;
1816     wt_conv  <= wt_ch6;
1817     mul_cnt  <= 'MUL7;
1818 end
1819
1820 'MUL7: begin
1821     in1_conv <= mul_out;
1822     im_conv  <= fm_ch7;
1823     wt_conv  <= wt_ch7;
1824     mul_cnt  <= 'MUL8;
1825 end
1826
1827 'MUL8: begin
1828     in1_conv <= mul_out;
1829     im_conv  <= fm_ch8;
1830     wt_conv  <= wt_ch8;
1831     mul_cnt  <= 'ACC7;
1832 end
1833
1834 'ACC7: begin
```

```
1835     in1_conv <= mul_out;
1836     mul_cnt  <= 'ACC8;
1837 end
1838
1839 'ACC8: begin
1840     in1_conv <= mul_out;
1841     mul_cnt  <= 'CMPLT;
1842 end
1843
1844 'CMPLT: begin
1845     start_bit_conv <= 1'b0;
1846     acc_mul_cmplt  <= 1'b1;
1847     if(acc_mul_cmplt && has_fm_1) begin
1848         //accumulating all channel convolved results
1849         //bring in your adder
1850         //acc_out is 23 bits -> add_in1 is 23
1851         add_in1  <= acc_out;
1852         //conv_op is 24 bits , add_in2 is 24 bits
1853         add_in2  <= conv_op[row_cnt][col_cnt];
1854         //collect the current r, c, ch values as they change
1855         //in FM read state machine next clock cycle.
1856         curr_row <= row_cnt;
1857         curr_col <= col_cnt;
1858         curr_ch  <= conv_ch_count;
1859         mul_cnt  <= 'ADD;
```

```
1860     end
1861 end
1862
1863 'ADD: begin
1864     if (!has_fm_1) begin
1865         if (curr_ch == conv_num_ch-1) begin
1866             //determining if all input channels are
1867             //convolved and accumulated.
1868             if (fm_ch_cmplt && fm_cmplt) begin
1869                 has_relu_fm <= 1;
1870             end
1871         end
1872     end
1873     mul_cnt <= 'ADD1;
1874 end
1875
1876 'ADD1: begin
1877     if (curr_ch == conv_num_ch-1) begin
1878         conv_op[curr_row][curr_col] <= add_out;
1879         acc_mul_cmplt <= 1'b0;
1880         mul_cnt <= 'RELU;
1881     end else begin
1882         conv_op[curr_row][curr_col] <= add_out;
1883         acc_mul_cmplt <= 1'b0;
1884         mul_cnt <= 'ACCMUL_IDLE;
```

```
1885     end
1886 end
1887
1888 'RELU: begin
1889     val_in_conv <= conv_op[curr_row][curr_col];
1890     mul_cnt     <= 'COLLECT_RELU;
1891     end
1892
1893 'COLLECT_RELU: begin
1894
1895     //figure the math for number of bits for
1896     //concatenating 0's before assignment of relu out
1897     conv_op[curr_row][curr_col] <= relu_out;
1898
1899     //what's the next state?
1900     if(has_relu_fm && !mp_flag) begin
1901         mul_cnt     <= 'WRITE_RELU;
1902     end else if(has_relu_fm && mp_flag) begin
1903         mp_row_1p <= mp_row + 1;
1904         mp_col_1p <= mp_col + 1;
1905
1906         if(conv_op_size_c[0]) begin
1907             mp_iterator_c <= conv_op_size_c - 3;
1908         end else begin
1909             mp_iterator_c <= conv_op_size_c - 2;
```

```
1910         end
1911
1912         if(conv_op_size_r[0]) begin
1913             mp_iterator_r <= conv_op_size_r - 3;
1914         end else begin
1915             mp_iterator_r <= conv_op_size_r - 2;
1916         end
1917
1918         mul_cnt <= 'MAXPOOL_1;
1919     end else begin
1920         acc_mul_cmplt <= 1'b0;
1921         mul_cnt      <= 'ACCMUL_IDLE;
1922     end
1923 end
1924
1925 'WRITE_RELU: begin
1926     // Write ReLu values to memory
1927     if (!mp_cmplt && !op_wreq) begin
1928         // if(!op_wreq) begin
1929         //rd_op_adr is set to base adr initially
1930         //Write data to buffer
1931         if(!rd_op_adr[0]) begin
1932             //even number indexed value resides in
1933             //lower significant bits of 32 bit block in MM
1934             selx_0      <= 1'b1; //4'b0011;
```

```
1935     write_buf[15:0]  <= conv_op[mp_row][mp_col];
1936     if ((mp_col < conv_op_size_c-1)) begin
1937         //if the current value is NOT the LAST value then
1938         //wait for next 16 bits to be filled before
1939             raising wreq.
1939     mp_col    <= mp_col + 1;
1940     rd_op_adr <= rd_op_adr + 1; //16 bit access
1941         address ptr.
1941     end else begin
1942         if(mp_row < conv_op_size_r-1) begin
1943             mp_col    <= 0;
1944             mp_row    <= mp_row + 1;
1945             rd_op_adr <= rd_op_adr + 1; //16 bit access
1946                 address ptr.
1946         end else begin
1947             //if the current value is the LAST value then
1948             //raise the wreq.
1949             write_buf[15:0]  <= conv_op[mp_row][mp_col];
1950             op_wreq          <= 1'b1;
1951         end
1952     end
1953     end else begin
1954         //odd number indexed value resides in MSB 16 bits
1955         of 32 bit block in MM
```



```
1955         //once MSB is written it means the buffer is full
           for current mem access or
1956         //the even indexed value was previously written.
1957         //In either case raise wreq.
1958         selx_1      <= 1'b1; //4'b1100;
1959         write_buf[31:16] <= conv_op[mp_row][mp_col];
1960         op_wreq     <= 1'b1;
1961     end
1962 end else if(!mp_cmplt && op_wreq) begin
1963     if(i_m_wb_ack) begin
1964         op_wreq     <= 1'b0;
1965         selx_0      <= 0;
1966         selx_1      <= 0;
1967         //enter here if the write is serviced,
1968         //and set index & address pointers for next value.
1969         rd_op_adr  <= rd_op_adr + 1; //16 bit access
           address ptr.
1970         //OP_SIZE2M -> rotates half number of times as it
           is reading four val's at once
1971         if(mp_col < conv_op_size_c-1) begin
1972             mp_col   <= mp_col + 1;
1973         end else begin
1974             mp_col   <= 0;
1975             if(mp_row < conv_op_size_r-1) begin //OP_SIZE2M
1976                 mp_row   <= mp_row + 1;
```

```
1977         end else begin
1978             mp_row         <= 0;
1979             mp_row_1p      <= 0;
1980             mp_cmplt      <= 1;
1981             mp_fil_count  <= mp_fil_count + 1;
1982             for (j=0; j < MAX_OP_SIZE_R; j=j+1) begin
1983                 for(k=0; k< MAX_OP_SIZE_C; k=k+1) begin
1984                     conv_op[j][k] <= 0;
1985                 end
1986             end
1987             has_relu_fm    <= 0;
1988             mul_cnt       <= 'ACCMUL_IDLE;
1989         end
1990     end
1991 end
1992 end
1993 end
1994
1995 'MAXPOOL_1: begin
1996     mp1_val_1 <= conv_op[mp_row][mp_col];           // r0 c0
1997     mul_cnt   <= 'MAXPOOL_2;
1998 end
1999
2000 'MAXPOOL_2: begin
2001     mp1_val_2 <= conv_op[mp_row][mp_col_1p];       // r0 c1
```

```
2002     mul_cnt     <= 'MAXPOOL_3;
2003 end
2004
2005 'MAXPOOL_3: begin
2006     mp1_val_3    <= conv_op[mp_row_1p][mp_col];    // r1 c0
2007     mul_cnt     <= 'MAXPOOL_4;
2008 end
2009
2010 'MAXPOOL_4: begin
2011     mp1_val_4    <= conv_op[mp_row_1p][mp_col_1p]; // r1 c1
2012     mul_cnt     <= 'WRITE_MAXPOOL;
2013 end
2014
2015 'WRITE_MAXPOOL: begin
2016     // when max pool is complete ,
2017     // mp_cmplt is off indicating new values are available .
2018     if (!mp_cmplt && !op_wreq) begin
2019         // assign values to maxpool after
2020         // write is serviced => op_wreq=0.
2021         //rd_op_adr is set to base adr initially
2022         //Write data to buffer
2023         if(!rd_op_adr[0]) begin
2024             //even number indexed value resides in
2025             //lower significant bits of 32 bit block in MM
2026             selx_0     <= 1'b1;    //4'b0011;
```

```
2027     write_buf[15:0]    <= mp1_out_1;
2028     // if ((mp_col < conv_op_size_c - 2)) begin
2029     if ((mp_col < mp_iterator_c)) begin
2030         // if the current value is NOT the LAST value then
2031         // wait for next 16 bits to be filled before
2032             raising wreq.
2033     mp_col    <= mp_col + 2;
2034     mp_col_1p <= mp_col_1p + 2;
2035     rd_op_adr <= rd_op_adr + 1; // 16 bit access
2036         address ptr.
2037     mul_cnt   <= 'MAXPOOL_1;
2038 end else begin
2039     if (mp_row < mp_iterator_r) begin
2040         mp_col    <= 0;
2041         mp_col_1p <= 1;
2042         mp_row    <= mp_row + 2;
2043         mp_row_1p <= mp_row_1p + 2;
2044         rd_op_adr <= rd_op_adr + 1; // 16 bit access
2045             address ptr.
2046         mul_cnt   <= 'MAXPOOL_1;
2047     end else begin
2048         // if the current value is the LAST value then
2049         // raise the wreq.
2050         write_buf[15:0] <= mp1_out_1;
2051         op_wreq         <= 1'b1;
```

```
2049         end
2050     end
2051 end else begin
2052     //odd number indexed value resides in MSB 16 bits
2053     //of 32 bit block in MM
2054     //once MSB is written it means the buffer is full
2055     //for current mem access or
2056     //the even indexed value was previously written.
2057     //In either case raise wreq.
2058     selx_1      <= 1'b1; //4'b1100;
2059     write_buf[31:16] <= mp1_out_1;
2060     op_wreq     <= 1'b1;
2061 end
2062 end else if(!mp_cmplt && op_wreq) begin
2063     if(i_m_wb_ack) begin
2064         op_wreq     <= 1'b0;
2065         selx_0      <= 0;
2066         selx_1      <= 0;
2067         //enter here if the write is serviced,
2068         //and set index & address pointers for next value.
2069         rd_op_adr  <= rd_op_adr + 1; //16 bit access
2070         address ptr.
2071         //OP_SIZE2M -> rotates half number of times as it
2072         //is reading four val's at once
2073         if(mp_col < mp_iterator_c) begin
```

```
2070         mp_col      <= mp_col + 2;
2071         mp_col_1p <= mp_col_1p + 2;
2072         mul_cnt     <= 'MAXPOOL_1;
2073     end else begin
2074         mp_col      <= 0;
2075         mp_col_1p <= 1;
2076         if(mp_row < mp_iterator_r) begin //OP_SIZE2M
2077             mp_row      <= mp_row + 2;
2078             mp_row_1p   <= mp_row_1p + 2;
2079             mul_cnt     <= 'MAXPOOL_1;
2080         end else begin
2081             mp_row      <= 0;
2082             mp_row_1p   <= 0;
2083             mp_cmplt    <= 1;
2084             mp_fil_count <= mp_fil_count + 1;
2085             //Reset conv_op after every filter operation.
2086             for (j=0; j < MAX_OP_SIZE_R; j=j+1) begin
2087                 for(k=0; k< MAX_OP_SIZE_C; k=k+1) begin
2088                     conv_op[j][k] <= 0;
2089                 end
2090             end
2091             has_relu_fm <= 0;
2092             mul_cnt     <= 'ACCMUL_IDLE;
2093         end
2094     end
```

```
2095         end
2096     end
2097 end
2098
2099     default:
2100         mul_cnt <= 'ACCMUL_IDLE;
2101     endcase
2102 end
2103 end
2104 end //ACC MUL
2105
2106
2107 //
2108 // *****
2109 //           Wishbone Interface
2110 // *****
2111
2112 assign start_write = i_s_wb_stb && i_s_wb_we && !start_read_1;
2113 assign start_read  = i_s_wb_stb && !i_s_wb_we && !o_s_wb_ack;
2114 always @( posedge reset_mux or posedge clk )
2115     if (reset_mux)
2116         start_read_1 <= 1'b0;
```

```
2116     else
2117         start_read_1 <= start_read;
2118
2119     assign o_s_wb_err = 1'd0;
2120     assign o_s_wb_ack = i_s_wb_stb && ( start_write || start_read_1
2121         );
2122
2123     generate
2124     if (WB_DWIDTH == 128)
2125         begin : wb128
2126             assign wb_wdata32 = i_s_wb_adr[3:2] == 2'd3 ? i_s_wb_dat
2127                 [127:96] :
2128                 i_s_wb_adr[3:2] == 2'd2 ? i_s_wb_dat[
2129                     95:64] :
2130                 i_s_wb_adr[3:2] == 2'd1 ? i_s_wb_dat[
2131                     63:32] :
2132                 i_s_wb_dat[
2133                     31: 0] ;
2134
2135             assign o_s_wb_dat = {4{wb_rdata32}};
2136         end
2137     else
2138         begin : wb32
2139             assign wb_wdata32 = i_s_wb_dat;
2140             assign o_s_wb_dat = wb_rdata32;
```



```
2136     end
2137 endgenerate
2138
2139
2140 //
                *****
2141 //     Register writes
2142 //
                *****
2143
2144 always @(posedge clk or posedge reset_mux)
2145 begin
2146     if (reset_mux == 1'b1) begin
2147         sw_reset <= 1'b0;
2148         wb_adr_cnfg_ptr    <= 11'b0;
2149         wb_adr_cnfg_base_ptr <= 13'b0;
2150         start_bit    <= 1'b0;
2151     end else begin
2152         if ( start_write ) begin
2153             case ( i_s_wb_adr[15:0] )
2154                 ACNN_SRESET      : sw_reset          <=
                wb_wdata32[0];      // 16'h0000
```

```

2155     ACNN_LAYER_CONFIG: { start_bit , wb_adr_cnfg_base_ptr ,
                          wb_adr_cnfg_ptr } <= wb_wdata32 [24:0]; // , mp_flag ,
                          conv_fc_mode ,
2156     endcase
2157 end
2158 end
2159 end
2160
2161
2162 //
                *****
2163 //     Register reads
2164 //
                *****
2165
2166 always @(posedge clk or posedge reset_mux)
2167 begin
2168     if (reset_mux == 1'b1) begin
2169         wb_rdata32 <= 32'h00COFFEE;
2170     end else begin
2171         if ( start_read ) begin
2172             case ( i_s_wb_adr [15:0] )
2173                 ACNN_SRESET      : wb_rdata32 <= {31'b0, sw_reset };

```

```
2174     ACNN_WT_BASE_ADR: wb_rdata32 <= {8'b0 ,
        wb_adr_wt_base_ptr };
2175     ACNN_FM_BASE_ADR: wb_rdata32 <= {8'b0 ,
        wb_adr_fm_base_ptr };
2176     ACNN_OP_BASE_ADR: wb_rdata32 <= {8'b0 ,
        wb_adr_fm_base_ptr };
2177     ACNN_WT_VALADR  : wb_rdata32 <= {31'b0 ,
        valid_wt_base_adr };
2178     ACNN_FM_VALADR  : wb_rdata32 <= {31'b0 ,
        valid_fm_base_adr };
2179     ACNN_OP_VALADR  : wb_rdata32 <= {31'b0 ,
        valid_op_base_adr };
2180     ACNN_WT_CUR_PTR  : wb_rdata32 <= {19'b0 , rd_wt_adr
        [13:0]};
2181     ACNN_FM_CUR_PTR  : wb_rdata32 <= {19'b0 , rd_fm_adr
        [13:0]};
2182     ACNN_OP_CUR_PTR  : wb_rdata32 <= {19'b0 , rd_op_adr
        [13:0]};
2183     ACNN_FM_CUR_CH   : wb_rdata32 <= {26'b0 , conv_ch_count };
2184     ACNN_CUR_FIL     : wb_rdata32 <= {26'b0 , conv_fil_count };
2185     ACNN_WT_STATUS   : wb_rdata32 <= {31'b0 , has_wts };
2186     ACNN_FM_STATUS   : wb_rdata32 <= {31'b0 , has_fm_1 };
2187     ACNN_RELU_STATUS : wb_rdata32 <= {31'b0 , has_relu_fm };
2188     ACNN_MP_STATUS   : wb_rdata32 <= {31'b0 , mp_cmplt };
2189     ACNN_CUR_ADR     : wb_rdata32 <= curr_adr ;
```

```
2190     ACNN_COMPLETE    :wb_rdata32 <= {26'b0, mp_fil_count};
2191     ACNN_FC_DONE      :wb_rdata32 <= {31'b0, fc_done};
2192     ACNN_LAYER_CMPLT :wb_rdata32 <= {28'b0, layer_count};
2193     ACNN_DONE         :wb_rdata32 <= {31'b0, ACNN_done};
2194         default       :wb_rdata32 <= 32'h00C0FFEE;
2195
2196         endcase
2197     end
2198 end
2199 end
2200
2201
2202 endmodule // ACNN
```

Listing I.13: ACNN

I.14 SystemVerilog Test-bench Interface

```
1 interface intf_dut(input clk);
2
3 parameter BIT_WIDTH=16, NUM_LAYERS=8, MAX_IP_SIZE_R=64,
   MAX_IP_SIZE_C=64, MAX_OP_SIZE_R = 62, MAX_OP_SIZE_C = 62;
4 parameter WB_DWIDTH=32, WB_SWIDTH=4, COMPARE_3 =9830, COMPARE_8
   = 26213;
5
6 var logic scan_out0 , scan_out1 , scan_out2
   ;
7 var logic reset;
8 var logic scan_in0 , scan_en ,
9   test_mode;
10 var logic [31:0] icnn_o_m_wb_adr;
11 var logic [WB_SWIDTH-1:0] icnn_o_m_wb_sel;
12 var logic icnn_o_m_wb_stb;
13 var logic icnn_i_m_wb_ack ,
14   icnn_i_m_wb_err;
15 var logic icnn_o_m_wb_we ,
16   icnn_o_m_wb_cyc;
17 var logic [WB_DWIDTH-1:0] icnn_o_m_wb_dat;
18 var logic icnn_o_s_wb_ack ,
19   icnn_o_s_wb_err;
20 var logic [31:0] icnn_i_s_wb_adr;
```

```
21 var logic [WB_SWIDTH-1:0] icnn_i_s_wb_sel;
22 var logic      icnn_i_s_wb_we;
23 var logic [WB_DWIDTH-1:0] icnn_i_s_wb_dat;
24 var logic [WB_DWIDTH-1:0] icnn_o_s_wb_dat;
25 var logic [WB_DWIDTH-1:0] icnn_i_m_wb_dat;
26 var logic      icnn_i_s_wb_cyc ,
27               icnn_i_s_wb_stb;
28
29 var logic  [31:0]      acnn_o_m_wb_adr;
30 var logic [WB_SWIDTH-1:0] acnn_o_m_wb_sel;
31 var logic      acnn_o_m_wb_stb;
32 var logic      acnn_i_m_wb_ack ,
33               acnn_i_m_wb_err;
34 var logic      acnn_o_m_wb_we ,
35               acnn_o_m_wb_cyc;
36 var logic [WB_DWIDTH-1:0] acnn_o_m_wb_dat;
37 var logic      acnn_o_s_wb_ack ,
38               acnn_o_s_wb_err;
39 var logic [31:0]      acnn_i_s_wb_adr;
40 var logic [WB_SWIDTH-1:0] acnn_i_s_wb_sel;
41 var logic      acnn_i_s_wb_we;
42 var logic [WB_DWIDTH-1:0] acnn_i_s_wb_dat;
43 var logic [WB_DWIDTH-1:0] acnn_o_s_wb_dat;
44 var logic [WB_DWIDTH-1:0] acnn_i_m_wb_dat;
45 var logic      acnn_i_s_wb_cyc ,
```

```
46             acnn_i_s_wb_stb ;
47
48 var logic     mm_o_s_wb_ack ,
49             mm_o_s_wb_err ;
50 var logic [31:0]     mm_i_s_wb_adr ;
51 var logic [WB_SWIDTH-1:0] mm_i_s_wb_sel ;
52 var logic     mm_i_s_wb_we ;
53 var logic [WB_DWIDTH-1:0] mm_i_s_wb_dat ;
54 var logic [WB_DWIDTH-1:0] mm_o_s_wb_dat ;
55 var logic     mm_i_s_wb_cyc ,
56             mm_i_s_wb_stb ;
57
58 var logic     [WB_SWIDTH-1:0] fake_master_sel ;
59 var logic [31:0]     fake_master_adr ;
60 var logic     fake_master_we ;
61 var logic [WB_DWIDTH-1:0] fake_master_data ;
62 var logic     fake_master_cyc ;
63 var logic     fake_master_stb ;
64 var logic     fake_master_ack ;
65
66 endinterface
```

Listing I.14: SystemVerilog Test-bench Interface

I.15 Driver

```
1
2 'include " ../include/register_addresses.vh"
3
4
5 class driver;
6
7     virtual intf_dut intf;
8     scoreboard sb;
9
10    function new(virtual intf_dut intf , scoreboard sb);
11        this.intf = intf;
12        this.sb    = sb;
13    endfunction
14
15    integer k;
16    var reg [31:0] w_data;
17
18
19    task reset();
20        k= 0;
21        intf.reset = 1'b0;
22            intf.scan_in0 = 1'b0;
23            intf.scan_en = 1'b0;
```



```
24         intf.test_mode = 1'b0;
25         intf.fake_master_sel  = 4'b0;
26         intf.fake_master_cyc  = 1'b0;
27     intf.fake_master_stb  = 1'b0;
28         intf.fake_master_adr  = 32'b0;
29         intf.fake_master_we    = 1'b0;
30         intf.fake_master_data = 32'b0;
31
32     sb.fm_r_size = 16'b0;
33         sb.fm_c_size = 16'b0;
34         sb.fm_depth  = 8'b0;
35         sb.fm_adr    = 32'b0;
36
37     sb.num_layers  = 16'b0;
38     sb.label       = 8'b0;
39     sb.num_sample  = 8'b0;
40     sb.mp_status   = 32'b0;
41
42     sb.wt_file_path = 0;
43     sb.fm_file_path = 0;
44
45     sb.layer_config_addrss = 0;
46     for(k=0; k<8; k=k+1) begin
47         sb.wt_r_size[k]    = 16'b0;
48         sb.wt_c_size[k]    = 16'b0;
```

```
49
50         sb.w_adr[k]          = 32'b0;
51
52         sb.layer_wt_addrss[k] = 32'b0;
53         sb.layer_ip_addrss[k] = 32'b0;
54         sb.layer_op_addrss[k] = 32'b0;
55
56         sb.mp_cmp_conv_fc[k]  = 8'b0;
57         sb.conv_num_fil[k]    = 8'b0;
58         sb.conv_num_ch[k]     = 8'b0;
59         sb.conv_ip_size_2r[k] = 16'b0;
60         sb.conv_ip_size_r[k]  = 16'b0;
61         sb.conv_ip_size_c[k]  = 16'b0;
62
63         sb.conv_op_size_r[k]  = 8'b0;
64         sb.conv_op_size_c[k]  = 8'b0;
65         sb.conv_mp_size_r[k]  = 8'b0;
66         sb.conv_mp_size_c[k]  = 8'b0;
67         sb.conv_ip_size_rc[k] = 8'b0;
68         sb.conv_op_size_rc[k] = 8'b0;
69     end
70
71     for (k=0; k<3; k++) begin
72         sb.mntr_mp_cmod[k] = 0;
73         sb.mntr_mp_dut[k]  = 0;
```

```
74     end
75
76     w_data = 32'b0;
77     @(posedge intf.clk);
78     intf.reset = 1'b1;
79     #100;
80     @(posedge intf.clk)
81     intf.reset = 1'b0;
82     #100;
83 endtask
84
85 task set_constants ();
86     // writes values to memory
87     sb.num_layers = 8;
88     sb.num_sample = 1;
89     sb.fm_depth    = 3;
90     sb.fm_r_size  = 64;
91     sb.fm_c_size  = 64;
92     sb.fm_adr     = 32'h0_19FFFFFF; // 32'h0_2000000
93
94     sb.wt_r_size[0] = 3;
95     sb.wt_r_size[1] = 3;
96     sb.wt_r_size[2] = 3;
97     sb.wt_r_size[3] = 3;
98     sb.wt_r_size[4] = 1024;
```

```
99         sb.wt_r_size[5] = 512;
100        sb.wt_r_size[6] = 256;
101        sb.wt_r_size[7] = 128;
102
103        sb.wt_c_size[0] = 3;
104        sb.wt_c_size[1] = 3;
105        sb.wt_c_size[2] = 3;
106        sb.wt_c_size[3] = 3;
107        sb.wt_c_size[4] = 512;
108        sb.wt_c_size[5] = 256;
109        sb.wt_c_size[6] = 128;
110        sb.wt_c_size[7] = 4;
111
112        //WEIGHT BASE ADDRESS LAYERS
113        sb.w_adr[0] = 32'h0_0FFFFFF; //LAYER0
114        sb.w_adr[1] = 32'h0_10FFFFFF; //LAYER1
115        sb.w_adr[2] = 32'h0_11FFFFFF; //LAYER2
116        sb.w_adr[3] = 32'h0_12FFFFFF; //LAYER3
117        sb.w_adr[4] = 32'h0_13FFFFFF; //LAYER4
118        sb.w_adr[5] = 32'h0_16FFFFFF; //LAYER5
119        sb.w_adr[6] = 32'h0_17FFFFFF; //LAYER6
120        sb.w_adr[7] = 32'h0_18FFFFFF; //LAYER7
121
122        sb.wt_file_path = "../.. / Python/ICNN/results/weights /
        fxpt_16/weights_0.5.csv";
```

```
123         sb.fm_file_path = "../.. / Python / ICNN / datasets / csv / test /
           fxpt_16 / test.csv";
124
125         // Convolution
126
127         // 0, MP flag ,cmp flag ,conv_fc_mode
128         sb.mp_cmp_conv_fc[0] = 8'h04; // 32'h06_500000; //0000 -> 4
129         sb.mp_cmp_conv_fc[1] = 8'h04; // 32'h06_500001; //0110 -> 6
130         sb.mp_cmp_conv_fc[2] = 8'h04; // 32'h06_500002; //0110 -> 6
131         sb.mp_cmp_conv_fc[3] = 8'h00; // 32'h04_500003; //0100 -> 4
132         sb.mp_cmp_conv_fc[4] = 8'h01; // 32'h05_500004; //0101 -> 5
133         sb.mp_cmp_conv_fc[5] = 8'h01; // 32'h05_500005; //0101 -> 5
134         sb.mp_cmp_conv_fc[6] = 8'h01; // 32'h05_500006; //0101 -> 5
135         sb.mp_cmp_conv_fc[7] = 8'h03; // 32'h05_500007; //0101 -> 5
136
137         sb.conv_num_fil[0] = 8'd8; // conv
138         sb.conv_num_fil[1] = 8'd16; // conv
139         sb.conv_num_fil[2] = 8'd32; // conv
140         sb.conv_num_fil[3] = 8'd64; // conv
141         sb.conv_num_fil[4] = 8'd64; //num fils of last conv
           layer
142         sb.conv_num_fil[5] = 8'd1; // conv
143         sb.conv_num_fil[6] = 8'd1; // conv
144         sb.conv_num_fil[7] = 8'd1; // conv
145
```

```
146     sb.conv_num_ch[0] = 8'd3; // conv
147     sb.conv_num_ch[1] = 8'd8; // conv
148     sb.conv_num_ch[2] = 8'd16; // conv
149     sb.conv_num_ch[3] = 8'd32; // conv
150     sb.conv_num_ch[4] = 8'd4; // op size_c of last conv layer
151         sb.conv_num_ch[5] = 8'd1; // conv
152         sb.conv_num_ch[6] = 8'd1; // conv
153         sb.conv_num_ch[7] = 8'd1; // conv
154
155
156     sb.conv_ip_size_2r[0] = 8'h80; // 8'd128;
157     sb.conv_ip_size_2r[1] = 8'h3E; // 8'd62;
158     sb.conv_ip_size_2r[2] = 8'h1C; // 8'd28;
159     sb.conv_ip_size_2r[3] = 8'h0C; // 8'd12;
160     sb.conv_ip_size_2r[4] = 8'd4; // op size_r of last conv
161         layer
162     sb.conv_ip_size_2r[5] = 8'd0;
163     sb.conv_ip_size_2r[6] = 8'd0;
164     sb.conv_ip_size_2r[7] = 8'd0;
165
166     // Conv
167     sb.conv_ip_size_r[0] = 16'h0040; // 16'd64;
168     sb.conv_ip_size_r[1] = 16'h001F; // 16'd31;
169     sb.conv_ip_size_r[2] = 16'h000E; // 16'd14;
170     sb.conv_ip_size_r[3] = 16'h0006; // 16'd6;
```

```
170     // Fully connected
171     sb.conv_ip_size_r[4] = 16'd1024;
172     sb.conv_ip_size_r[5] = 16'd512;
173     sb.conv_ip_size_r[6] = 16'd256;
174     sb.conv_ip_size_r[7] = 16'd128;
175
176     // Conv
177     sb.conv_ip_size_c[0] = 16'h0040; // 16'd64;
178     sb.conv_ip_size_c[1] = 16'h001F; // 16'd31;
179     sb.conv_ip_size_c[2] = 16'h000E; // 16'd14;
180     sb.conv_ip_size_c[3] = 16'h0006; // 16'd6;
181     // Fully Connected
182     sb.conv_ip_size_c[4] = 16'd512;
183     sb.conv_ip_size_c[5] = 16'd256;
184     sb.conv_ip_size_c[6] = 16'd128;
185     sb.conv_ip_size_c[7] = 16'd4;
186
187     //OUTPUT SIZE_R
188     sb.conv_op_size_r[0] = 8'h3E; // 8'd62;
189     sb.conv_op_size_r[1] = 8'h1D; // 8'd29;
190     sb.conv_op_size_r[2] = 8'h0C; // 8'd12;
191     sb.conv_op_size_r[3] = 8'h04; // 8'd4;
192
193     //OUTPUT SIZE_C
194     sb.conv_op_size_c[0] = 8'h3E; // 8'd62;
```

```
195     sb.conv_op_size_c[1] = 8'h1D; // 8'd29;
196     sb.conv_op_size_c[2] = 8'h0C; // 8'd12;
197     sb.conv_op_size_c[3] = 8'h04; // 8'd4;
198
199     //MP SIZE_R
200         sb.conv_mp_size_r[0] = 8'h1F; // 8'd31;
201     sb.conv_mp_size_r[1] = 8'h0E; // 8'd14;
202     sb.conv_mp_size_r[2] = 8'h08; // 8'd6;
203     sb.conv_mp_size_r[3] = 8'h00; // 8'd0;
204
205         //MP SIZE_C
206         sb.conv_mp_size_c[0] = 8'h1F; // 8'd31;
207     sb.conv_mp_size_c[1] = 8'h0E; // 8'd14;
208     sb.conv_mp_size_c[2] = 8'h08; // 8'd6;
209     sb.conv_mp_size_c[3] = 8'h00; // 8'd0;
210
211     //TOT NEURONS
212     sb.conv_ip_size_rc[0] = 16'h1000; // 16'd4096;
213     sb.conv_ip_size_rc[1] = 16'h03C1; // 16'd961;
214     sb.conv_ip_size_rc[2] = 16'h00C4; // 16'd196;
215     sb.conv_ip_size_rc[3] = 16'h0024; // 16'd36;
216
217     //TOT NEURONS
218     sb.conv_op_size_rc[0] = 16'h0F04; // 16'd3844; //62 x62
219     sb.conv_op_size_rc[1] = 16'h0349; // 16'd841;
```



```
220     sb.conv_op_size_rc[2] = 16'h0090; //16'd144;
221     sb.conv_op_size_rc[3] = 16'h0010; //16'd16;
222
223     //WEIGHT BASE ADDRESS LAYERS
224     sb.layer_wt_addrss[0] = 32'h00_100000; //LAYER0
225     sb.layer_wt_addrss[1] = 32'h00_110000; //LAYER1
226     sb.layer_wt_addrss[2] = 32'h00_120000; //LAYER2
227     sb.layer_wt_addrss[3] = 32'h00_130000; //LAYER3
228     sb.layer_wt_addrss[4] = 32'h00_140000; //LAYER4
229     sb.layer_wt_addrss[5] = 32'h00_170000; //LAYER5
230     sb.layer_wt_addrss[6] = 32'h00_180000; //LAYER6
231     sb.layer_wt_addrss[7] = 32'h00_190000; //LAYER7
232
233     //I/P FM BASE ADDRESS LAYERS
234     sb.layer_ip_addrss[0] = 32'h00_1A0000; //LAYER0
235     sb.layer_ip_addrss[1] = 32'h00_210000; //LAYER1
236     sb.layer_ip_addrss[2] = 32'h00_220000; //LAYER2
237     sb.layer_ip_addrss[3] = 32'h00_230000; //LAYER3
238     sb.layer_ip_addrss[4] = 32'h00_240000; //LAYER4
239     sb.layer_ip_addrss[5] = 32'h00_250000; //LAYER5
240     sb.layer_ip_addrss[6] = 32'h00_260000; //LAYER6
241     sb.layer_ip_addrss[7] = 32'h00_270000; //LAYER7
242
243     //O/P FM BASE ADDRESS LAYERS
244     sb.layer_op_addrss[0] = 32'h00_210000; //LAYER0
```

```
245     sb.layer_op_addrss[1] = 32'h00_220000; //LAYER1
246     sb.layer_op_addrss[2] = 32'h00_230000; //LAYER2
247     sb.layer_op_addrss[3] = 32'h00_240000; //LAYER3
248     sb.layer_op_addrss[4] = 32'h00_250000; //LAYER4
249     sb.layer_op_addrss[5] = 32'h00_260000; //LAYER5
250     sb.layer_op_addrss[6] = 32'h00_270000; //LAYER6
251     sb.layer_op_addrss[7] = 32'h00_280000; //LAYER7
252
253     //CONFIG ADDRESS MEMORY
254     sb.layer_config_addrss = 32'h0_3000000; //LAYER CONFIG
        ADDRSS
255
256     //Op addresses for Task in Monitor
257     sb.layer_dest_addrss[0] = 32'h0_2100000;
258     sb.layer_dest_addrss[1] = 32'h0_2200000;
259     sb.layer_dest_addrss[2] = 32'h0_2300000;
260     sb.layer_dest_addrss[3] = 32'h0_2400000;
261     sb.layer_dest_addrss[4] = 32'h0_2500000;
262     sb.layer_dest_addrss[5] = 32'h0_2600000;
263     sb.layer_dest_addrss[6] = 32'h0_2700000;
264     sb.layer_dest_addrss[7] = 32'h0_2800000;
265
266
267     endtask
268
```

```
269 task write_weights();
270     integer k;
271     // All Layers
272     $display("Weights:");
273     for(k=0; k<sb.num_layers; k=k+1) begin
274         // One fake read
275         // Every row starts with letter w1-8
276         w_data = read_csv_1(sb.wt_file_path, k);
277         get_val_and_write_to_mem(sb.conv_num_fil[k],
278             sb.conv_num_ch[k],
279             sb.wt_r_size[k],
280             sb.wt_c_size[k],
281             sb.w_adr[k], 1, sb.mp_cmp_conv_fc[k]);
282     end
283
284 endtask
285
286 task write_image();
287     input integer img_num;
288     $display("Image: %d", img_num);
289     sb.label = read_csv_3(sb.fm_file_path, img_num);
290     get_val_and_write_to_mem(sb.num_sample,
291         sb.fm_depth,
292         sb.fm_r_size,
293         sb.fm_c_size,
```

```
294             sb.fm_adr, 0, sb.mp_cmp_conv_fc[0]);
295     endtask
296
297     task write_start_bit();
298         write_to_slave({32'h01_300000}, {16'h3300,
299             ICNN_LAYER_CONFIG});
300
301     task layer_config_mem();
302     input [5:0] num_layers;
303         // Writing to config space
304     integer i;
305     var reg [31:0] addr_ptr;
306     addr_ptr = 0;
307     addr_ptr = sb.layer_config_addrss[31:0];
308     //LAYER 1
309     for (i=0;i<num_layers; i=i+1) begin
310         if(i<4) begin
311             write_to_slave( {{sb.conv_num_fil[i]}, {sb.conv_num_ch
312                 [i]}},
313                 {sb.conv_ip_size_2r[i]}, {sb.mp_cmp_conv_fc[i]}},
314                 addr_ptr); // layer 0
315
316             // write_to_slave( {{8'h3}, {8'h3}, {8'd24}, {8'h4}},
317                 addr_ptr); // layer 0
318
319             addr_ptr = {{addr_ptr[31:2] + 1}, {2'b0}};
```

```
315     write_to_slave( {{sb.conv_ip_size_r[i]}, {sb.
           conv_ip_size_c[i]}}, addr_ptr);
316     // write_to_slave( {{16'd12}, {16'd12}}, addr_ptr);
317     addr_ptr      = {{addr_ptr[31:2] + 1},{2'b0}};
318     write_to_slave( {{sb.conv_op_size_r[i]}, {sb.
           conv_op_size_c[i]},
319           {sb.conv_mp_size_r[i]}, {sb.conv_mp_size_c[i]}},
           addr_ptr);
320     // write_to_slave( {{8'd10}, {8'd10}, {8'd5},{8'd5}},
           addr_ptr);
321     addr_ptr      = {{addr_ptr[31:2] + 1},{2'b0}};
322     write_to_slave( {{sb.conv_ip_size_rc[i]}, {sb.
           conv_op_size_rc[i]}}, addr_ptr);
323     // write_to_slave( {{16'd144},{16'd100}}, addr_ptr);
324     addr_ptr      = {{addr_ptr[31:2] + 1},{2'b0}};
325     write_to_slave( sb.layer_wt_addrss[i], addr_ptr);
326     addr_ptr      = {{addr_ptr[31:2] + 1},{2'b0}};
327     write_to_slave( sb.layer_ip_addrss[i], addr_ptr);
328     addr_ptr      = {{addr_ptr[31:2] + 1},{2'b0}};
329     write_to_slave( sb.layer_op_addrss[i], addr_ptr);
330     addr_ptr      = {{addr_ptr[31:2] + 1},{2'b0}};
331     end else begin
332     write_to_slave( {{sb.conv_num_fil[i]}, {sb.conv_num_ch
           [i]}},
```

```
333         {sb.conv_ip_size_2r[i]}, {sb.mp_cmp_conv_fc[i]}},
           addr_ptr); // layer 0
334 // write_to_slave( {8'b0,8'b0,8'b0,{8'h1}}, addr_ptr);
335 addr_ptr      = {{addr_ptr[31:2] + 1},{2'b0}};
336 write_to_slave( {{sb.conv_ip_size_r[i]}, {sb.
           conv_ip_size_c[i]}}, addr_ptr);
337 addr_ptr      = {{addr_ptr[31:2] + 1},{2'b0}};
338 write_to_slave( sb.layer_wt_addrss[i], addr_ptr);
339 addr_ptr      = {{addr_ptr[31:2] + 1},{2'b0}};
340 write_to_slave( sb.layer_ip_addrss[i], addr_ptr);
341 addr_ptr      = {{addr_ptr[31:2] + 1},{2'b0}};
342 write_to_slave( sb.layer_op_addrss[i], addr_ptr);
343 addr_ptr      = {{addr_ptr[31:2] + 1},{2'b0}};
344     end
345 end
346 endtask
347
348
349 task stop_icnn();
350 begin
351     var reg [31:0] data;
352     data = 0;
353     while(!data[0]) begin
354         read_from_slave({16'h3300, ICNN_DONE}, data);
355     end
```

```
356     if (data [0]) begin
357         write_to_slave ({32'h00_300000}, {16'h3300,
                ICNN_LAYER_CONFIG});
358     end
359 end
360 endtask
361
362 task stop_icnn_layer();
363 input [3:0] nth_layer;
364
365     var reg [31:0] data;
366     data = 0;
367     while (data [3:0] != nth_layer) begin
368         read_from_slave ({16'h3300, ICNN_LAYER_CMPLT}, data);
369     end
370     if (data [0]) begin
371         write_to_slave ({32'h00_300000}, {16'h3300,
                ICNN_LAYER_CONFIG});
372     end
373 endtask
374
375 task get_val_and_write_to_mem();
376     input [15:0] num_fil;
377     input [15:0] depth;
378     input [15:0] row_size;
```

```
379     input [15:0] col_size;
380     input [31:0] addrs;
381     input fm_wt_flag;
382     input [7:0] conv_fc_flag;
383
384     integer k, m, params;
385
386     var reg [31:0] data;
387     if(!conv_fc_flag[0])
388         params = (num_fil*depth*row_size*col_size)/2;
389     else
390         params = (row_size*col_size)/2;
391
392     if(fm_wt_flag) begin
393         $display("r: %d, c: %d, params: %d", row_size,
394                 col_size, params);
395         // layer 1 -> 216 parameters/2 = 108
396         for(k = 0; k < params; k = k + 1) begin
397             // Read two values from csv and concatenate
398             data[15:0] = read_csv_1(sb.wt_file_path, k+1);
399             data[31:16] = read_csv_1(sb.wt_file_path, k+1);
400             addrs = {{ addrs[31:2] + 1},{2'b0}};
401             // Write values to mem
402             write_to_slave(data, addrs);
```



```
403         end
404     end else begin
405         //layer 1 -> 216 parameters/2 = 108
406         for(k = 0; k < params; k = k + 1) begin
407             // Read two values from csv and concatenate
408             data[15:0] = read_csv_3(sb.fm_file_path, k+1);
409             data[31:16] = read_csv_3(sb.fm_file_path, k+1);
410             addrs = {{ addrs[31:2] + 1},{2'b0}};
411             // Write values to mem
412             write_to_slave(data, addrs);
413             // Write values to sb.tot_neus
414             //sb.tot_neurons[m] = data[15:0];
415             //sb.tot_neurons[m+1] = data[31:16];
416             //m = m+2;
417         end
418     end
419 endtask
420
421 task write_to_slave();
422 input [31:0] data; // data to be written to a reg.
423 input [31:0] addrs; // address of the reg to access.
424 begin
425     intf.fake_master_cyc = 1'b1;
426     intf.fake_master_stb = 1'b1;
427     intf.fake_master_we = 1'b1;
```

```
428     intf.fake_master_adr = addr;
429     intf.fake_master_data = data;
430     intf.fake_master_sel = 4'hF;
431     @(posedge intf.fake_master_ack);
432     @(posedge intf.clk);
433     intf.fake_master_cyc = 1'b0;
434     intf.fake_master_stb = 1'b0;
435     intf.fake_master_we = 1'b0;
436     intf.fake_master_sel = 4'b0;
437     @(posedge intf.clk);
438 end
439 endtask
440
441 task read_from_slave();
442     input [31:0] addr;
443     output [31:0] data;
444     begin
445         intf.fake_master_cyc = 1'b1;
446         intf.fake_master_stb = 1'b1;
447         intf.fake_master_we = 1'b0;
448         intf.fake_master_adr = addr;
449         intf.fake_master_sel = 4'h0;
450         @(posedge intf.fake_master_ack);
451         //@(posedge intf.clk);
452         data = intf.icnn_o_s_wb_dat;
```

```
453     // $display(" read_from_slave data[0] = %d", data[0]);
454     intf.fake_master_cyc = 1'b0;
455     intf.fake_master_stb = 1'b0;
456     intf.fake_master_we  = 1'b0;
457     @(posedge intf.clk);
458     end
459     endtask
460
461 endclass
```

Listing I.15: SystemVerilog Test-bench Driver

I.16 Monitor Class

```
1 class monitor;
2
3   virtual intf_dut intf;
4   scoreboard sb;
5
6   function new(virtual intf_dut intf , scoreboard sb);
7       this.intf = intf;
8       this.sb   = sb;
9   endfunction
10
11   task compare_lyr();
12       input  [31:0]  addr;
13       input  [7:0]   layer;
14       input  [7:0]   mp_cmp_conv_fc;
15
16       var reg [31:0] data_dut , data_c_mod;
17       integer i , k , max_idx;
18       integer loop_cnt , max;
19       static integer flag;
20       string cls [] = {"Car" ,"Lane" , "Pedestrian" , "Light"};
21
22       loop_cnt = 0;
23       i = 0;
```

```
24     k = 0;
25     data_dut = 0;
26     data_c_mod = 0;
27     flag = 0;
28
29     if (!mp_cmp_conv_fc[0] && mp_cmp_conv_fc[2]) begin //
        Convolution with max pool
30     loop_cnt = (sb.conv_mp_size_r[layer]*sb.conv_mp_size_c[
        layer]);
31     //divided by 2, cuz reads two values at once.
32     loop_cnt = (loop_cnt * sb.conv_num_fil[layer])/2;
33     // $display("conv with mp, r%d c %d num fil: %d",sb.
        conv_mp_size_r[layer], sb.conv_mp_size_c[layer], sb.
        conv_num_fil[layer]);
34
35     end else if (!mp_cmp_conv_fc[0] && !mp_cmp_conv_fc[2]) begin
        // Convolution WITHOUT maxpool
36     loop_cnt = (sb.conv_op_size_r[layer]*sb.conv_op_size_c[
        layer]);
37     loop_cnt = (loop_cnt * sb.conv_num_fil[layer])/2;
38     // $display("conv without mp, r%d c %d num fil: %d",sb.
        conv_op_size_r[layer], sb.conv_op_size_c[layer], sb.
        conv_num_fil[layer]);
39     end else if (mp_cmp_conv_fc[0]) begin // Fully
        Connected
```

```
40     loop_cnt = (sb.conv_ip_size_c[layer])/2;
41     // $display("Fully connected, c%d ",sb.conv_ip_size_c[
        layer]);
42 end
43
44 // $display("Monitor: loop count: %d",loop_cnt);
45
46 for(i=0; i<loop_cnt; i++) begin
47     //Read from DUT
48     read_from_slave(addr, data_dut);
49     // $display("address: %H", addr);
50
51     //Read from C-Model
52     data_c_mod[15:0] = sb.mp_fm[k];
53     data_c_mod[31:16] = sb.mp_fm[k+1];
54     addr = {{addr[31:2] + 1},{2'b0}};
55     $display("layer: %d k. %d val0_dut: %d, val0_cmod: %d",
        layer, k, data_dut[15:0], data_c_mod[15:0] );
56     $display("layer: %d k. %d val1_dut: %d, val1_cmod: %d",
        layer, k, data_dut[31:16],data_c_mod[31:16]);
57
58     //Compare outputs
59     if(data_dut != data_c_mod) begin
60         $display("layer: %d k. %d val0_dut: %d, val0_cmod: %d
            ", layer, k, data_dut[15:0], data_c_mod[15:0] );
```

```
61     $display("layer: %d k. %d val1_dut: %d, val1_cmod: %d
        ", layer, k, data_dut[31:16], data_c_mod[31:16]);
62     flag++;
63     end else if(layer == 7) begin
64         sb.mntr_mp_cmod[k] = sb.mp_fm[k];
65         sb.mntr_mp_cmod[k+1] = sb.mp_fm[k+1];
66         sb.mntr_mp_dut[k] = data_dut[15:0];
67         sb.mntr_mp_dut[k+1] = data_dut[31:16];
68     end
69
70     k = k+2;
71     end
72
73     // if(flag == 0 && layer <7) begin
74         // $display("%d. layer completed successfully", layer);
75     //end else
76     if(flag == 0 && layer == 7) begin
77         //DUT or C model
78         max = 0;
79         max_idx = 0;
80         for(k=0; k<4; k++) begin
81             if(sb.mntr_mp_dut[k] > max) begin
82                 max = sb.mntr_mp_dut[k];
83                 max_idx = k;
84             end
```

```
85     end
86
87     if(max_idx == sb.label) begin
88         $display("ICNN says Image is: %s", cls[max_idx]);
89     end else begin
90         $display("Original label: %d, computed label: %d", sb
91             .label, max_idx);
92         $display("DUT: %d %d %d %d", sb.mntr_mp_dut[0], sb.
93             mntr_mp_dut[1],
94             sb.mntr_mp_dut[2], sb.mntr_mp_dut[3]);
95         $display("CMOD: %d %d %d %d", sb.mntr_mp_cmod[0], sb.
96             mntr_mp_cmod[1],
97             sb.mntr_mp_cmod[2], sb.mntr_mp_cmod[3]);
98     end
99     end else begin
100        if(flag !=0) begin
101            $display("Something's wrong in Layer: %d", layer);
102            flag = 0;
103        end
104    end
105
106    task read_from_slave();
```



```
107     input  [31:0]  addr;
108     output [31:0]  data;
109     begin
110         intf.fake_master_cyc = 1'b1;
111         intf.fake_master_stb = 1'b1;
112         intf.fake_master_we  = 1'b0;
113         intf.fake_master_adr = addr;
114         intf.fake_master_sel = 4'h0;
115         @(posedge intf.fake_master_ack);
116         @(posedge intf.clk);
117         data = intf.mm_o_s_wb_dat;
118         // $display("read_from_slave data[0] = %d", data[0]);
119         intf.fake_master_cyc = 1'b0;
120         intf.fake_master_stb = 1'b0;
121         intf.fake_master_we  = 1'b0;
122         @(posedge intf.clk);
123     end
124     endtask
125
126 endclass
```

Listing I.16: SystemVerilog Test-bench Monitor

I.17 Environment

```
1 class environment;
2     driver drv;
3     scoreboard sb;
4     monitor mntr;
5     cnn_model cnn;
6
7     virtual intf_dut intf;
8
9     function new(virtual intf_dut intf);
10         this.intf = intf;
11         sb = new();
12         cnn = new(intf, sb);
13         drv = new(intf, sb);
14         mntr = new(intf, sb);
15     endfunction
16
17 endclass
```

Listing I.17: SystemVerilog Test-bench Environment

I.18 Top Module

```
1
2 module test();
3     reg clk = 0;
4
5     initial
6         forever #10 clk = ~clk;
7
8     intf_dut intf(clk);
9
10    `ifdef NETLIST
11    ICNN    top(
12    `else
13    ICNN    #(.BIT_WIDTH(intf.BIT_WIDTH), .WB_DWIDTH(intf.WB_DWIDTH
14            ), .WB_SWIDTH(intf.WB_SWIDTH),
15            .NUM_LAYERS(intf.NUM_LAYERS), .MAX_IP_SIZE_R(intf.
16            MAX_IP_SIZE_R),
17            .MAX_IP_SIZE_C(intf.MAX_IP_SIZE_C), .MAX_OP_SIZE_R(intf.
18            MAX_OP_SIZE_R),
19            .MAX_OP_SIZE_C(intf.MAX_OP_SIZE_C), .COMPARE_3(intf.COMPARE_3
20            ),
21            .COMPARE_8(intf.COMPARE_8)) top(
22    `endif
23    .reset    (intf.reset),
```

```
20     .clk      ( intf.clk ),
21     .scan_in0 ( intf.scan_in0 ),
22     .scan_en   ( intf.scan_en ),
23     .test_mode ( intf.test_mode ),
24     .scan_out0 ( intf.scan_out0 ),
25     .o_m_wb_adr( intf.icnn_o_m_wb_adr ),
26     .o_m_wb_sel( intf.icnn_o_m_wb_sel ),
27     .o_m_wb_stb( intf.icnn_o_m_wb_stb ),
28     .i_m_wb_ack( intf.icnn_i_m_wb_ack ),
29     .i_m_wb_err( intf.icnn_i_m_wb_err ),
30     .o_m_wb_we ( intf.icnn_o_m_wb_we ),
31     .o_m_wb_cyc( intf.icnn_o_m_wb_cyc ),
32     .o_m_wb_dat( intf.icnn_o_m_wb_dat ),
33
34     .o_s_wb_ack( intf.icnn_o_s_wb_ack ),
35     .o_s_wb_err( intf.icnn_o_s_wb_err ),
36     .i_s_wb_adr( intf.icnn_i_s_wb_adr ),
37     .i_s_wb_sel( intf.icnn_i_s_wb_sel ),
38     .i_s_wb_we ( intf.icnn_i_s_wb_we ),
39     .i_s_wb_dat( intf.icnn_i_s_wb_dat ),
40     .o_s_wb_dat( intf.icnn_o_s_wb_dat ),
41     .i_m_wb_dat( intf.icnn_i_m_wb_dat ),
42     .i_s_wb_cyc( intf.icnn_i_s_wb_cyc ),
43     .i_s_wb_stb( intf.icnn_i_s_wb_stb ));
44
```

```
45
46 ACNN cnn(
47     .reset      ( intf.reset ),
48     .clk        ( intf.clk ),
49     .scan_in0   ( intf.scan_in0 ),
50     .scan_en    ( intf.scan_en ),
51     .test_mode  ( intf.test_mode ),
52     .scan_out0  ( intf.scan_out1 ),
53     .o_m_wb_adr( intf.acnn_o_m_wb_adr ),
54     .o_m_wb_sel( intf.acnn_o_m_wb_sel ),
55     .o_m_wb_stb( intf.acnn_o_m_wb_stb ),
56     .i_m_wb_ack( intf.acnn_i_m_wb_ack ),
57     .i_m_wb_err( intf.acnn_i_m_wb_err ),
58     .o_m_wb_we  ( intf.acnn_o_m_wb_we ),
59     .o_m_wb_cyc( intf.acnn_o_m_wb_cyc ),
60     .o_m_wb_dat( intf.acnn_o_m_wb_dat ),
61
62     .o_s_wb_ack( intf.acnn_o_s_wb_ack ),
63     .o_s_wb_err( intf.acnn_o_s_wb_err ),
64     .i_s_wb_adr( intf.acnn_i_s_wb_adr ),
65     .i_s_wb_sel( intf.acnn_i_s_wb_sel ),
66     .i_s_wb_we  ( intf.acnn_i_s_wb_we ),
67     .i_s_wb_dat( intf.acnn_i_s_wb_dat ),
68     .o_s_wb_dat( intf.acnn_o_s_wb_dat ),
69     .i_m_wb_dat( intf.acnn_i_m_wb_dat ),
```

```
70  .i_s_wb_cyc( intf .acnn_i_s_wb_cyc ) ,
71  .i_s_wb_stb( intf .acnn_i_s_wb_stb ) );
72
73  wishbone_arbiter arb(
74  .reset      ( intf .reset ) ,
75  .clk        ( intf .clk ) ,
76  .scan_in0   ( intf .scan_in0 ) ,
77  .scan_en    ( intf .scan_en ) ,
78  .test_mode  ( intf .test_mode ) ,
79  .scan_out0  ( intf .scan_out2 ) ,
80
81  .i_m0_wb_adr( intf .fake_master_adr ) ,
82  .i_m0_wb_sel( intf .fake_master_sel ) ,
83  .i_m0_wb_we ( intf .fake_master_we ) ,
84  .o_m0_wb_dat() ,
85  .i_m0_wb_dat( intf .fake_master_data ) ,
86  .i_m0_wb_cyc( intf .fake_master_cyc ) ,
87  .i_m0_wb_stb( intf .fake_master_stb ) ,
88  .o_m0_wb_ack( intf .fake_master_ack ) ,
89  .o_m0_wb_err() ,
90
91  .i_m1_wb_adr( intf .icnn_o_m_wb_adr ) ,
92  .i_m1_wb_sel( intf .icnn_o_m_wb_sel ) ,
93  .i_m1_wb_we ( intf .icnn_o_m_wb_we ) ,
94  .o_m1_wb_dat( intf .icnn_i_m_wb_dat ) ,
```

```
95  .i_m1_wb_dat( intf.icnn_o_m_wb_dat ),
96  .i_m1_wb_cyc( intf.icnn_o_m_wb_cyc ),
97  .i_m1_wb_stb( intf.icnn_o_m_wb_stb ),
98  .o_m1_wb_ack( intf.icnn_i_m_wb_ack ),
99  .o_m1_wb_err( intf.icnn_i_m_wb_err ),
100
101  .i_m2_wb_adr( intf.acnn_o_m_wb_adr ),
102  .i_m2_wb_sel( intf.acnn_o_m_wb_sel ),
103  .i_m2_wb_we ( intf.acnn_o_m_wb_we ),
104  .o_m2_wb_dat( intf.acnn_i_m_wb_dat ),
105  .i_m2_wb_dat( intf.acnn_o_m_wb_dat ),
106  .i_m2_wb_cyc( intf.acnn_o_m_wb_cyc ),
107  .i_m2_wb_stb( intf.acnn_o_m_wb_stb ),
108  .o_m2_wb_ack( intf.acnn_i_m_wb_ack ),
109  .o_m2_wb_err( intf.acnn_i_m_wb_err ),
110
111  .o_s0_wb_adr( intf.icnn_i_s_wb_adr ),
112  .o_s0_wb_sel( intf.icnn_i_s_wb_sel ),
113  .o_s0_wb_we ( intf.icnn_i_s_wb_we ),
114  .i_s0_wb_dat( intf.icnn_o_s_wb_dat ),
115  .o_s0_wb_dat( intf.icnn_i_s_wb_dat ),
116  .o_s0_wb_cyc( intf.icnn_i_s_wb_cyc ),
117  .o_s0_wb_stb( intf.icnn_i_s_wb_stb ),
118  .i_s0_wb_ack( intf.icnn_o_s_wb_ack ),
119  .i_s0_wb_err( intf.icnn_o_s_wb_err ),
```

```
120
121     .o_s1_wb_adr( intf .acnn_i_s_wb_adr ),
122     .o_s1_wb_sel( intf .acnn_i_s_wb_sel ),
123     .o_s1_wb_we ( intf .acnn_i_s_wb_we ),
124     .i_s1_wb_dat( intf .acnn_o_s_wb_dat ),
125     .o_s1_wb_dat( intf .acnn_i_s_wb_dat ),
126     .o_s1_wb_cyc( intf .acnn_i_s_wb_cyc ),
127     .o_s1_wb_stb( intf .acnn_i_s_wb_stb ),
128     .i_s1_wb_ack( intf .acnn_o_s_wb_ack ),
129     .i_s1_wb_err( intf .acnn_o_s_wb_err ),
130
131     .o_s2_wb_adr( intf .mm_i_s_wb_adr ),
132     .o_s2_wb_sel( intf .mm_i_s_wb_sel ),
133     .o_s2_wb_we ( intf .mm_i_s_wb_we ),
134     .i_s2_wb_dat( intf .mm_o_s_wb_dat ),
135     .o_s2_wb_dat( intf .mm_i_s_wb_dat ),
136     .o_s2_wb_cyc( intf .mm_i_s_wb_cyc ),
137     .o_s2_wb_stb( intf .mm_i_s_wb_stb ),
138     .i_s2_wb_ack( intf .mm_o_s_wb_ack ),
139     .i_s2_wb_err( intf .mm_o_s_wb_err ));
140
141 main_mem mm(
142     .clk      ( clk ),
143     .reset    ( intf .reset ),
144     .scan_en  ( intf .scan_en ),
```



```
145     .test_mode ( intf.test_mode ),
146     .i_mem_ctrl(1'b0),
147     .i_wb_adr  ( intf.mm_i_s_wb_adr ),
148     .i_wb_sel  ( intf.mm_i_s_wb_sel ),
149     .i_wb_we   ( intf.mm_i_s_wb_we ),
150     .o_wb_dat  ( intf.mm_o_s_wb_dat ),
151     .i_wb_dat  ( intf.mm_i_s_wb_dat ),
152     .i_wb_cyc  ( intf.mm_i_s_wb_cyc ),
153     .i_wb_stb  ( intf.mm_i_s_wb_stb ),
154     .o_wb_ack  ( intf.mm_o_s_wb_ack ),
155     .o_wb_err  ( intf.mm_o_s_wb_err ));
156
157     testcase test(intf);
158         assertion_acov acov(intf);
159
160     endmodule
```

Listing I.18: SystemVerilog Test-bench Top Module

I.19 Test

```
1
2 program testcase(intf_dut intf);
3
4     environment env = new(intf);
5     var reg [7:0] layer;
6     integer nth_img;
7     initial
8     begin
9         $timeformat(-9,2,"ns", 16);
10        layer = 0;
11        nth_img = 0;
12        env.drvr.reset();
13        env.drvr.set_constants();
14        env.drvr.write_weights();
15        env.drvr.layer_config_mem(5'h8);
16
17        for (nth_img=0; nth_img < 10; nth_img++) begin
18            env.drvr.write_image(nth_img);
19            env.drvr.write_start_bit();
20            //env.drvr.stop_icnn_layer(4'h1);
21            env.drvr.stop_icnn();
22            for (layer = 0; layer < 8; layer ++) begin
23                //Layer - 0
```

```
24     env.cnn.reset_c_model(layer);
25     // nth_lyr , lyr_ip_size_r , lyr_ip_size_c , lyr_mp_flag ,
        lyr_depth , lyr_num_fil;
26     // $display(" test_1: ip_size_r: %d, ip_size_c: %d,
        mp_conv_fc:%d, num_ch:%d, num_fil:%d",
27     // env.sb.conv_ip_size_r[layer], env.sb.conv_ip_size_c[
        layer ],
28     // env.sb.mp_cmp_conv_fc[layer], env.sb.conv_num_ch[layer
        ], env.sb.conv_num_fil[layer ]));
29     env.cnn.c_model(layer , nth_img ,
30     env.sb.conv_ip_size_r[layer ],
31     env.sb.conv_ip_size_c[layer ],
32     env.sb.mp_cmp_conv_fc[layer ],
33     env.sb.conv_num_ch[layer ],
34     env.sb.conv_num_fil[layer ]);
35     //Compare outputs of each layer
36     #50 env.mntr.compare_lyr(env.sb.layer_dest_addrss[layer
        ],
37     layer ,
38     env.sb.mp_cmp_conv_fc[layer ]);
39     end
40
41     end
42
43     $finish;
```

```
44  
45     end  
46 endprogram
```

Listing I.19: SystemVerilog Test-bench Test Case