A

Project Report

Entitled

Hardware Accelerator Design on FPGA

Submitted to the Department of Electronics Engineering in Partial Fulfilment for the Requirements for the Degree of

Bachelor of Technology

(Electronics and Communication)

: Presented & Submitted By :

Krishil Gandhi, Shantanu Banerjee, Mohit Sapkal

Roll No. (U20EC013, U20EC016, U20EC175)

B. TECH. IV(EC), 7th Semester

: Guided By :

Dr. Pinalkumar J. Engineer

Assistant Professor, DoECE

(Year: 2023-24)

DEPARTMENT OF ELECTRONICS ENGINEERING

SARDAR VALLABHBHAI NATIONAL INSTITUTE OF TECHNOLOGY

Surat-395007, Gujarat, INDIA.

Sardar Vallabhbhai National Institute Of Technology

Surat - 395 007, Gujarat, India

DEPARTMENT OF ELECTRONICS ENGINEERING

CERTIFICATE

This is to certify that the Project Report entitled "**Hardware Accelerator Design on FPGA**" is presented & submitted by Krishil Gandhi, Shantanu Banerjee, Mohit Sapkal, bearing Roll No. U20EC013, U20EC016, U20EC175, of B.Tech. IV, 7^{th} Semester in the partial fulfillment of the requirement for the award of B.Tech. Degree in Electronics & Communication Engineering for academic year 2023-24.

They have successfully and satisfactorily completed their **Project Exam** in all respects. We, certify that the work is comprehensive, complete and fit for evaluation.

Dr. Pinalkumar J. Engineer
Assistant Professor& Project Guide

PROJECT EXAMINERS:

Name of Examiners	Signature with Date
1. Dr. R. N. Dhavse	_____
2. Dr. Z. M. Patel	_____
3. Dr. Vivek Garg	_____
4. Dr. Shivendra Yadav	_____

Dr. Jignesh N. Sarvaiya
Head, DoECE, SVNIT

Seal of The Department
(December 2023)

Acknowledgements

I would like to express my profound gratitude and deep regards to my guide Dr. Pinalkumar J. Engineer for his guidance. I am heartily thankful for suggestion and the clarity of the concepts of the topic that helped me a lot for this work. I would also like to thank Prof. Dr. Jignesh N. Sarvaiya, Head of the Electronics Engineering Department, SVNIT and all the faculties of DoECE for their co-operation and suggestions. I am very much grateful to all my classmates for their support

Krishil Gandhi, Shantanu Banerjee, Mohit Sapkal
Sardar Vallabhbhai National Institute of Technology
Surat

December 2023

Abstract

With the slowing down of Moore's law and the rising complexity of Dense Neural Networks like ResNet, BERT, DLRM, and GPT (large language model), there is a need for specialized hardware, such as accelerators, for inference and training of these networks. Thus, we have created a hardware accelerator made especially for the edge, where the focus of the accelerators is to be as efficient as possible without the loss of accuracy in the inference of the Neural Network. We have used a 4×4 MAC or Multiply and Accumulate Unit architecture using the IEEE 754 FP8 floating point format for the convolution layers. We have also made average pooling hardware and non-linear hardware functions so that all the computational happening in the network occurs in the accelerator. Existing accelerators use FP16, BF16, and other formats, which have a tradeoff of higher power and area usage in order to achieve acceptable accuracy. We have proposed the FP8 format as the number system for our processing element, as it consumes half the bits of as that of FP16 and it consumes 89.6736059 % less power, 76.3349021 % less area and 51.364003 % less clock period.

Table of Contents

List of Figures

List of Tables

List of Abbreviations

ML	Machine Learning
NoP	Network on Package
BERT	Bidirectional Encoder Representations from Transformers
PCI	Peripheral Component Interconnect
DLRM	deep learning recommendation model
GPT	Generative Pre-Trained Transformers
MAC	Multiply and Accumulate
FP	Floating Point
AI	Artificial Intelligence
DL	Deep Learning
DNN	Deep Neural Network
TPU	Tensor Processing Unit
PE	Processing Element
API	Application Programming interface
CPU	Central processing unit
AiM	Acceleration in memory
ReLU	Rectified Linear unit
GPU	Graphic Processing Unit
CNN	Convolution Neural Network
NoC	Network on chip
DRAM	Dynamic Random Access memory
BRAM	Block Random Access memory
SRAM	Static Random Access memory
MCM	Multi-Chip module
RNN	Recurrent Neural Network
BWMS	Band wide mantissa shift
LUT	Lookup Table
GDDR	Graphics Double Data rate
RLC	Run length compression

Chapter 1
Introduction

1.1 Motivation

Artificial Intelligence (AI) and Machine Learning (ML) have profoundly transformed modern life, touching virtually every aspect of our daily existence. These technologies have revolutionized industries, optimizing processes, enhancing efficiency, and fostering innovation. In healthcare, AI assists in diagnostics, personalizing treatments, and predicting disease outbreaks—ML algorithms power recommendation systems in entertainment and e-commerce platforms, tailoring content and products to individual preferences. The automotive sector is undergoing a radical shift with the development of self-driving cars, made possible by sophisticated AI algorithms. Smart homes leverage AI to automate tasks, improve energy efficiency, and enhance security. In finance, ML models analyze vast datasets to detect patterns, reduce risks, and optimize investment strategies. Additionally, AI-driven natural language processing and image recognition advancements have led to virtual assistants, language translation tools, and facial recognition systems. Different aspects of AI are shown in Figure 1.1.

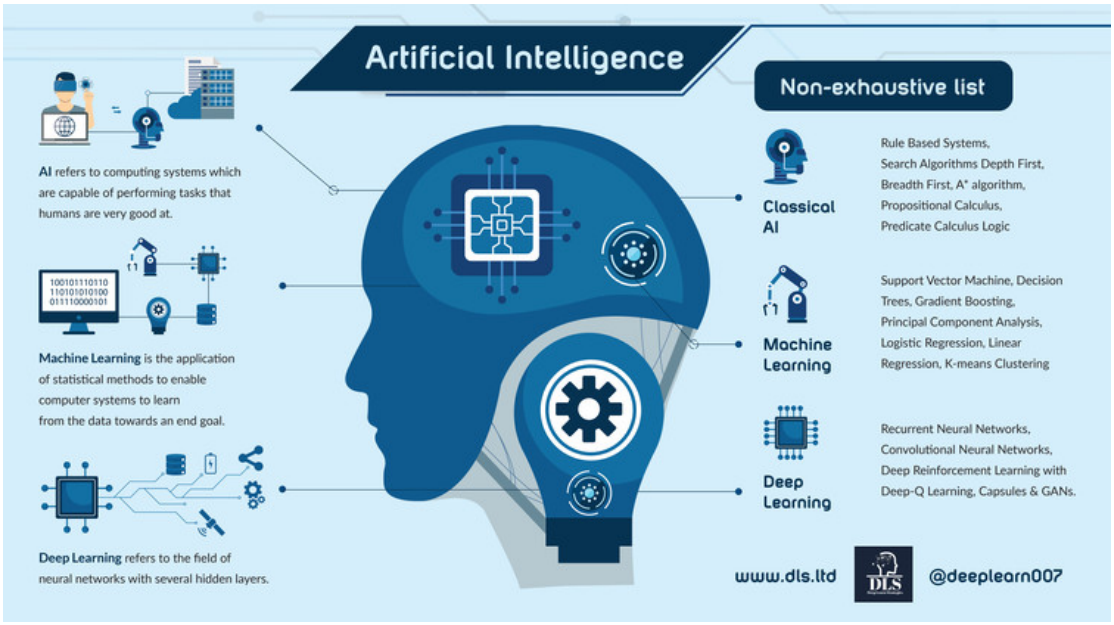

Figure 1.1: Artificial Intelligence

Given this, hardware solutions that can accelerate these AI-ML workloads have emerged as promising solutions to deal with the power and memory wall inherent to von Neumann-based general-purpose computing in handling massive data manip-

ulations. DNN hardware accelerators (HA) are optimally designed for multiply-and-accumulate (MAC) operation, the core computing operation in DNNs. The main advantage is gained by operating thousands of MAC units in parallel to reduce the task's latency. Specifically, with technology scaling, there is potential to add many MAC units on a large die area. One of the advantages of creating energy-efficient accelerators is that they can be used on the edge without any connection to the cloud. Processing data locally on the edge device using dedicated AI accelerators can enhance privacy by reducing the need to send sensitive data to centralized cloud servers. This is important for applications dealing with personal or confidential information. They are also suitable for real-time processing requirements of edge applications. This is critical for applications like autonomous vehicles, robotics, or augmented reality, where immediate responses are necessary for safe and effective operation.

Figure 1.2: Tenstorrent Grayskull AI inference Accelerator [1].

There is much activity in this field in the industry, with established players like NVIDIA, AMD, and INTEL creating their own AI accelerators in the form of server GPUs or adding them as a part of a CPU, as in the case of H100 and MI300 GPU's in the case of NVIDIA and AMD and AMD's XDNA architecture in laptop CPUs. Many startups, such as Tenstorrent and Cerebras, have created these accelerators and are selling them as standalone PCIe cards alongside a main system or as a large computer server for a highly computationally demanding workload. This shows that there is a high demand for these types of semiconductor chips and a lot of scope for innovation to fulfill various applications in AI.

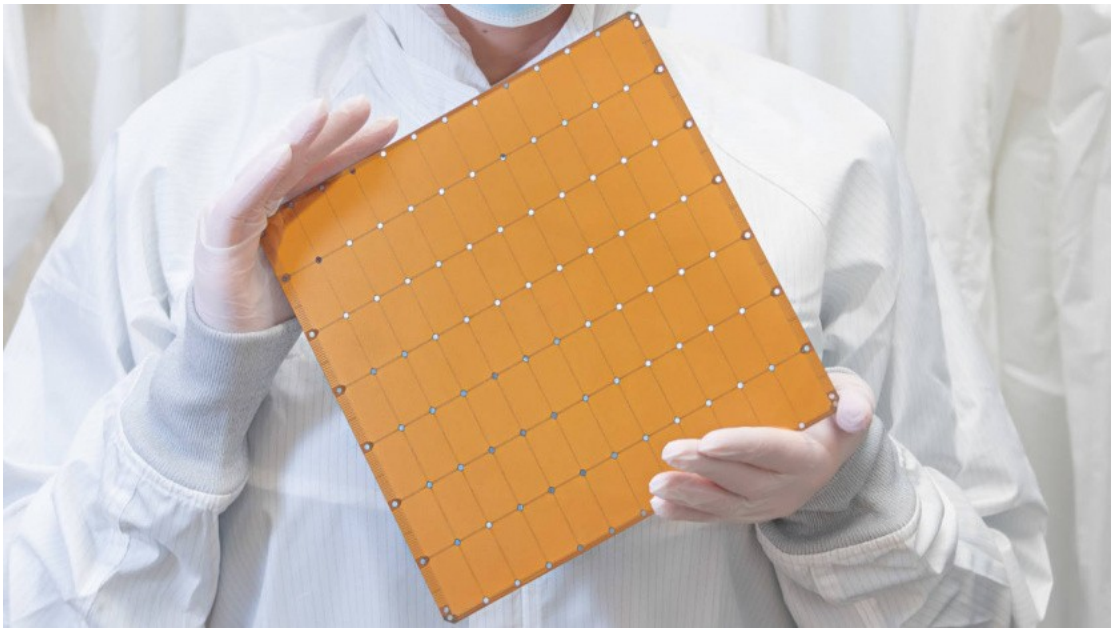

Figure 1.3: Cerebras Waferscale Engine AI inference Accelerator [2].

1.2 Objective

Thus, this work aims to provide a design for energy-efficient hardware accelerators for deep neural networks. First, we set out to find the best floating point format that will consume the least power and area, and there is a lower amount of accuracy drop-off due to the quantization of the data. Next, we created multipliers and adders using the best format to make our novel 4x4 MAC architecture, which takes advantage of parallelism in the problem of neural networks to accelerate the workload. Then, we connect the necessary memories and simulate the convolution layer of any neural network. In this work, we have decided to do this on a modified version of the Lenet-5 architecture. We have also created specialized hardware to implement pooling and nonlinear activation functions in the accelerator so that all the computations can happen parallel inside the accelerator.

Chapter 2
Literature Review

The realm of deep learning continues to witness groundbreaking advancements, with a convergence of hardware and software innovations shaping the trajectory of neural network accelerators. This literature review embarks on a comprehensive exploration of seminal research papers that delineate the evolution and transformative potential of deep-learning accelerators. The Gemmini [3] paper spearheads this discourse by proposing a holistic framework for evaluating deep-learning architectures within full-stack environments, establishing a foundation for comprehensive assessments. Furthermore, the Accelerator in Memory [4] paper introduces an innovative accelerator-in-memory design, leveraging 1 ynm technology to achieve unparalleled performance through versatile activation functions and high-throughput MAC operations.

In addition to these pioneering works, the Eyeriss [5] paper emphasizes energy efficiency as a cornerstone in accelerating neural network operations, fostering sustainable computing architectures. Simultaneously, the Simba [6] paper addresses scalability challenges by harnessing multi-chip-module-based architectures for efficient deep-learning inference. Together, these seminal research endeavors encapsulate diverse facets of deep-learning accelerator design, spanning systematic evaluation methodologies, high-performance memory-based accelerators, energy-efficient architectures, scalable inference strategies, and innovative hardware-software co-design paradigms. A comprehensive understanding of these contributions is pivotal in deciphering the trajectory and implications of deep-learning architectures for system-level performance, energy efficiency, and scalability within the artificial intelligence domain.

2.1 Gemmini

In this literature review we will will dive deeper into the reviews of all the mentioned papers starting with the first paper Gemmini [3].

2.1.1 Introduction

The quest for accelerated Deep Neural Network (DNN) processing, spanning edge to cloud, has spurred an array of research endeavors to craft specialized accelerators. Among these advancements, Gemini emerges as a pivotal contribution, offering a full-stack, open-source generator designed to facilitate comprehensive evaluations of various accelerator architectures. This review aims to distill Gemmini's architectural innovations, programming paradigms, performance benchmarks, and implications within

the evolving landscape of DNN accelerator research.

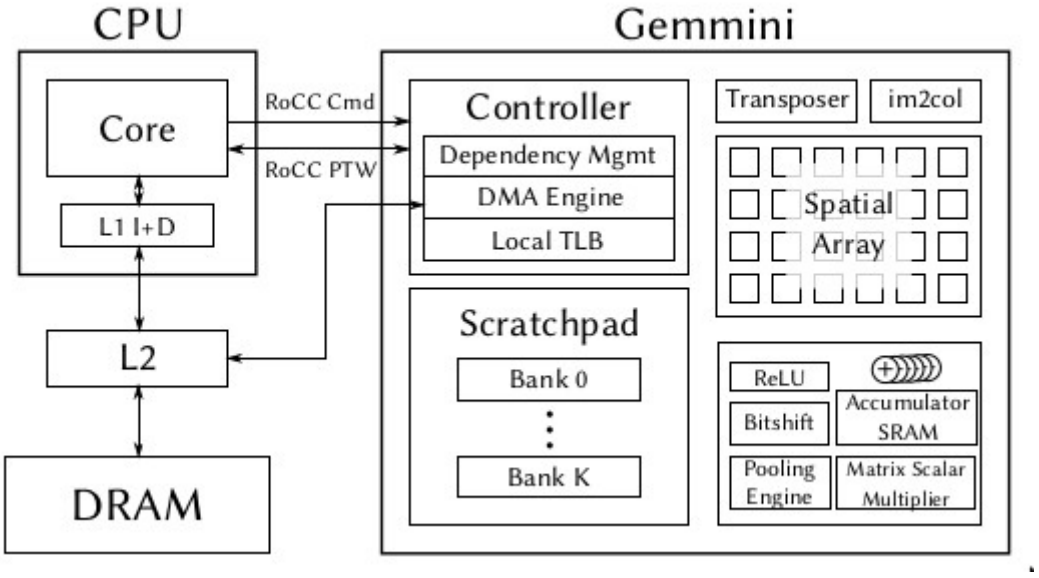

Figure 2.1: Gemmini hardware architectural template overview [3].

2.1.2 Architectural Innovation in DNN Accelerators

The landscape of DNN accelerators is characterized by a diverse array of specialized architectures aimed at meeting varying performance and energy efficiency targets [3]. Notably, Gemmini's breakthrough lies in its comprehensive support for an expansive spectrum of data types and data flow configurations, enabling a nuanced comparative analysis of efficiency and scalability across different accelerator designs [3]. The core architectural framework of Gemmini revolves around a spatial array housing Processing Elements (PEs) capable of executing dot products and accumulations. This hierarchical spatial array structure, delineated by a two-level layout, provides a versatile template for exploring a continuum of accelerator designs—from highly-pipelined architectures reminiscent of TPUs to parallel vector engines akin to NVDLA [3].

2.1.3 Programming Paradigms and Runtime Optimization

Gemmini's programming paradigm encompasses a multi-level software flow, bridging high-level abstractions for mapping DNN descriptions to accelerators and low-level interactions via C/C++ APIs. This multi-pronged approach empowers programmers to optimize data staging and mapping during runtime, supported by robust virtual memory handling [3]. Furthermore, Gemmini's runtime data staging capabilities, coupled with its virtual memory support, not only simplify programming complexities but also

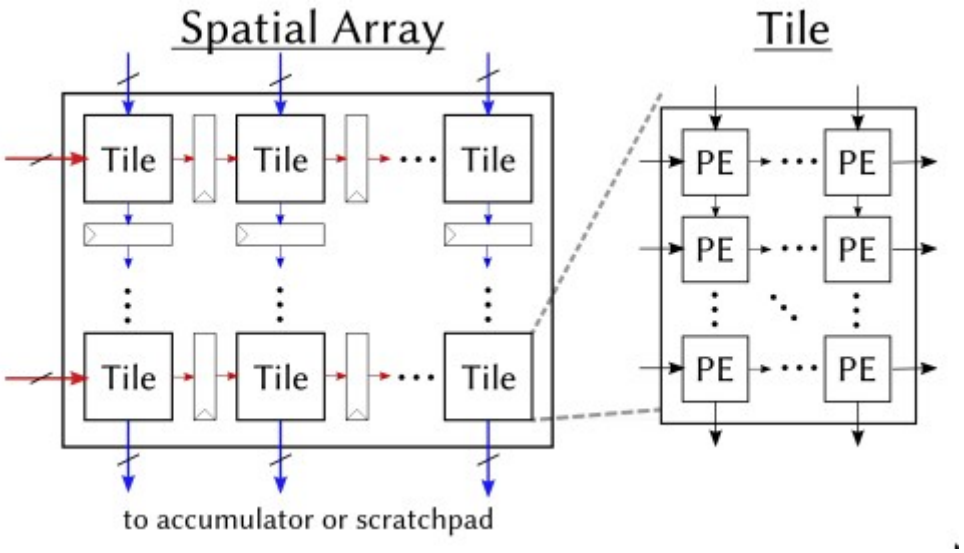

Figure 2.2: Gemmini hardware architectural template overview [3].

facilitate explorations into virtual address translation systems [3]. Such features offer a playground for programmers and researchers to fine-tune accelerator performance based on specific computational demands.

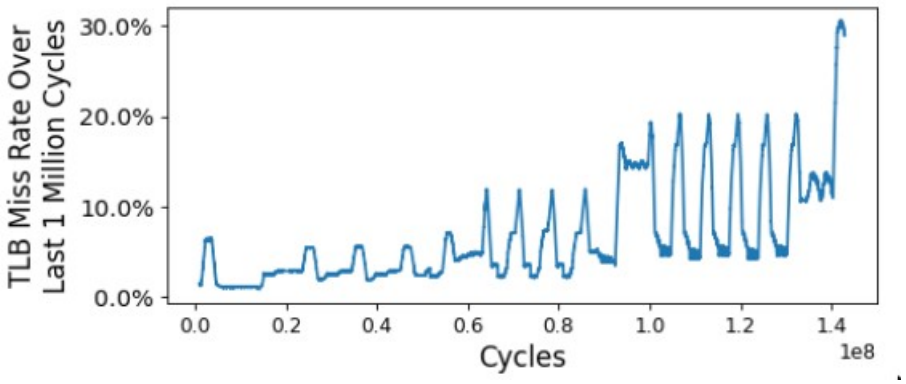

Figure 2.3: TLB miss rate over a full ResNet50 inference, profiled on a Gemmini-generated accelerator [3].

2.1.4 Performance Evaluation and Use Case Demonstrations

A hallmark of Gemmini's prowess lies in its exceptional performance benchmarks when pitted against host CPUs. For instance, the platform exhibits unparalleled speedups, achieving a staggering 2,670x speedup over in-order CPUs and an 1,130x speedup over out-of-order CPUs during ResNet50 inference when equipped with an on-the-fly im2col unit [3]. These findings underscore Gemmini's competitive edge across diverse DNN

models and applications. Moreover, the platform's case studies vividly illustrate its versatility in system-level co-design. Investigations into virtual address translation systems and system-level resource partitioning unveil Gemmini's potential in optimizing edge device performance and making critical system-level design decisions based on the unique computational characteristics of specific DNN models [3].

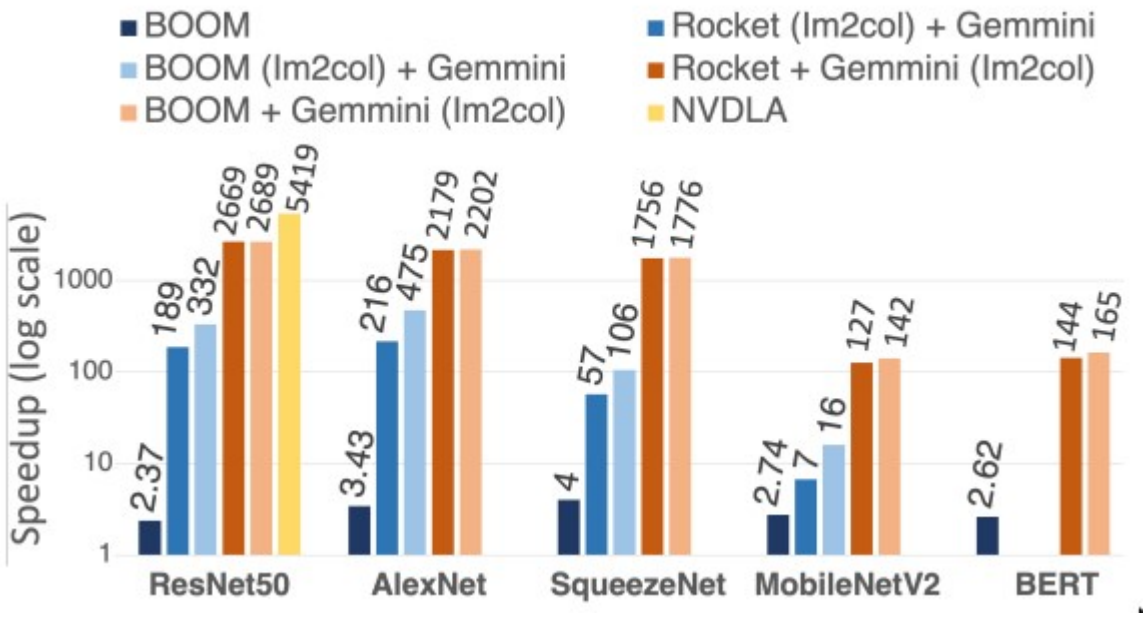

Figure 2.4: Speedup compared to an in-order CPU baseline. For CNNs, im2col was performed on either the CPU, or on the accelerator [3].

2.2 Accelerator-in-Memory

Moving on to the Accelerator in Memory [4] research paper

2.2.1 Introduction

The rapid evolution of Deep Learning (DL) techniques has precipitated an insatiable demand for computational architectures that can efficiently handle the complex computations inherent in DL tasks. In response to this demand, the GDDR6-AiM architecture emerges as a pioneering solution at the intersection of GDDR6 memory technology and innovative AiM (Acceleration in Memory) techniques [4].

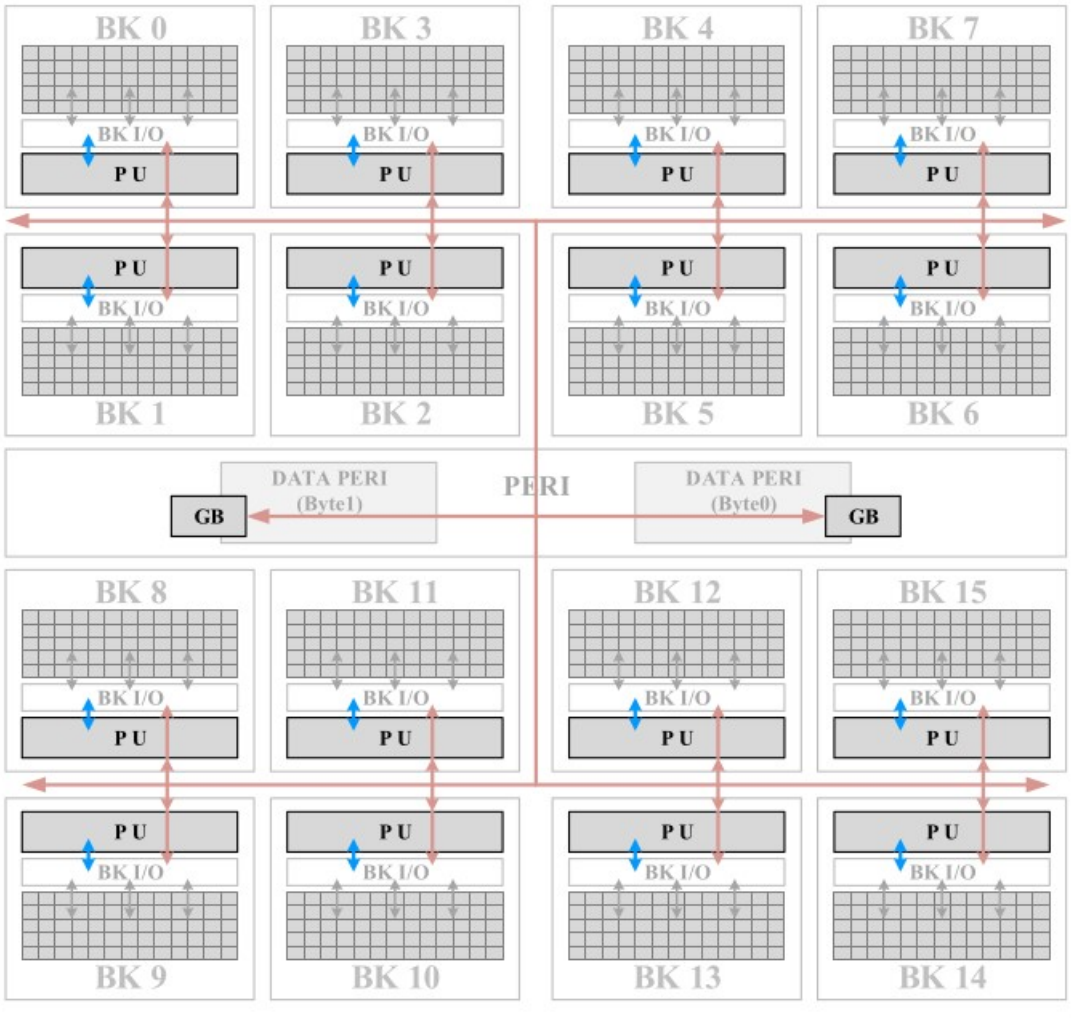

Figure 2.5: AiM Architecture [4].

2.2.2 Evolution of DL Architectures

The landscape of DL architectures has evolved drastically over the years, propelled by the escalating requirements for computational prowess and memory bandwidth to accommodate increasingly intricate DL workloads [4]. Conventional architectures have grappled with meeting these escalating demands, setting the stage for revolutionary advancements in architectural design [4].

2.2.3 GDDR6-AiM Architecture Overview

At the crux of the GDDR6-AiM architecture lie several critical components meticulously designed to optimize DL operations:

1. **Multiplier-Accumulator (MAC) Enhancement:** The incorporation of the inno-

vative Bank-Wide Mantissa Shift (BWMS) scheme within the adder tree stands as a testament to the architecture's ingenuity. Through BWMS, the architecture achieves remarkable reductions in power consumption by up to 66%, computational time by 52%, and a substantial 75% decrease in the required area compared to conventional methodologies [4].

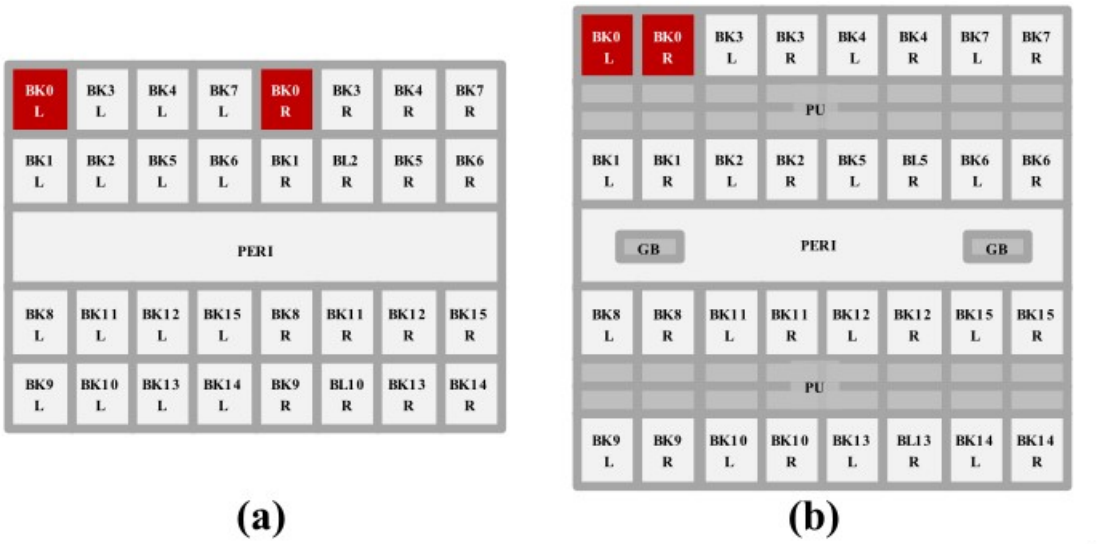

Figure 2.6: (a) Normal GDDR6 bank L/R architecture. (b) AiM adjacent bank L/R architecture [4].

2. **Accumulator Optimization:** Explores the architecture's optimization strategies within the accumulator, employing adjacent bank L/R structures to facilitate seamless integration of adder trees, reducing the need for multiple accumulators and enhancing computational efficiency. Efforts to streamline the architecture extend to the accumulator, where the integration of adjacent bank L/R structures results in an optimization strategy aimed at enhancing computational efficiency and minimizing potential bottlenecks [4].

3. **Activation Functions and Linear Interpolation:** The architecture's support for various Activation Functions (AFs) spans from computation-based approaches like ReLU and Leaky ReLU to Look-Up Table (LUT)-based methods such as sigmoid, Tanh, and GELU [4]. Additionally, the implementation of linear interpolation techniques contributes to precise function value derivation within confined inputs, enriching the computational accuracy [4].

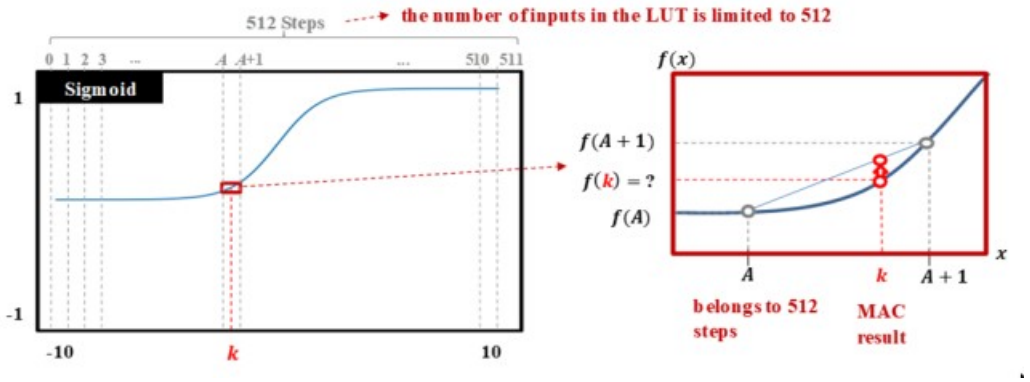

Figure 2.7: Linear Interpolation [4].

2.2.4 In-Depth Analysis of Architecture Components

Each architectural component is dissected, elucidating the underlying mechanisms, design methodologies, and the rationale behind their incorporation. This section delves into the intricate technical details, exploring circuit-level optimizations, computational methodologies, and integration strategies. This section meticulously dissects each architectural component, offering comprehensive insights into the underlying design philosophies, intricate technical nuances, circuit-level optimizations, and the rationale driving their integration [4].

2.2.5 Performance Evaluation and Validation

The GDDR6-AiM architecture undergoes a battery of meticulous evaluations and validation processes:

1. **Chip-Level Measurements:** GATE-level measurements unequivocally validate the architecture's operational power, affirming its seamless performance at 16 Gbps while operating at a significantly reduced voltage of 1.10 V, showcasing a remarkable 0.15 V reduction from the intended design target [4].

2. **FPGA-Based System-Level Evaluation:** Despite inherent limitations restricting operations to 2 Gbps due to FPGA constraints, the comprehensive evaluation conducted on Xilinx UltraScale+ FPGA-based platforms offers invaluable insights into the architecture's performance dynamics [4].

2.2.6 Comparative Analysis and Impact Assessment

A comparative analysis juxtaposes the GDDR6-AiM architecture against existing systems, particularly A10 and GV100 GPUs, highlighting the profound performance ad-

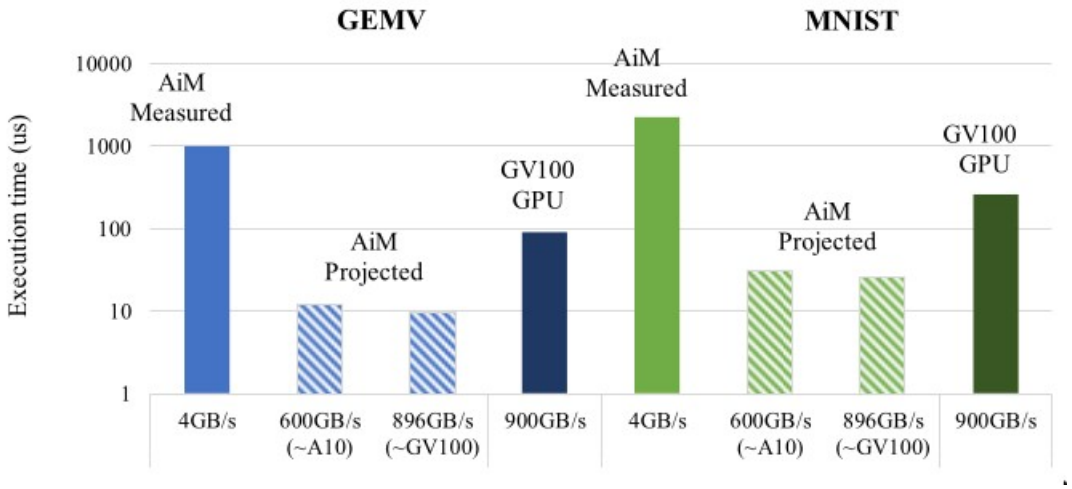

Figure 2.8: Performance Measurement [4].

vantages offered by GDDR6-AiM. It underscores the architecture's potential for accelerating DL operations with notable speedups ranging from 7.5× to 10.5× over these existing systems.

2.3 Eyeriss

Moving on to the Eyeriss [5] paper.

2.3.1 Introduction

The contemporary landscape of artificial intelligence is shaped significantly by Deep Learning (DL) techniques, especially Convolutional Neural Networks (CNNs), renowned for their exceptional accuracy in modern applications. However, these networks demand extensive computational power, predominantly from the heavy data movement involved, stressing the necessity for efficient hardware accelerators [5]. This review focuses on Eyeriss, an accelerator designed to optimize energy efficiency across various CNN shapes by dynamically configuring its architecture [5]. The emphasis lies in minimizing energy costs associated with data movement to achieve both high throughput and energy efficiency [5].

2.3.2 Evolution and Challenges

CNNs, fundamental to modern AI, pose challenges due to their immense data requirements, leading to substantial energy consumption in data movement. The hardware must balance high parallelism for throughput and optimize data movement efficiency

to ensure energy-efficient processing. This optimization becomes more critical with the varying shapes of high-dimensional convolutions in CNNs [5].

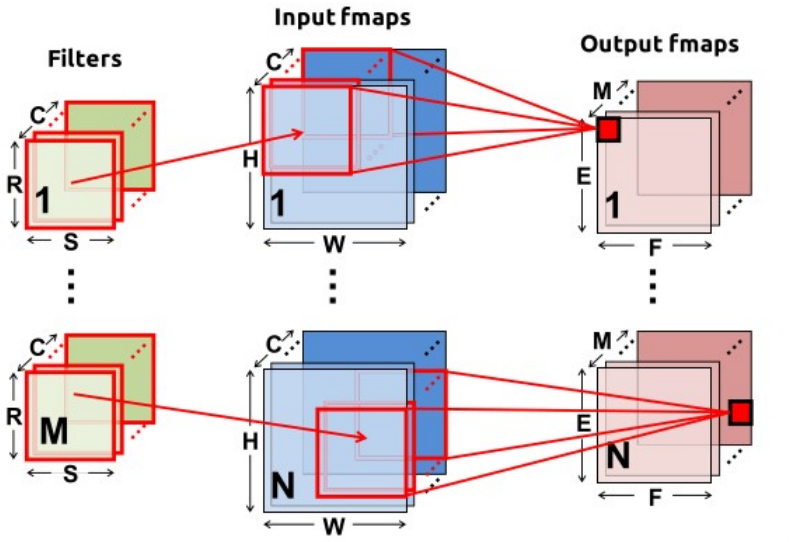

Figure 2.9: Computation of a CNN layer [5].

2.3.3 Design Strategies

The paper highlights critical design strategies:

1. **Row Stationary (RS) Dataflow:** Eyeriss implements an RS dataflow, reconfiguring computation mappings based on a CNN shape, maximizing data reuse to mitigate costly data movements [5]. Spatial Architecture: Utilizes a 168 Processing Element (PE) spatial array, creating a memory hierarchy exploiting multilevel memory accesses to minimize costly DRAM accesses [5].

2. **Network-on-Chip (NoC):** Employs multicast and point-to-point single-cycle data delivery to support RS dataflow, enhancing efficient data transport. Compression and Gating: Leveraging zero-data statistics in CNNs, techniques like run-length compression (RLC) and PE data gating are applied, further boosting energy efficiency [5].

2.3.4 Comparative Analysis

While prior works have proposed CNN accelerators, most lack verified measurements from fabricated chips or comprehensive benchmarks against state-of-the-art CNNs [5]. Eyeriss stands out as it not only implements and fabricates the proposed accelerator but

also benchmarks it against widely-used CNNs—AlexNet and VGG-16. This comprehensive evaluation includes chip energy efficiency, DRAM access requirements, and real-time integration with Caffe for practical applications [5].

2.3.5 CNN Basics and System Architecture

The review explains the fundamental operations of CNN layers, emphasizing the high-dimensional convolutions and computation intricacies involved in these layers. It further dives into the system architecture of Eyeriss, depicting a multilevel memory hierarchy, asynchronous control structures, and reconfigurable processing elements.

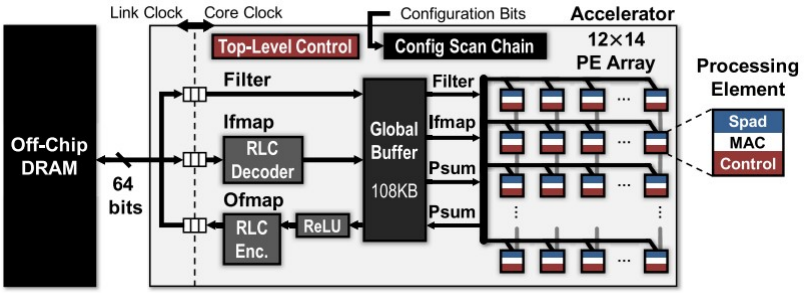

Figure 2.10: Eyeriss System Architecture [5].

2.3.6 Energy-Efficient Features

Energy-Efficient Dataflow: Row Stationary (RS)

1. **Maximizing Data Reuse:** Eyeriss capitalizes on three forms of data reuse to reduce ifmap and filter movement:

   (a) **Convolutional Reuse:** Each filter weight is reused E × F times within the same ifmap plane, and each ifmap pixel undergoes R × S reuse within the same filter plane.

   (b) **Convolutional Reuse:** Each filter weight is reused E × F times within the same ifmap plane, and each ifmap pixel undergoes R × S reuse within the same filter plane.

   (c) **Filter and Ifmap Reuse:** Filters are reused across the batch of N ifmaps, while ifmap pixels are reused across M filters to generate M ofmap channels.

2. **RS Dataflow Implementation:**

(a) **1- D Convolution Primitive:** Eyeriss divides the computation into parallel 1- D convolution primitives, each handling filter weight and ifmap value rows, resulting in localized computation within each PE [5].

(b) **2-D Convolution PE Set:** This configuration groups PEs to perform 2-D convolutions, sharing rows of filter or ifmap across primitives and accumulating psums, minimizing accesses to high-cost memory levels [5].

3. **PE Set Mapping Strategies:** Mapping onto PE Array: Eyeriss employs a mapping strategy to accommodate PE sets onto the PE array, optimizing local data sharing and psum accumulation. Handling Exceptions: For PE sets exceeding 168 PEs or dimensions surpassing the array's capacity, strip mining or segmentation techniques are utilized [5].

4. **Beyond 2-D Processing in PE Array:**

(a) **Multiple 2-D Convolutions:** Exploiting the spatial array's capabilities, Eyeriss interleaves computation primitives for simultaneous execution, enhancing data reuse [5].

(b) **Multiple PE Sets:** Leveraging the PE array's potential, multiple sets are mapped to increase processing throughput, maximizing data reuse and accumulation within the array [5].

Exploiting Data Statistics

1. **ReLU Function's Impact on Data Sparsity:** Zero Density in Fmaps: The ReLU function often introduces zeros in feature maps (fmaps) by converting negative filtering results to zero. The proportion of zeros in fmaps tends to rise with deeper layers. Filter Pruning: Studies indicate significant potential for pruning filter weights to zero, ranging from 16% to 78% across CNN layers [5].

2. **Run-Length Compression (RLC) Implementation:** Eyeriss employs RLC to exploit fmap sparsity, effectively compressing fmap data and reducing DRAM bandwidth usage. Decompression in Accelerator: Compressed ifmaps are read from DRAM, decompressed using the RLC decoder within the accelerator, and stored in the GLB. Similarly, computed ofmaps are compressed and transmitted back to DRAM, saving both space and R/W bandwidth.

2.3.7 System Modules

Global Buffer (GLB)

GLB Structure: Eyeriss features a 108 kB GLB responsible for communication between DRAM, PE array, and storage of ifmaps, filters, and psums/ofmaps. Dynamic GLB Allocation: The 100 kB allocation for ifmaps and psums is configurable to suit different proportions based on the layer's requirements. The GLB architecture provides simultaneous access to both ifmaps and psums from distinct banks [5].

Network-on-chip(NOC)

Optimized NoC Architecture: Eyeriss's custom NoC handles data delivery between the GLB and PE array, designed to support RS dataflow's delivery patterns, optimize energy efficiency, and offer adequate bandwidth. Distinct Networks: GINs (Global Input Networks) and GONs (Global Output Networks) facilitate efficient data transmission between the GLB and PE array. Additionally, a Local Network (LN) facilitates direct data transfer between neighboring PEs [5].

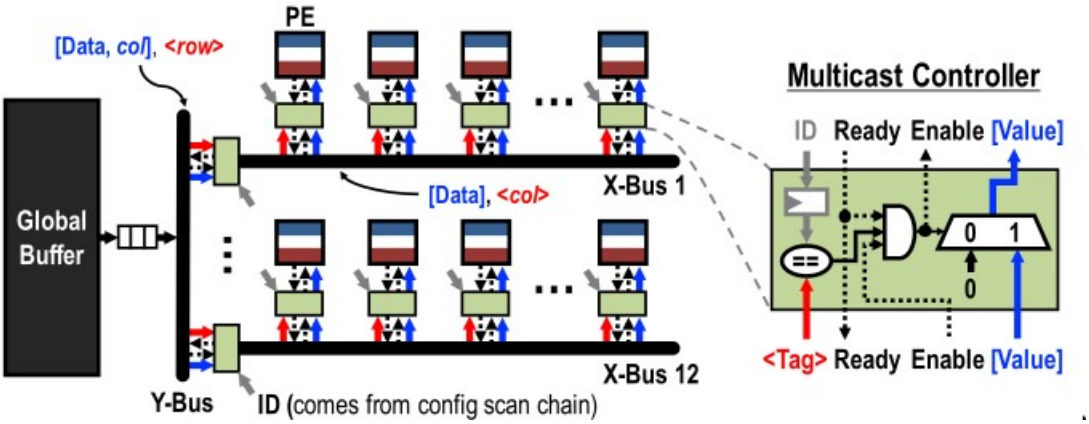

Figure 2.11: Architecture of GIN [5].

Processing Element (PE) and Data Gating

PE Architecture: Each PE employs FIFOs to balance NoC and computation loads. The PE configuration, including the number of filters and channels processed simultaneously, dictates its processing state. Data Gating Logic: Eyeriss implements data gating to exploit zeros in ifmaps, reducing processing power consumption significantly. The gating logic disables unnecessary computations based on zero detections in ifmaps, contributing to a 45% power consumption reduction in the PE [5].

2.3.8 Performance and Benchmarking

Chip Specifications and Performance with AlexNet

1. **Chip Summary:** The Eyeriss chip, designed in 65-nm CMOS, integrated into the Caffe framework, boasts a peak throughput of 33.6 GMAC/s at 1 V and 200-MHz core clock. Its architecture supports CNN shapes without requiring modifications [5].

2. **Area Breakdown:** The core area, excluding I/O pads, comprises logic cells, registers, SRAMs, with the PE array consuming a significant portion. The combined on-chip storage (GLB and spads) occupies two-thirds of the total area [5].

3. **Performance with AlexNet:** Performance metrics based on benchmarking with AlexNet indicate an average frame rate of 34.7 frames/s, achieving a processing throughput of 23.1 GMACS at 1 V. The chip power consumption measures 278 mW, resulting in an energy efficiency of 83.1 GMACS/W [5].

4. **Performance Factors:** Factors contributing to a lower actual throughput than peak throughput include inactive PEs (88%), data loading time from GLB, and idling during DRAM data movement. Refinements in DRAM traffic control could optimize performance further [5].

Power Distribution and Efficiency Insights

Simulated power breakdown during CONV1 and CONV5 reveals varying power distributions due to different data flow mappings and reuse patterns. Computation-related components (ALUs) account for less than 10% of total power, while data movement-related elements (spads, GLB, NoC) contribute up to 45%, emphasizing the energy consumption linked to data movement. Voltage Scaling Impact: Voltage scaling analysis demonstrates performance scaling with different voltages. The maximum throughput reaches 45 frames/s at 1.17 V, while the peak energy efficiency hits 122.8 GMACS/W at 0.82 V.

Performance Analysis with VGG-16

Performance with VGG-16: Evaluation of the 13 CONV layers in VGG-16 showcases an average operation rate of 0.7 frames/s at 1 V, consuming 236 mW. The disparity in performance between AlexNet and VGG-16 is due to VGG-16's significantly higher computational requirements and layer configuration. Impact of Layer Configuration: Despite similar MAC operations in layers like CONV1-2 and CONV4-2, the former takes longer due to increased processing passes in early layers. This elongated time

stems from the need for more passes to ramp up processing in the PE array, predominantly driven by larger fmap sizes.

2.4 Simba

Moving on to the Simba [6] paper.

2.4.1 Introduction

Package-level integration employing Multi-Chip Modules (MCMs) stands as a promising avenue for constructing large-scale systems. The integration of numerous chiplets into a unified system offers a distinct advantage, significantly reducing both fabrication and design costs [6].Deep Learning (DL) inference demands substantial compute power and on-chip storage. The conventional approach to meeting these requirements involves large monolithic chips or multi-chip boards. However, MCMs have been explored as a solution for DL inference accelerators.

Evolution of DL Hardware Architectures

Historically, DL inference hardware relied on monolithic chips or multi-chip boards [6]. The emergence of MCM-based DL accelerators, exemplified by projects like Simba, highlights a shift toward leveraging fine-grained chiplets for DL inference tasks.

Simba: An MCM-Based DL Accelerator

Simba represents a pioneering approach, integrating 36 chiplets into a prototype MCM system tailored for DL inference. Each chiplet demonstrates noteworthy performance, collectively contributing to a package capable of achieving high throughput and energy efficiency. Its adaptable architecture allows for flexible mapping of DNN layers across distributed compute and storage units.

Challenges and Optimizations in MCM-Based DL Systems

MCM-based DL accelerators face challenges related to non-uniform latency and bandwidth across chiplets, impacting inference layer execution times. Simba introduces three tail-latency-aware tiling optimizations, emphasizing data locality improvement, reduced communication overhead, and enhanced system utilization.

Significance of Simba in DL Hardware Architectures

Simba's comprehensive characterization and novel optimizations underscore the crucial role of task and data placement in MCM-based DL systems. Its quantitative evaluation highlights the importance of addressing non-uniform communication characteristics in scalable MCM integration for DL inference, paving the way for future advancements in DL hardware architectures.

2.4.2 Simba Architecture and System

In this we discuss about the Simba Architecture and system.

Simba Architecture

Simba adopts a hierarchical interconnect model comprising a Network-on-Chip (NoC) connecting Processing Elements (PEs) within chiplets and a Network-on-Package (NoP) connecting chiplets across the package. This hierarchical structure ensures efficient communication among different processing elements while mitigating tile-to-tile communication latency challenges [6].

Simba PE and Global PE

Each Simba chiplet comprises an array of PEs, a Global PE, a NoP router, and a controller interconnected via a chiplet-level network. The Simba PE consists of distributed weight buffers, input buffers, vector MAC units, accumulation buffers, and post-processing units. Employing a weight-stationary data flow approach, the vector MACs perform efficient spatial reduction, and configurable cross-PE reductions enhance flexibility in computation [6]. The Global PE, acting as a secondary storage unit, facilitates flexible data partitioning across PEs, enabling both unicast and multicast data transmission. Additionally, it supports near-memory computation to optimize certain DNN operations, reducing communication overhead for specific tasks [6].

Simba Silicon Prototype

The silicon prototype of Simba comprises chiplets interconnected through an organic package substrate. Using ground-referenced signaling (GRS) technology for intra-package communication, each chiplet incorporates transceiver macros allowing configurable data lanes and clock speeds from 11 Gbps/pin to 25 Gbps/pin, providing a peak chiplet bandwidth of 100 GB/s. GRS technology exhibits superior bandwidth per unit area and lower energy per bit, outperforming other MCM interconnects [6]. Implementing chiplets using a globally asynchronous, locally synchronous (GALS)

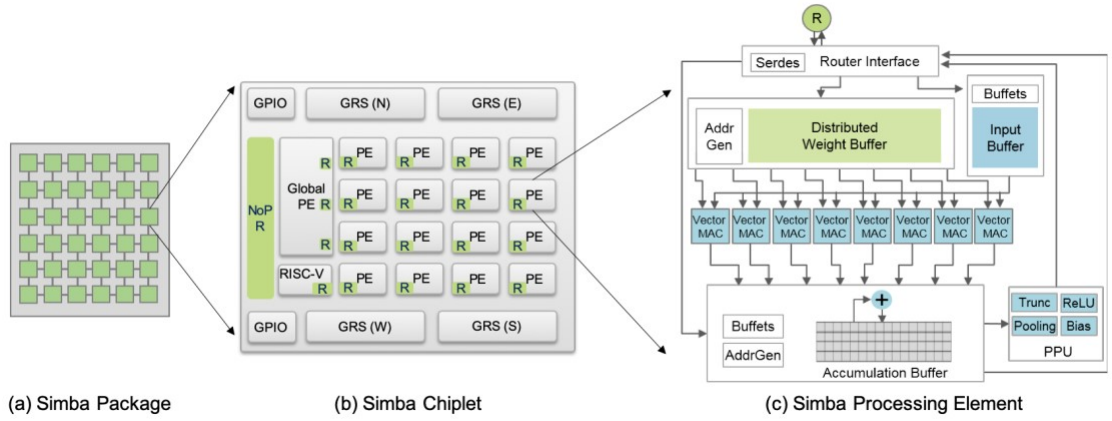

Figure 2.12: Simba architecture from package to processing element (PE) [6].

clocking methodology ensures independent clock rates for different elements within the chiplet. The prototype demonstrates operational efficiency across various voltage ranges, achieving significant power efficiency at lower voltages and notable throughput at higher voltages [6].

Simba Baseline Tiling

To map DNN layers onto Simba's tile-based architecture, a default tiling strategy uniformly partitions weights spatially, leveraging model parallelism. This strategy allows flexible configuration of loop bounds and orderings, supporting users' flexibility in mapping computations to the Simba system. However, this uniform tiling approach has limitations when applied to a large-scale, non-uniform network architecture like an MCM-based system.

2.4.3 Simba Characterization

In this we discuss about the characterization of Simba Architecture.

Methodology

The performance characterization of Simba focused on evaluating scalability using the uniform-tiling baseline [6]. The experimental setup involved a silicon prototype test board connected to an x86 host via PCI-E using a Xilinx FPGA. Cycle counters built into the RISC-V microcontrollers facilitated performance measurement [6], while power measurements employed sense resistors on board power supplies and a digital acquisition module.

Overview

Each point represents a unique mapping for a layer, while colors depict the number of active chiplets for that mapping [6]. Notably, Simba's mappings exhibit diverse performance and energy profiles, highlighting the significance of efficient DNN mapping strategies. Layers with high data reuse factors tend to perform computations more efficiently than those requiring extensive data movement [6].

Mapping Sensitivity

Examining the performance of ResNet-50's layers mapped across chiplets reveals intriguing patterns [6] . For instance, when mapped across chiplets, certain layers exhibit execution times that don't scale proportionally with the number of PEs employed. Inter-chiplet communication and synchronization overheads notably affect execution time, emphasizing the need for robust mapping strategies considering the NoC and NoP characteristics.

Layer Sensitivity

The scalability of different layers in ResNet-50 varies significantly with the number of chiplets [33]. Some layers demonstrate performance improvements with increased chiplet count, while others plateau or degrade due to communication overheads. This variation underscores the diversity in compute parallelism across layers and the impact of communication costs on efficiency [6].

NoP Bandwidth Sensitivity

Adjusting NoP bandwidth relative to intra-chiplet compute performance reveals interesting insights [6]. While increased bandwidth marginally affects execution time for certain layers, others experience substantial improvements, indicating layers bottlenecked by inter-chiplet communication [6].

NoP Latency Sensitivity

Higher NoP latency significantly impacts execution time, as demonstrated by experiments modulating chiplet locations. Increased inter-chiplet communication latency considerably elongates execution time, emphasizing the crucial role of latency management for optimal system performance [6].

Weak Scaling Sensitivity

Simba's weak scaling trend, demonstrated using DriveNet, highlights a throughput increase with increased chiplet count [6]. However, the synchronization cost across chiplets contributes to a latency increase despite throughput improvements [6].

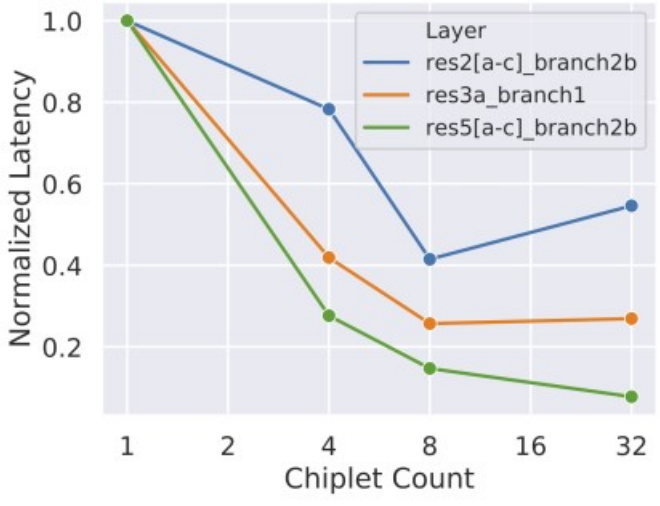

Figure 2.13: Simba scalability across different layers fromResNet-50. Latency is normalized to the latency of the best-performing tiling with one chiplet [6].

Comparisons with GPUs

Comparative analysis between Simba and NVIDIA's V100 and T4 GPUs reveals Simba's superior throughput and energy efficiency, particularly at low batch sizes [49]. Simba's design prioritizes low-latency inference, leveraging distributed and persistent weight storage to minimize data movement [6].

2.4.4 Simba Non-Uniform Tiling

In this section we discuss about the Simba Non-Uniform Tiling.

Non-Uniform Work Partitioning

Efficient utilization of parallel systems requires balanced load distribution among system components to prevent increased tail latency [6]. Current DNN tiling strategies distribute work uniformly across available resources, which can lead to inefficiencies in large-scale systems with spatially distributed PEs and varying communication latencies. To address this issue, we propose a non-uniform work partitioning strategy that accounts for communication latencies. Instead of uniformly assigning work to PEs, we

distribute it non-uniformly, allocating more work to PEs closer to data sources while reducing the workload for distant PEs [6]. This strategy maximizes data locality and mitigates tail latency effects.

Implementing this approach requires leveraging performance counters within each PE to gather latency and utilization data during initial execution, allowing subsequent workload adjustments based on latency variations across PEs [6].

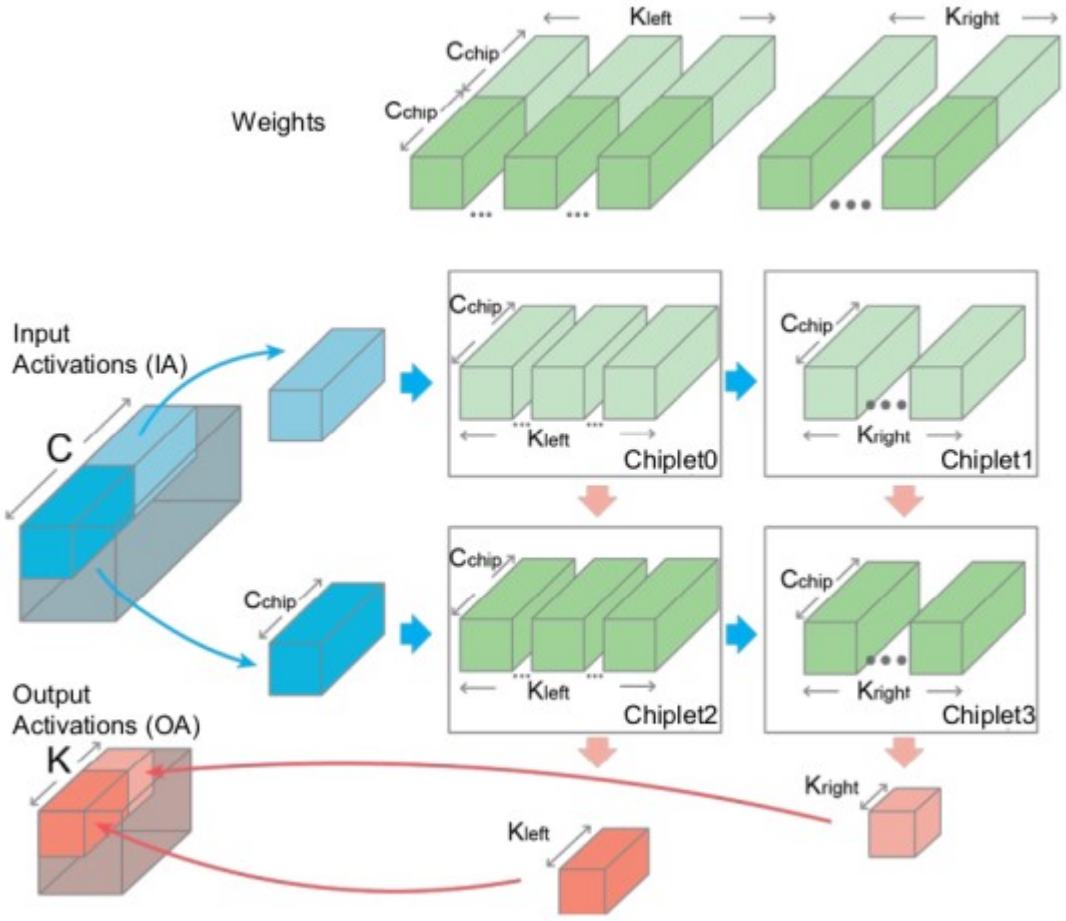

Figure 2.14: Illustration of communication-aware, non-uniform work partitioning. The top green tensors represent weights (W), the left blue tensors represent input activations (IA), and the bottom red tensors represent the output activation (OA). In this example, IA is stored in Chiplet0 and Chiplet2 [6].

Communication-Aware Data Placement

Communication latency's significant impact on system performance in parallel systems necessitates optimizing data placement [6]. Large-scale MCM systems with spatially

distributed buffers demand careful data placement to mitigate communication latencies. Despite the NP-hard nature of optimal data placement, we employ a practical greedy algorithm to iteratively determine the optimal placement of input and output activation data in the Simba system [6].

Cross-Layer Pipelining

Mapping DNN layers to large-scale systems confronts challenges in achieving high utilization when computation lacks parallelism. Pipelined execution serves as a solution to enhance overall system utilization [6]. Simba's hierarchical interconnect flexibility enables assigning different-sized clusters of chiplets to various layers.Throughput improvements achieved through pipelining three residual blocks, showcasing up to a 2.3× throughput increase compared to sequential execution. However, throughput remains limited by the longest pipelining stage, with varying degrees of improvement across layers due to differing input activation and weight sizes [6].

2.5 Summary

Thus, we have looked at various existing hardware accelerators from various organizations. Gemmini [3], exhibits unparalleled speedups, achieving a staggering 2,670x speedup over in-order CPUs and an 1,130x speedup over out-of-order CPUs during ResNet50 inference when equipped with an on-the-fly im2col unit .These findings underscore Gemmini's competitive edge across diverse DNN models and applications. The GDDR6-AiM [4] architecture against existing systems, particularly A10 and GV100 GPUs, achieves 7.5× to 10.5× over these existing systems. The Eyeriss chip [5], designed in 65-nm CMOS, integrated into the Caffe framework, boasts a peak throughput of 33.6 GMAC/s at 1 V and 200-MHz core clock. Simba's chiplet [6] based design hierarchical design achieves 128 TOPS with exceptional energy efficiency, unveiling groundbreaking strategies like non-uniform work partitioning and communication-aware data placement. After looking at these state-of-the-art architectures, we are proposing our own architecture based on the modified MATAC [8] and the floating point format in order to improve the performance as compared to these architectures.

Chapter 3
Mathematical analysis of computation in DNN

In this chapter, we will look into the basics concept before diving into the details for the architecture. We will be covering the detail analysis of floating point arithmetic and Deep Learning.

3.1 Need of Floating point arithmetics

Computation in science and engineering requires precision, range, and speed tradeoffs. A number format has to provide accuracy for numbers at very different magnitudes. Usually, an extensive range of numbers is required, but some inaccuracy in precision can be tolerated. One of the potential formats is fixed point arithmetic. It involves using fixed bits for the integer part of the number and fixed bits for the fractional part. This leads to a high amount of precision but a smaller range, which is undesirable. Hence, fixed point arithmetic is not being used. Thus, all computers use floating-point arithmetic in some form or the other. The term floating point refers to the fact that the number's radix point can "float" anywhere to the left, right, or between the significant digits of the number. A floating-point system can represent, with a fixed number of digits, numbers of very different orders of magnitude — such as the number of meters between galaxies or protons in an atom. The most widespread form of floating point is the IEEE 754 floating point format. The IEEE Standard for Floating-Point Arithmetic (IEEE 754) is a technical standard for floating-point arithmetic established in 1985 by the Institute of Electrical and Electronics Engineers (IEEE).

$$Number = (-1)^{sign} * 2^{exponent} * 1.mantissa \tag{3.1}$$

The number consists of the sign bit, the exponent bit, and the mantissa bits, where the sign is the sign bit's value. If the sign bit is 1, the number is negative; otherwise, the number will be positive. The exponent bits are the value of the exponent for the base, which has to be subtracted from the bias. The value of the bias depends on the number of exponent bits. The value of bias will be $2^{numberofexponentbits}$ -1. The mantissa bits work to provide values other than the powers of 2. Since 1 of the fraction is always present, we don't explicitly store the value of 1 to save the memory.

For example, in the IEEE 754 single precision floating point format or the fp32 format, there are one sign bit, eight exponent bits, and 23 mantissa bits. The value of bias will be 127 since the exponent bits is 8. Various formats have varying amounts of range and precision for different applications. For example, the double precision fp64 format

ranges from 2^{-1022} to 2^{+1023} and has a precision of 2^{-53}. This gives the user considerable flexibility, but it takes 64 bits to represent a number, which is a massive amount of storage to store a single number. Hence, this format is used only in applications such as scientific computing, where precision and range are essential, and memory is not an issue.

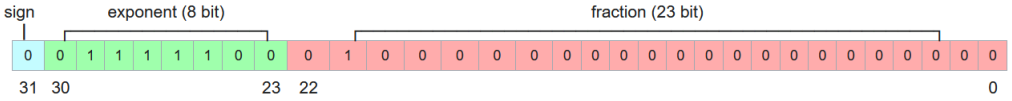

Figure 3.1: IEEE 754 floating point format [7].

3.2 Floating Point Arithmetic

In deep neural networks, the most basic operation are multiplication and addition. In this section, we will briefly discuss the logic related to it.

3.2.1 Floating Point Multiplication

Multiplication and additions are the two most essential Deep Neural Network workload operations. Now, we will look at how these operations occur in hardware for floating point numbers.

Following is the flow for the floating point multiplication. The operation on sign, exponent, and mantissa bits are handled separately. To find the sign of the output operand, we XOR the sign of both the input operands. For example, if both the operands are positive or negative, the output will be positive or the value 0. Similarly, if one of the operands is negative and the other positive, the output will be negative or 1. This behavior is achieved with the help of the XOR gate.

In the case of exponent, the multiplication of two numbers with the same base exponent is the addition of the two powers of the base of the two numbers. So we add the exponent of the two numbers to get the intermediate answer. Also, since we add the bias twice during addition, we have to subtract it once to get the correct value. Then, the value of the exponent can be changed based on the multiplication in the mantissa. To normalize the exponent, we check the multiplication value of the mantissa. If the MSB is 1, one must be added to the exponent.

For the operation in the mantissa, we first add 1 to the MSB of both the mantissa as it is implicitly assumed, but we have to add it back into the mantissa. Then, we multiply the mantissa using integer multiplication. If the MSB of the product is 0, then we right-shift the value of the mantissa by 1. Now, we have to round the product as

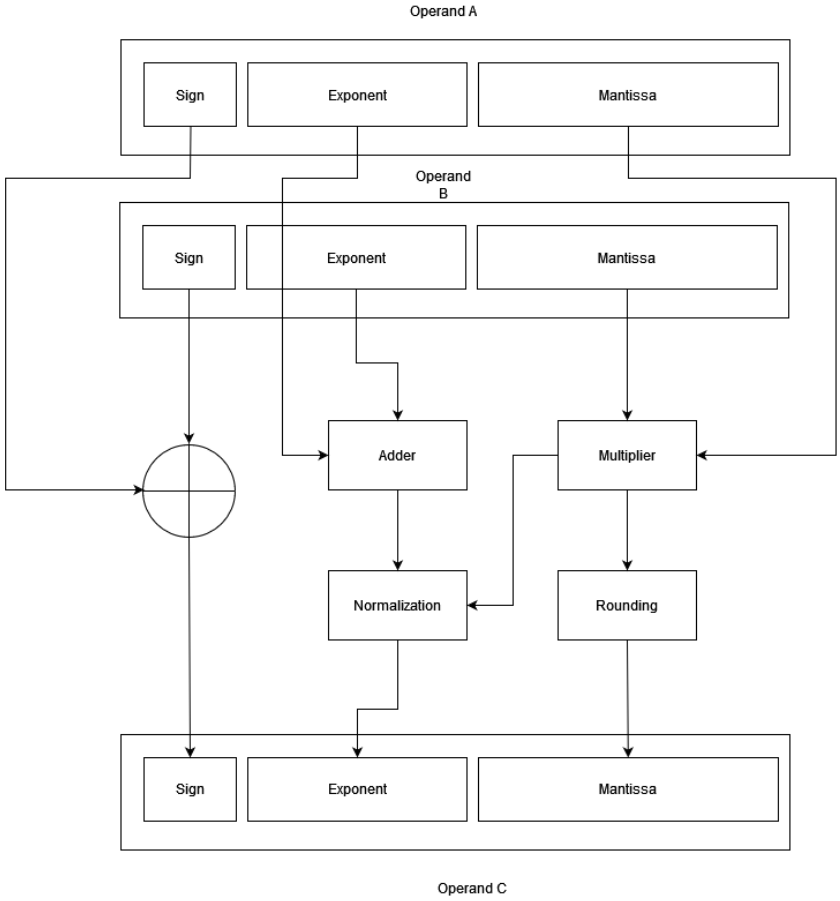

Figure 3.2: Floating point multiplication hardware

the number of bits in the product has doubled, so we have to reduce it by half so that it can be fitted back into the same space in the memory. There are many ways we can round the number. We can remove the extra bits, round to the next value, round to the previous value, and many more. We are rounding to the nearest even value where all the LSB are 0.

The extra bits used in intermediate calculations to improve the result's precision are called guard bits. It is only a tradeoff of hardware cost (keeping extra bits) and speed versus accumulated rounding error because these extra bits have to be rounded off to conform to the IEEE standard. For round-to-nearest-even, we need to know the value to the right of the LSB (round bit) and whether any other digits to the right of the round digit are 1s (the sticky bit is the OR of these digits). The IEEE standard requires three extra bits of less significance than the 24 bits (of mantissa) implied in the single precision representation – guard bit, round bit, and sticky bit. When a mantissa is to be shifted to align radix points, the bits that fall off the least significant end of the mantissa go into these extra bits (guard, round, and sticky bits). These bits can also be set by the normalization step in multiplication and by extra bits of quotient (remainder)

in division. The guard and round bits are just two extra bits of precision that are used in calculations. The sticky bit indicates what is/could be in lesser significant bits that are not kept. If a value of 1 ever is shifted into the sticky bit position, that sticky bit remains a 1 ("sticks" at 1) despite further shifts.

Since the component that will lead to the highest delay will be the operation on the mantissa, we have to focus on optimizing the multiplication hardware or reducing the number of mantissa bits to improve the multiplier's performance.

3.2.2 Floating point Addition

In floating point addition, the hardware algorithm is described in Figure. 3.3

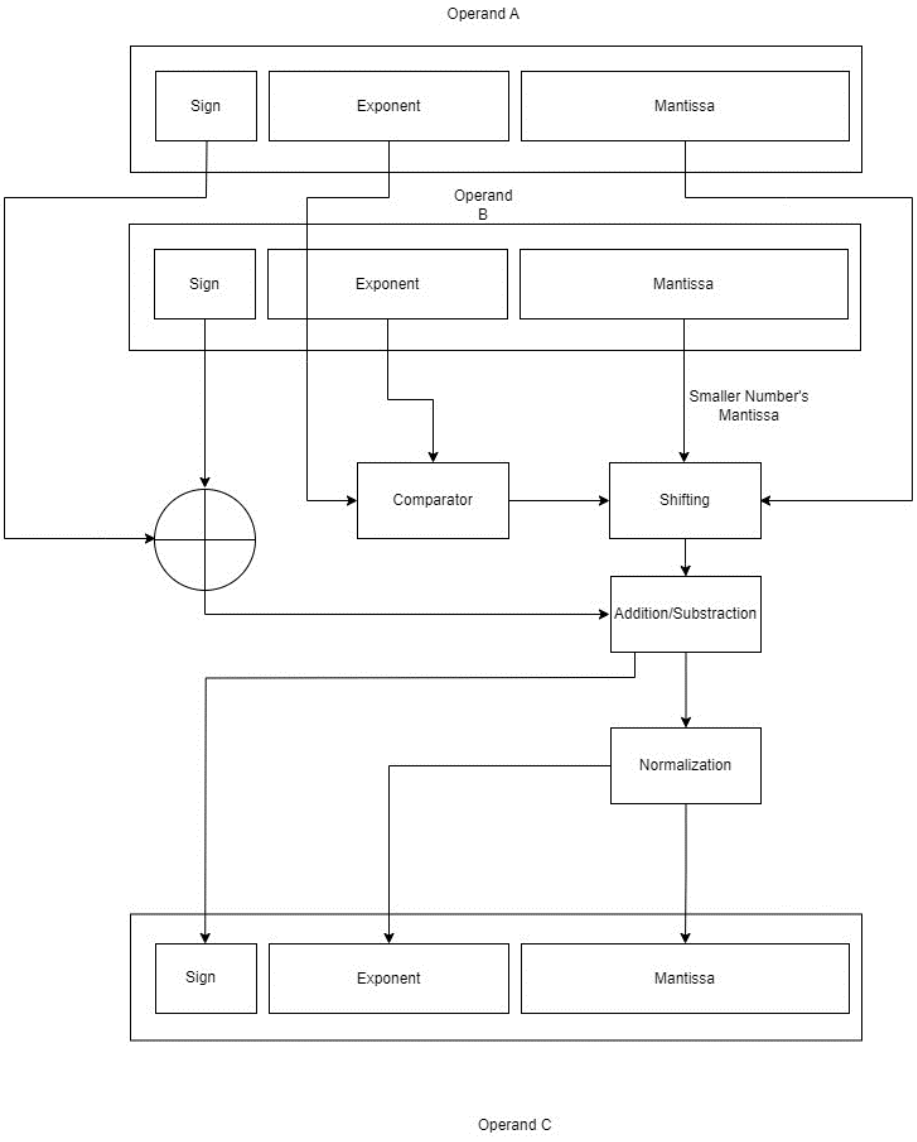

Figure 3.3: Floating point addition hardware

First, we compare the exponents bits as we could add or subtract the mantissa part only if the exponents are same. Here, we always try to convert the smaller number to a bigger number, this is because when we do so the mantissa bits of the smaller number are always shifted towards right-hand side which does not make the number greater than 2. This indirectly prevent the extra hardware logic, but it can sometimes cause extreme data loss if the exponent's difference is quite big. This is not a prominent issue in deep learning where the numbers are mostly between -1 to 1. Next, after making the exponents same, the mantissa part is added or subtracted based on xor operation of the sign bit. Again, after this normalization is needed if the added number increases beyond 2. After normalization, results are stored.

3.3 Choosing the best floating point format

Since there are a plethora of floating point formats, we need to find the best one based on our application, that is, neural network training and inference. We must check every multiplier's accuracy on the inference on many different CNN and transformer workloads to check how it will perform on light and heavy DNN workloads. We must also check its area and power performance on the same operating frequency to find the most appropriate multiplier. We must check the trade-off between accuracy and performance to get the best multiplier.

Table 3.1: Comparison of accuracy for different floating point system [9].

Workload	**FP32**	**FP16**	**BF16**	**INT16**	**FP8**
VGG-16	71.27	71.2	71.27	54.26%	71.11
Resnet-18	70.58	70.58	70.58	53.18%	70.12

We have measured the Top-1 Accuracy on both the networks on the CIFAR-10 dataset. As we can see, there is a negligible drop in accuracy as we move from FP32 to FP8 floating point format. However, there is a massive drop in accuracy for integer multipliers. This is a huge problem for integer multipliers because they can't be reliably used for DNN inference.

Above is the table regarding the power and area consumption of different floating point and integer MATAC units. We have performed the analysis using Cadence Genus using the 45nm technology node. We have given all the designs a clock of 20ns, so it will be a relevant comparison without the timing constraint affecting the decision. We have provided a VCD file or the signals for the design, which has 10000 operands between -1 to 1 so that we can simulate the real-life scenario on the multiplier as the majority of the numbers found in this workload will be between this range as all the inputs and weights are normalized. The graph has been made by normalizing all the

Table 3.2: Comparison of performance for different floating point system.

Floating Point Format	**Power (in mW)**	**Area (in um^2)**	**Minimum Clock Period (in ns)**
FP32	12.071	123515	20
FP16	3.397	34422	6.195
INT16	0.272	6404	1.015
BF16	2.263	32402	6.136
FP8(4E 3M)	0.472	9314	3.22
FP8(5E 2M)	0.350	8146	3.013

metrics with respect to the metric of INT16 which has the lowest values from the given formats.

We can see that FP8 provides the least power and area consumption, runs at the highest frequency, and doesn't have a drop-off in accuracy with respect to baseline FP32 inference accuracy, which is the case of integer multipliers. Thus, we have decided to use the FP8 format for our design. Since there is a 6.5% change in frequency between the multipliers of 4E 3M and 5E 2M, both formats can be used in the design. We will use the 5E 2M design to get the maximum performance out of the multiplier and accumulator unit.

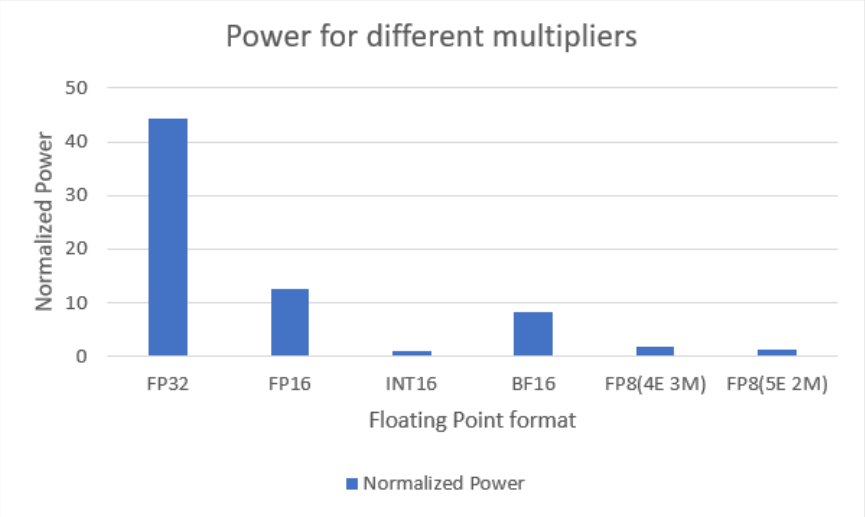

Figure 3.4: Normalized Power for different floating point system

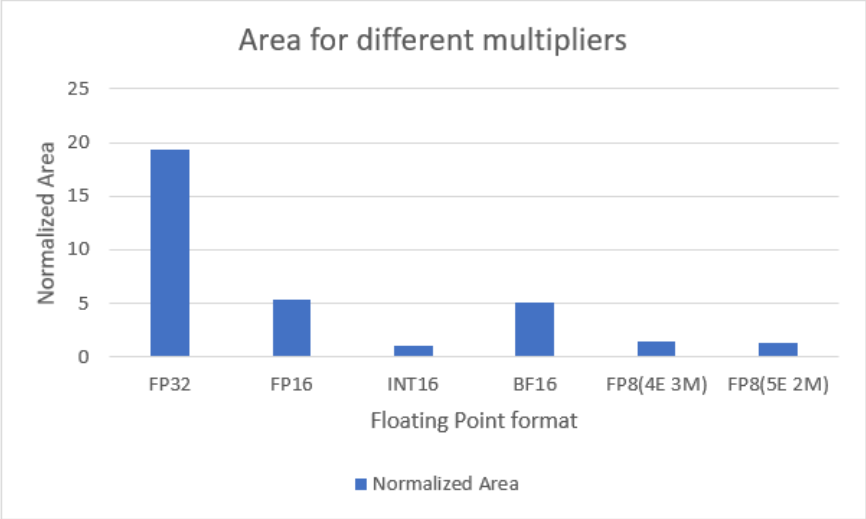

Figure 3.5: Normalized Area for different floating point system

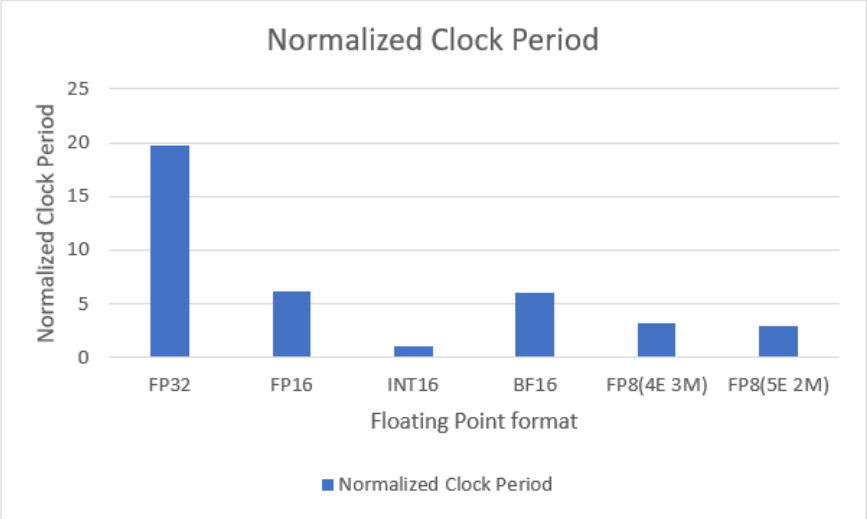

Figure 3.6: Normalized Clock Period for different floating point system

3.4 Address generation for the SRAM

There will be thousands of operations that will occur during a single convolution layer. So, it is not wise for the CPU to give the address during each MAC operation. So, some address-generation hardware must provide the address to the SRAM without the control unit's intervention. Since there are 16 operands in the MATAC, all the operands can go inside the MATAC in a 3×3 filter, but we need to send the data by first putting 16 and then 9 in the MATAC for the 5×5 filter. The good thing we can take advantage of is the

reused input during the convolution, so we don't have to fetch that data again from the SRAM. This will lead to fewer SRAM reads, and we must consider this for the address generation unit.

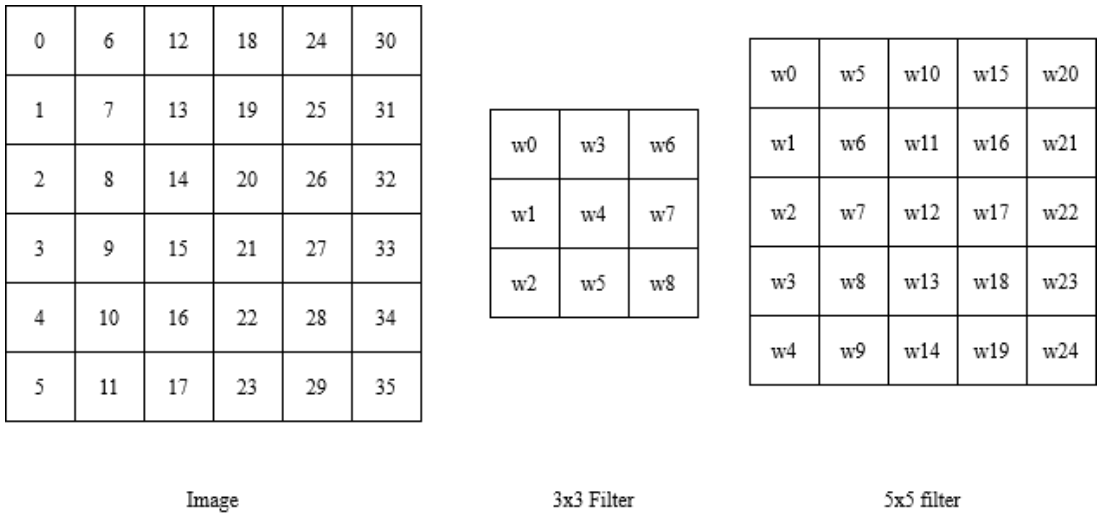

Figure 3.7: Convolution Operation

The pixel values are stored in the SRAM according to the figure. The pixels are arranged vertically in a column in the SRAM until the column is finished and the next column starts. This is done so that a pattern is available for the algorithm to predict the following address for the MAC operation. The filter will traverse along the image in the horizontal direction until the end of the image ends, and then it will go down by one pixel and then do the same process again. Currently, the algorithm only works for strides equal to 1, and further algorithm revision will add this feature.

The first 'for' loop is traversing across the image's different rows. First, we load the pixel values 0-2, 6-8, and 12 -14 in the input register file and the weights in their respective register file. This is executed in the 2nd, 3rd, and 4th 'for' loop. In the next convolution, 6-8 and 12-14 are repeated, so we don't have to bring them again. This will happen again and again until the end of the image. This is done using the 5th and 6th 'for' loop. This code is valid for any image size.

In the case of a 5×5 filter, the 1st 'for' loop is to traverse across the different rows of the image. Since there will be 25 pixels in the MAC, we must retrieve the first 16 pixels and then the next 9 pixels. This is done through the next 5 'for' loops. In the subsequent convolution, we can see that the pixel values 19-24 will repeat, so we don't have to fetch them from the SRAM. So, now we only have to fetch 11 pixels and then 9 pixels. This has to repeat until the filter traverses the entire row. This is achieved through the next 4 'for' loops. Thus, we need to create two different hardware blocks based on the filter being used in the convolution. These hardware blocks will automate

Algorithm 1 3x3 filter address generation algorithm

```
image_size Address
for i ← 0 to image_size do
    for j ← 0 to 3 do
        Address ←i + j end
        for j ← 0 to 3 do
            Address ←i + j+image_size
        end
        for j ← 0 to 3 do
            Address ←i + j + 2∗image_size
        end
        for k ← 1 to image_size − 2 do
            for l ← 0 to 3 do
                Address ←i + image_size∗k+l+image_size*2;
            end
        end
    end
```

the address generation logic and streamline the data flow.

3.5 Neural Networks

Neural networks, inspired by the human brain's neural structure, represent a powerful subset of machine learning algorithms. Comprising interconnected nodes or neurons organized into layers, they excel in recognizing patterns, solving complex problems, and making predictions. At the core lies the input layer receiving data, which then traverses through hidden layers, each consisting of neurons performing weighted computations and applying activation functions to produce an output. The intricate interconnections and synaptic weights between neurons enable these networks to learn from data, adjusting parameters through processes like backpropagation. Training involves exposing the network to vast datasets, allowing it to iteratively refine its internal mechanisms, minimizing errors, and enhancing accuracy. Neural networks find extensive applications across various domains, from image and speech recognition to natural language processing and autonomous vehicles, showcasing their adaptability and efficacy in addressing diverse real-world challenges.

3.5.1 Types of Neural Networks

Neural networks possess several types, each designed for specific tasks:

1. **Feedforward Neural Networks (FNNs):** The simplest form where information moves in one direction, from input nodes through hidden nodes to the output nodes. Multilayer Perceptrons (MLPs) are a common example, consisting of an input layer, one or more hidden layers, and an output layer.

2. **Recurrent Neural Networks (RNNs):** These networks are designed to work with sequential data, where connections between nodes form loops, allowing information to persist. They excel in tasks like time series prediction, natural language processing, and speech recognition.

3. **Convolutional Neural Networks (CNNs):** Primarily applied in image recognition and computer vision tasks, CNNs leverage specialized layers such as convolutional layers, pooling layers, and fully connected layers to automatically learn hierarchical representations of data.

4. **Generative Adversarial Networks (GANs):** Comprising two neural networks—the generator and the discriminator—GANs are used for generating synthetic data that resembles real data. They've found applications in creating images, videos, and even realistic text.

5. **Long Short-Term Memory Networks (LSTMs) and Gated Recurrent Units (GRUs):** Variants of RNNs designed to mitigate the vanishing gradient problem and effectively capture long-term dependencies in sequential data. They're prevalent in natural language processing and speech recognition tasks.

6. **Self-Organizing Maps (SOMs) and Autoencoders:** Unsupervised learning models used for dimensionality reduction, feature extraction, and data visualization tasks.

Neural networks continue to evolve with advancements like attention mechanisms, transformer architectures (e.g., BERT), and reinforcement learning, empowering them to tackle increasingly complex problems with improved efficiency and accuracy. The adaptability and versatility of these networks make them a cornerstone in the realm of artificial intelligence and machine learning.

3.5.2 Activation Functions

Activation functions are crucial components within neural networks, responsible for introducing non-linearities and enabling the network to learn and approximate complex functions. They operate on the output of a neuron, transforming it into the neuron's final output, which is then passed on to the next layer. Activation functions add flexibility to the network, allowing it to model and understand intricate relationships within the data

Several activation functions are commonly used in neural networks:

1. **Sigmoid Function:** This function squashes input values into a range between 0 and 1, which is useful for binary classification problems. However, its vanishing gradient problem—where gradients become extremely small for large inputs—can hinder training in deeper networks.

2. **Hyperbolic Tangent (Tanh) Function:** Similar to the sigmoid function, but it squashes input values into a range between -1 and 1. While it solves the vanishing gradient problem to some extent, it still suffers from the issue in deep networks.

3. **Rectified Linear Unit (ReLU):** Among the most popular activation functions, ReLU sets all negative values in the input to zero and keeps positive values unchanged. Its simplicity and computational efficiency make it effective, promoting faster convergence during training. However, it has a problem called "dying ReLU" where neurons may get stuck at zero and cease to update during training.

4. **Leaky ReLU:** A variation of ReLU that addresses the dying ReLU problem by allowing a small gradient for negative values, preventing neurons from being entirely inactive.

5. **Parametric ReLU (PReLU):** An extension of Leaky ReLU where the slope for negative values is learnable rather than fixed.

6. **Exponential Linear Unit (ELU):** Similar to ReLU but allows negative values with a smoother transition. It helps to mitigate the dying ReLU problem and has been shown to improve network performance.

7. **Swish:** Introduced by Google researchers, Swish functions as x multiplied by the sigmoid of x. It tends to perform well in various networks but might be computationally more expensive due to the sigmoid operation.

Activation functions are selected based on the specific requirements of the neural network and the nature of the problem being solved. Choosing the right activation function can significantly impact the network's learning speed, convergence, and generalization to new data. Researchers continue to explore new functions and modifications to improve neural network performance across various domains.

3.5.3 Pooling

Pooling is a technique commonly used in convolutional neural networks (CNNs) to downsample the spatial dimensions of feature maps, reducing the computational load while retaining the most important information. Pooling layers are interspersed between convolutional layers in CNN architectures. The primary types of pooling are:

1. **Max Pooling:** This is the most common pooling technique. In a max pooling operation, a window (usually 2×2 or 3×3) traverses the input feature map, and at each step, the maximum value within the window is selected to create the output. It retains the most prominent features within each window, discarding less relevant information. Max pooling helps make the network more robust to variations in input and reduces the spatial dimensions while retaining essential features.

2. **Average Pooling:** Instead of selecting the maximum value, average pooling computes the average value within the window. While it's simpler and computationally less expensive compared to max pooling, it might blur or lose subtle details present in the image.

3. **Global Pooling:** Global pooling reduces each channel in the feature map to a single value. Thus, an $nh \times nw \times nc$ feature map is reduced to $1 \times 1 \times nc$ feature map. This is equivalent to using a filter of dimensions $nh \times nw$ i.e. the dimensions of the feature map.

Pooling serves several purposes within CNNs:

- **Dimensionality Reduction:** By reducing the spatial dimensions (width and height) of feature maps, pooling decreases the number of parameters and computations in the network, preventing overfitting and reducing memory usage.

- **Translation Invariance:** Pooling helps achieve translation invariance, meaning that even if an object shifts slightly within the input image, the pooled features representing that object generally remain unchanged. This property aids in generalization and makes the network more robust to variations in the input.

- **Feature Aggregation:** Pooling aggregates similar features within the local neighborhood, enhancing the network's ability to recognize patterns and features irrespective of their precise spatial location.

- **Computation Efficiency:** By reducing the size of feature maps, subsequent layers require fewer computations, leading to faster training and inference times.

However, pooling also has some limitations. It discards spatial information, which might be crucial for detailed localization tasks. Moreover, recent advancements in neural network architectures, such as the use of strided convolutions or global average pooling, sometimes replace pooling layers to better preserve spatial information while achieving downsampling. In modern architectures, the use of pooling is often combined with other techniques like skip connections, dilated convolutions, or various attention

mechanisms to create more sophisticated models that effectively capture spatial hierarchies and dependencies within the data.

Neural networks, activation functions, and pooling are integral components of modern machine learning, each contributing uniquely to the success and efficiency of deep learning models. Neural networks, inspired by the human brain, have evolved into various architectures like feedforward networks, recurrent networks, and convolutional networks. These structures excel in learning from data, recognizing patterns, and making predictions across domains like image recognition, natural language processing, and more. Their adaptability and ability to handle complex tasks make them foundational in AI. Activation functions introduce non-linearities within neural networks, enabling them to learn complex relationships in data. From sigmoid and tanh to ReLU and its variations, each activation function has advantages and limitations. Choosing the right one impacts learning speed, convergence, and the network's ability to generalize to new data. Pooling, particularly in convolutional neural networks, helps reduce spatial dimensions while retaining essential features. Max pooling and average pooling are commonly used techniques for downsampling feature maps, aiding in computational efficiency, translation invariance, and feature aggregation. However, they can also discard spatial information, leading to potential loss of details. These components collectively form the backbone of deep learning models, constantly evolving as researchers explore new architectures, activation functions, and downsampling techniques. The synergy between these elements continues to push the boundaries of AI, enabling advancements in various fields and driving innovation across industries.

Chapter 4
Design of CNN Accelerator

After understanding the basics of number system and deep learning, we will jump into the design aspect of various blocks. This chapter describes how are the algorithm for various computations are converted into synthesizable hardware for FPGA. It also describes the interconnection between various blocks for implementing the complete LeNet on hardware accelerator.

4.1 Addition Block

Addition Blocks does addition of floating-point which are there in FP 8 (5,2) numbers. We cannot directly add the numbers in binary form. First, we need to align the exponent, before we add the mantissa part based on sign bit. The result would then need to be normalized in order to be consistent with the format.

In this algorithm, we first compare the exponents of two numbers. The larger exponent is taken into consideration and the smaller exponent number is converted by right shifting the mantissa bits of the smaller number by the amount represented by the difference between the exponents. We always convert the smaller number into the larger one because if we convert the larger number into the smaller one, the result would always be greater than 1, forcing it to always normalize the number.

We have designed the block in Xilinx Vivado, following the same logic. The discrete are explained below.

4.1.1 Top Module

Table 4.1: Signals for Top Module

Signal	Port Description
ip1	First Input Data
ip2	Second Input Data
out	Output Data

The top module takes two 8-bit input data and produce an output of 8 bit. This unit routes input to various algorithmic blocks to perform the addition. It also concatenates a 1 at the start of the mantissa according to the floating-point representation.

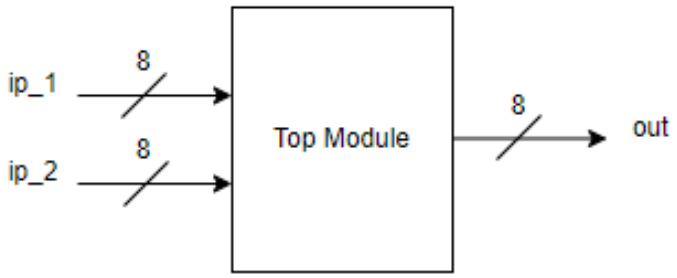

Figure 4.1: Top Module Representation

4.1.2 Comparator and Shifter

In this block, we first compare the exponential values of the two inputs. The mantissa of the smaller number is shifted right by the difference of exponents. Here, one change which we did is to already add 1 to the exponent value before adding. This is a prophylaxis move after analysing the fact that the sum is usually between 1 and 2 after adding the mantissa, and to normalize it, we will need to shift the mantissa by 1.

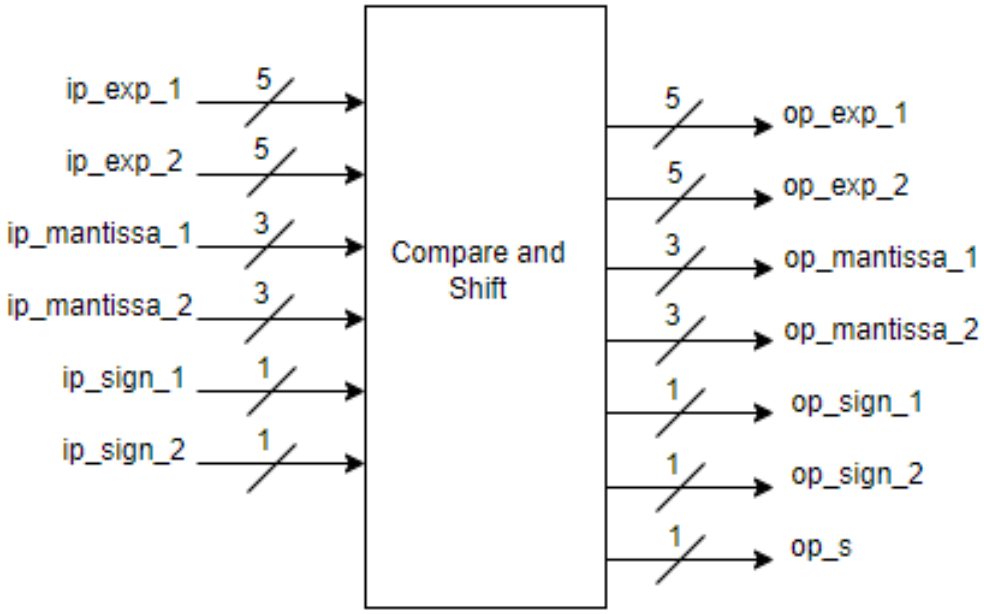

Figure 4.2: Compare and Shift block Representation

Table 4.2: Signals for Compare and Shift Block

Signal	Port Description
ip_exp_1	Exponent part of input data 1
ip_exp_2	Exponent part of input data 2
ip_mantissa_1	Mantissa part of input data 1
ip_mantissa_2	Mantissa part of input data 2
ip_sign_1	Sign bit of input data 1
ip_sign_2	Sign bit of input data 2
op_exp_1	Exponent part of input data 1 after shifting towards larger number
op_exp_2	Exponent part of input data 2 after shifting towards larger number
op_mantissa_1	Mantissa part of input data 1 after shifting
op_mantissa_2	Mantissa part of input data 2 after shifting
op_sign_1	Buffered sign bit of input data 1
op_sign_2	Buffered sign bit of input data 2
op_s	Comparison flag for magnitude of data after shifting

4.1.3 Addition and Subtraction

In this block, we first calculate the XOR of the sign bit and decide the operation to be done between the mantissa bit and the concatenated 1. If the signs are same, addition operation is performed and if they are different, subtraction operation is performed.

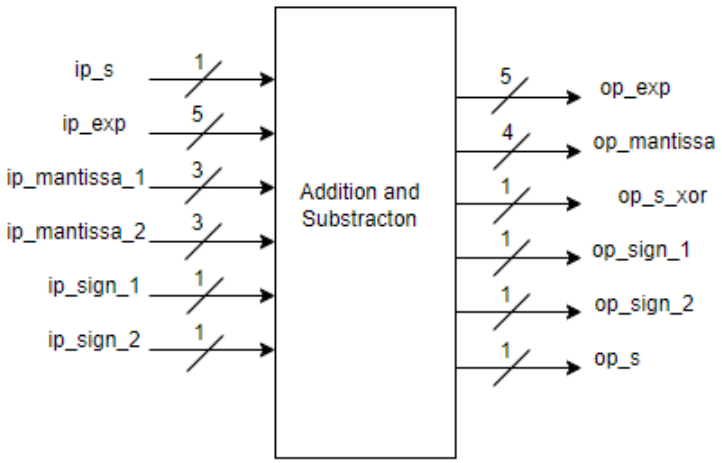

Figure 4.3: Addition and Subtraction block Representation

In this block, we first calculate the XOR of the sign bit and decide the operation to be done between the mantissa bit and the concatenated 1. If the signs are same, addition operation is performed and if they are different, subtraction operation is performed.

Table 4.3: Signals for Addition and Subtraction Module

Signal	Port Description
ip_mantissa_1	Mantissa part of input data 1 after shift
ip_mantissa_2	Mantissa part of input data 2 after shift
ip_exp	Common exponent part of both inputs
ip_sign_1	Sign bit of input data 1
ip_sign_2	Sign bit of input data 2
ip_s	Input flag from comparator and shift block
op_exp	Common exponent part of both inputs
op_mantissa	Mantissa part of output data 1 after addition or subtraction
op_s_xor	Xor between sign bit of two inputs
op_sign_1	Buffered sign bit of input data 1
op_sign_2	Buffered sign bit of input data 2
op_s	Buffered ip_s flag

4.1.4 Normalization

Till now, we have used a signal named "ip_s". This flag represents which input out of the two is bigger. This is because, in addition and subtraction block, we have performed the subtraction without comparing the mantissa value. This can let the answer of the addition and subtraction block to be negative in nature, and so in normalization block, we use the "ip_s" flag to take the 2's complement of the answer if we have performed the wrong addition.

If the MSB of mantissa does not start with 1, the output mantissa part will be left shifted by 1 and the exponent reduced by 1. This logic is used to normalize the output data in accordance to the format.

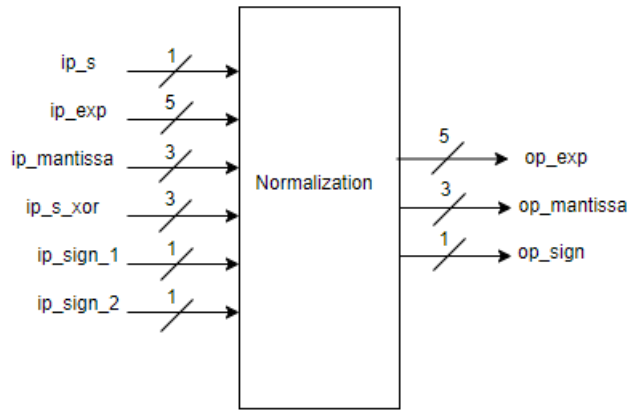

Figure 4.4: Normalization block Representation

Table 4.4: Signals for Normalization Module

Signal	Port Description
ip_mantissa	Mantissa after performing addition or subtraction operation
ip_s	Input flag from comparator and shift block
ip_exp	Common exponent part of both inputs
ip_sign_1	Sign bit of input data 1
ip_sign_2	Sign bit of input data 2
ip_s_xor	Xor between sign bit of two inputs
op_exp	Final exponent of the output data after normalization
op_mantissa	Final mantissa of the output data after normalization
op_sign	Final sign of the output data after normalization

4.2 Multiply and Accumulate Unit

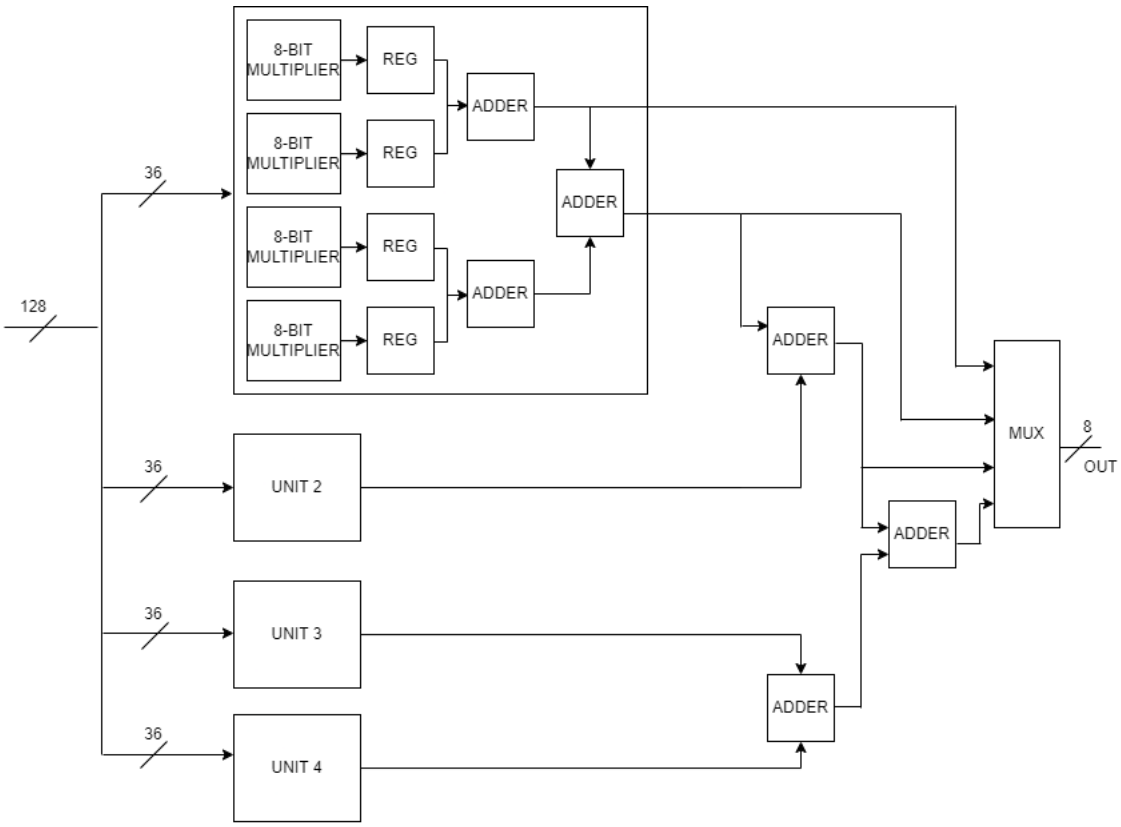

Figure 4.5: MAC Architecture [8].

We have used the MATAC architecture for the Multiplying and Accumulation Unit. In this architecture, we have used 4x4 Multiplier. We take 16, 8-bit input simultaneously and multiply it in the fashion, shown in the Figure 4.5. The multiplier and adder blocks specified in this are already discussed above. One change we did with regards to the original architecture was to provide forwarding of data after each multiply stage, this was done to access the results of 2×2, 4×4 and 8×8 multiplication operations. This was done because we would not always require the 16×16 multiplication operation and using all the stages would add to the latency.

4.3 Single PE Architecture

Latest Hardware Accelerators, like Simba [6] and Eyeriss [5], employ multi-chip-module-based integration. Each of these chiplets can be used as a standalone, edge-scale inference accelerator, while multiple chiplets can be packaged together to deliver data-centre-scale compute throughput. In modern day, each chiplet consist of an array of PEs, which are the heart of all computation operations performed by the accelerator, then a NoP router and a controller, which is usually a RISC-V core.

Each PE consist of weight buffers, input buffers, accumulation buffer, MAC units, Address generator blocks, Pooling Function and various activation functions needed for the DNN. Memory hierarchy is highly exploited in hardware accelerators keeping in mind the re-use. The problem lies in determining what kind of re-use to perform, here Timeloop/Accelergy comes to the rescue as it gives us the best mapping based on the architecture provided and various architectural and dataflow constraints provided by the designer.

Before implementing multi-level and multi-PE architecture, it is quite important to first start with the single level implementation to understand the various design complexities. We have designed a single PE based architecture containing single level of memory hierarchy. The architectural blocks are discussed below.

4.3.1 Buffers

As, currently we are focusing our implementation for FPGA we are using the BRAM resources to store data. Buffers are needed to store input image data, weights, and intermediate output from each layer.

The buffer has an enable signal which is an active high enable signal for the BRAM memory. It also consists of a write enable signal, which when high, allows the memory to be written. The output port of the memory always displays the data stored at the address provided by the input address signal, without considering the other input flags.

Three BRAM memories units are included to use as buffers. First one is to store the input image, second one is to store the weights, and the third one is to store the output of each layer.

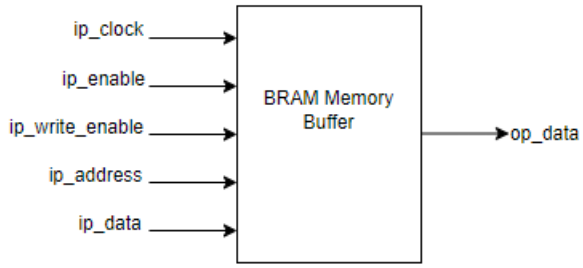

Figure 4.6: BRAM Memory

Table 4.5: Signals for BRAM memory buffer module

Signal	Port Description
ip_clock	Input clock
ip_enable	Enable signal for activating the memory for various operation
ip_write_enable	Enable signal for performing write operations for the memory
ip_address	Address from where data is to be written or read from
ip_data	Data to be written into memory
op_data	Data stored at a particular memory address

4.3.2 Register File

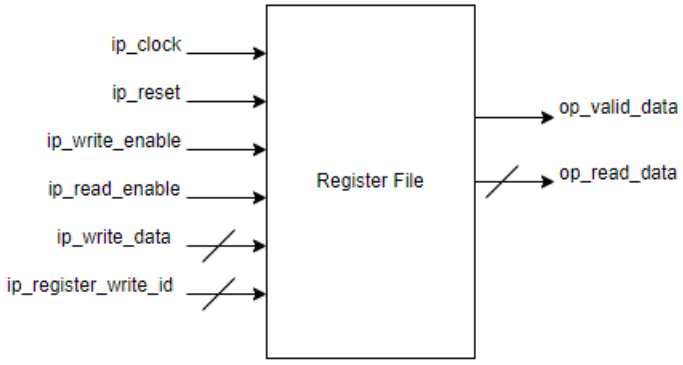

Figure 4.7: Register File

Register file is the set of register that is used a temporary fast memory. As we have already seen, the design contains a 4×4 MAC unit. This unit needs 8×16 = 128 bits

of input data simultaneously, so for this purpose the data is being read from the input buffer or the weight buffers and stored in register file before providing it simultaneously to the MAC. Another use of this register file is to provided data to the pooling unit simultaneously. 3 different register files are used – one to store input buffer data for convolution, second to store weight buffer data for convolution and third to store input buffer data for pooling.

Table 4.6: Signals for Register File module

Signal	Port Description
ip_clock	Input clock
ip_reset	Input reset to reset at register values to 0
ip_write_enable	Enable signal for performing write operations in the register file
ip_read_enable	Enable signal for performing read operations in the register file
ip_write_data	Data to be written into register file
ip_register_write_id	ID from where the data must be written or read from
op_valid_data	High when valid output is being produced
op_read_data	Data that is read from the register file at the address specified by the ID

4.3.3 MAC Unit

Discussed in detail in section 3.2.

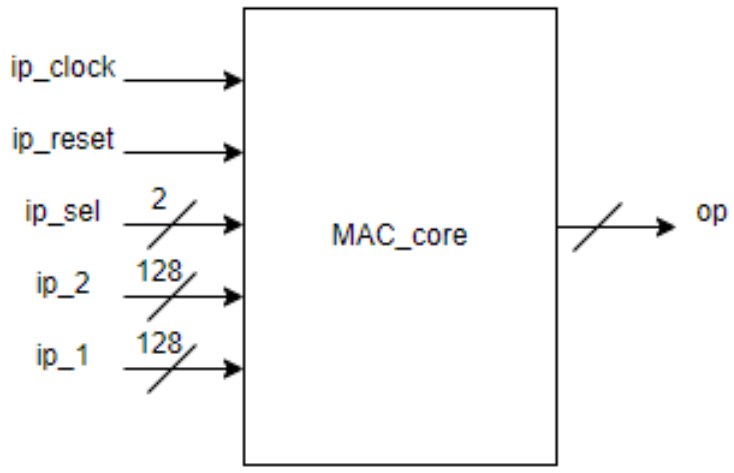

Figure 4.8: Multiply and Accumulate Unit

Table 4.7: Signals for Multiply and Accumulate module

Signal	Port Description
ip_clock	Input clock
ip_reset	Input reset
ip_sel	To select whether to perform 16×16 operation, 8×8 Operation, 4×4 Operation, or 2×2 Operations.
ip_1	Input data 1
ip_2	Input data 2
op	Output data

4.3.4 Pooling Unit

In Deep Neural Networks, pooling is one of the main operations. There are two types of pooling generally performed – average pooling and max pooling. In average pooling, the elements of the window are added and divided by the total number of elements. While, in max pooling, normal comparators are used. There arises a problem of division in average pooling which could be solved using 2 ways. In the first way, if the number of elements in a pooling window are in powers of 2, we could directly decrease the exponent. If this case is not applicable, then we would use the multiplication algorithm designed to multiply by the reciprocal of number of elements.

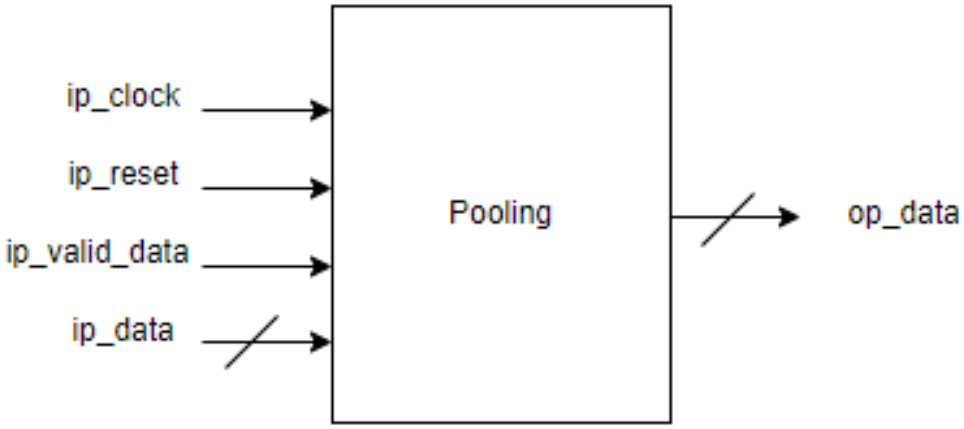

Figure 4.9: Pooling Logic Block

Table 4.8: Signals for Pooling Module

Signal	Port Description
ip_clock	Input clock
ip_reset	Input reset
ip_valid_data	To validate the incoming data
ip_data	Input data from pooling register file
op_data	Output data

4.3.5 Accumulator

During convolution operation, the values achieved after multiplication of the filter window needed to be added in order to get the output of convolution. Even after using 4*4 MAC architecture, if the number of elements to be added and multiplied is more than 16, we would need to use one MAC, twice for the same convolution. For example, in 5×5 filter convolution, we have use 9-16 technique, in which during the first operation, 16 elements are of the 25 elements are multiplied and added. This result is stored in an accumulator and then the second set of remaining 9 inputs are multiplied and added with the previously stored multiplied value. The need to only perform 9×9 operation, instead of 16×16 justifies our idea of bypassing values from each stage in the MATAC architecture. If the filter size is less than equal to 4×4, we do not need to use accumulator for addition as the MAC architecture can itself handle it.

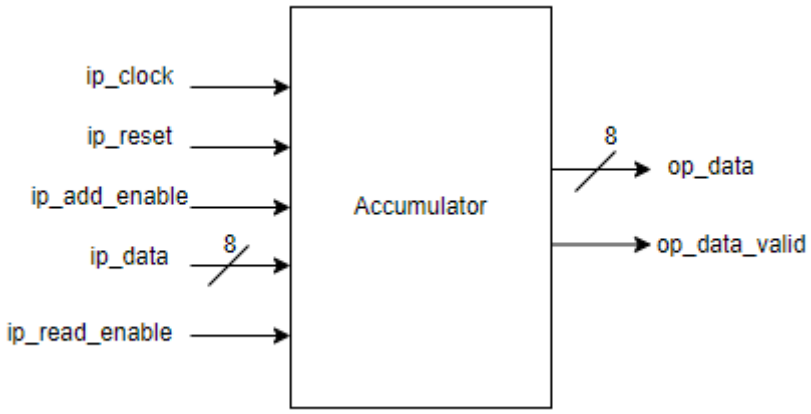

Figure 4.10: Accumulator

The accumulator accumulates the value only if add enable signal is asserted to 1. The value can only be read if read enable signal is asserted to 1. So, we first need to assert the add enable signal for 1 clock cycle, and then in the next clock cycle we will read the accumulated value.

Table 4.9: Signals for Accumulator module

Signal	Port Description
ip_clock	Input clock
ip_reset	Input reset
ip_add_enable	Previously stored value and current input value are added and stored
ip_read_enable	Signal to read stored value
ip_data	Input data
op_data	Output data
op_data_valid	Valid signal for output data

4.3.6 Address Generator for pooling

The logic for address generation is discussed in (3.4). The block provides necessary control signal for the transfer of data. First, pixel data is transferred from input buffer to the separate register file made for pooling action. Once all elements are received, the data is then transferred from register file to the pooling block. This is then stored in the output buffer for the next layer.

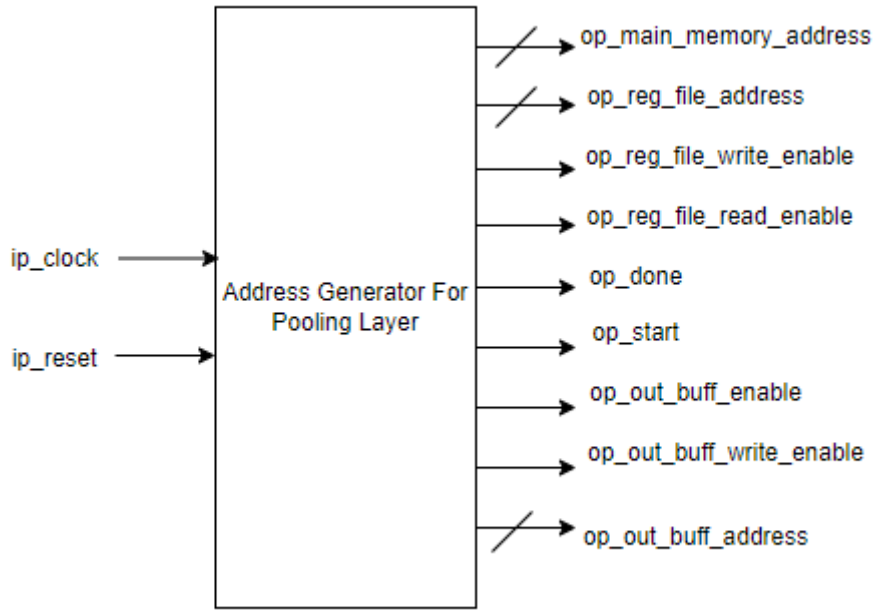

Figure 4.11: Address Generator for pooling layer

Table 4.10: Signals for Address generator for pooling layer module

Signal	Port Description
ip_clock	Input clock
ip_reset	Input reset
op_main_memory_address	Address to input buffer for read operation
op_reg_file_write_enable	Write enable signal to register file
op_reg_file_read_enable	Read enable signal to register file
op_reg_file_address	Read/Write id for register file
op_done	Done signal
op_start	Start signal
op_out_buff_enable	Output buffer enable
op_out_buff_write_enable	Output buffer write enable
op_out_buff_address	Output buffer address

4.3.7 Address generator for convolution

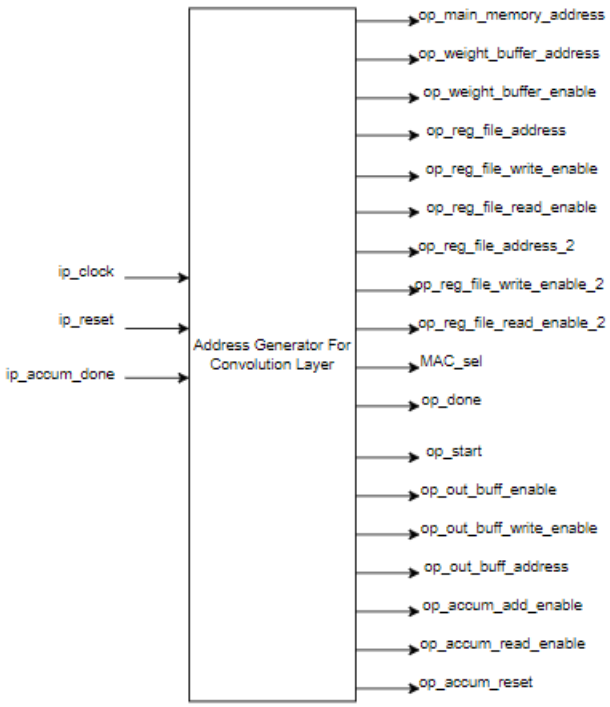

Figure 4.12: Address Generator for convolution layer

The logic for address generation is discussed in section 3.4. The block provides necessary control signal for the transfer of data. First, pixel data is transferred from input

buffer to the register file for input data according to the looping explained. Similarly, weights are also brought from weight buffer into the register file for weight data. These are then multiplied and accumulated. After a single convolution operation is completed, the data is stored in the output buffer. All the necessary control signals required for convolution operation are given by these blocks.

Table 4.11: Signals for Address generator for convolution layer module

Signal	Port Description
ip_clock	Input clock
ip_reset	Input reset
ip_accum_done	Input accumulator done signal
op_main_memory_address	Address to input buffer for read operation
op_weight_buffer_address	Address to weight buffer for read operation
op_weight_buffer_enable	Enable to weight buffer
op_reg_file_address	Read/Write id for register file
op_reg_file_write_enable	Write enable signal to register file
op_reg_file_read_enable	Read enable signal to register file
op_reg_file_write_enable_2	Write enable signal to register file of weights
op_reg_file_read_enable_2	Read enable signal to register file of weights
op_reg_file_address_2	Read/Write id for register file of weights
MAC_sel	To select from which stage of MAC output is needed
op_done	Done signal
op_start	Start signal
Op_accum_reset	Accumulator reset before new accumulation
Op_accum_add_enable	Accumulator add enable signal
Op_accum_read_enable	Accumulator read enable signal
Op_out_buff_enable	Output buffer enable
Op_out_buff_write_enable	Output buffer write enable
Op_out_buff_address	Output buffer address

4.3.8 Final Architecture

The final architecture after interconnecting all the necessary blocks is shown below. Two major control blocks of this architecture are the Address generator blocks for convolution and pooling operation. Either of them will be activated and data path will be selected.

For the convolution operation, the input data and the weights will be stored in the separate register files. This temporary memory allows data to be collected and passed

on to the multiplier and accumulator unit. After this, data is again accumulated for the single convolution operation. If the convolution filter size increases beyond the parallel input capacity of MAC, a second stage of accumulation would be required. After all the computations, data is being stored in output buffers for the next layers.

Now, for the pooling operation, the input data is brought into the register file. This is then passed on to the pooling block and finally data is being stored in the output buffers.

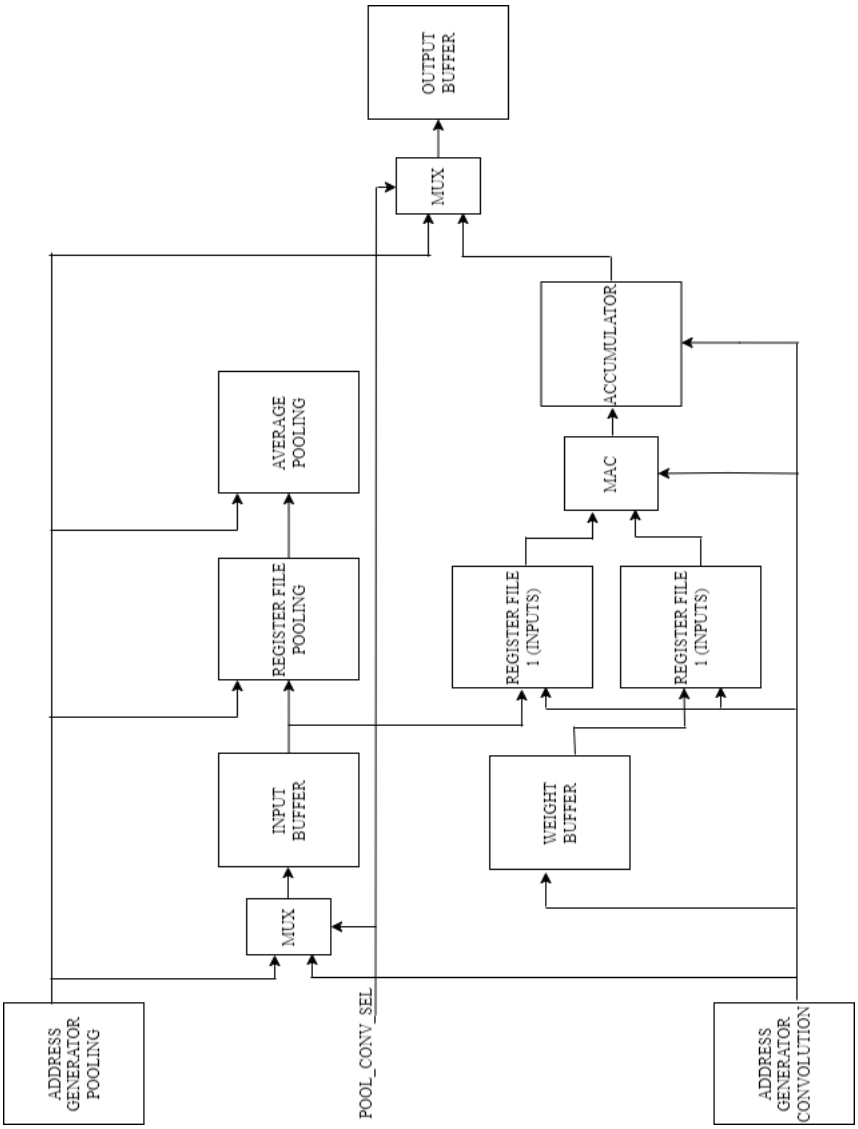

Figure 4.13: Final Architecture for complete implementation

Chapter 5
Results

This chapter presents the waveform and various other timing, utilization and power analysis of blocks used in the design of hardware accelerator. The synthesis is done on ZedBoard - Zynq SoC Development Board and platform used is Xilinx Vivado 2019.2.

5.1 Multiplier

The below figure shows the waveform for Multiplier output. The input to the multiplier are 1.25 and 1.5. The output data in the FP8 (5,2) format is 1.75, while the mathematical output is 1.875. This truncation, which might be seen as quite significant, doesn't affect the accuracy score as seen earlier.

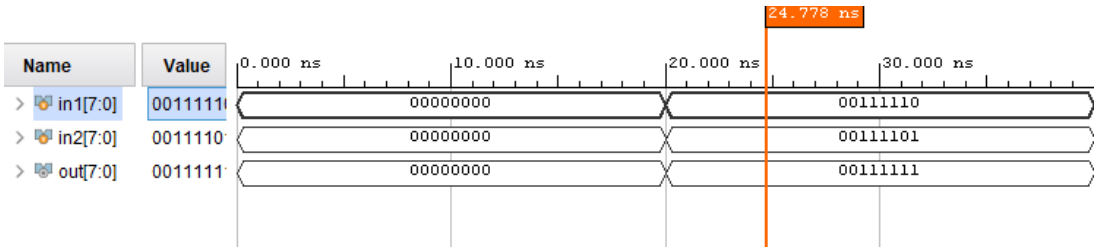

Figure 5.1: Multiplier Waveform

Table 5.1: Multiplier Analysis

	Parameter	Value
Timing	Logic Delay	3.904 ns
	Net Delay	2.988 ns
	Total Logic Delay	6.892 ns
Utilization	LUTs	15
	Flip Flops	0
Power	Dynamic	0.207 W
	Static	0.373 W
	Total-on chip	0.58 W

5.2 Adder

The below figure shows the waveform for Adder output. The input to the multiplier are 1.25 and 1.5. The output data in the FP 8(5,2) format is 2.5, while the mathematical output is 2.75. This truncation, which might be seen as quite significant, doesn't affect the accuracy score as seen earlier.

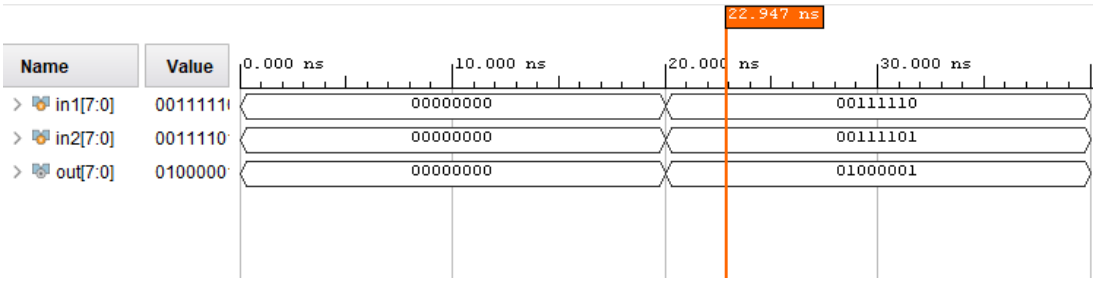

Figure 5.2: Adder Waveform

Table 5.2: Adder Reports

	Parameter	Value
Timing	Logic Delay	4.416 ns
	Net Delay	5.353 ns
	Total Logic Delay	9.797 ns
Utilization	LUTs	48
	Flip Flops	8
Power	Dynamic	0.301 W
	Static	0.456 W
	Total-on chip	0.757 W

5.3 MAC Core

The input given in the first stream is 0.0136, 16 times. While, the input given in the second stream is -0.0136, 16 times. The expected answer after the MAC operation is 2.99E-3, result achieved is 2.34E-3.

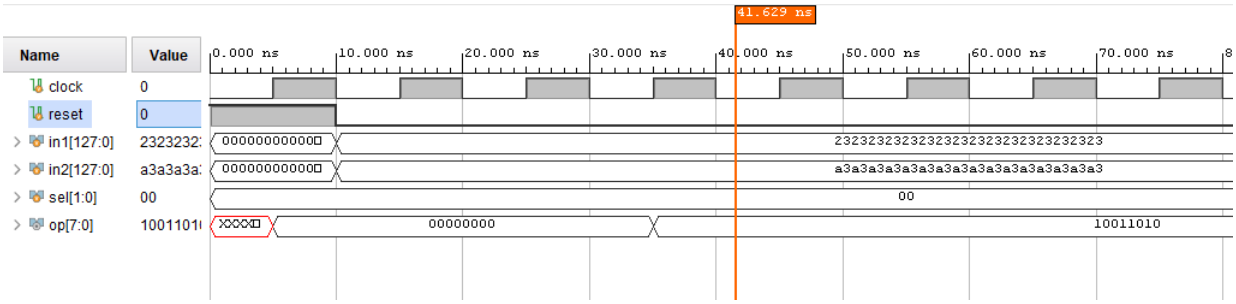

Figure 5.3: MAC Waveform

Table 5.3: MAC Reports

	Parameter	Value
Timing	Logic Delay	2.973 ns
	Net Delay	13.073 ns
	Total Logic Delay	16.046 ns
	Worst Negative Slack	0.911 ns
	Maximum Operating Frequency	41 Mhz
Utilization	LUTs	1199
	Flip Flops	512
Power	Dynamic	6.01 W
	Static	0.91 W
	Total-on chip	6.92 W

5.4 Average Pool

The below figure shows the waveform for Average Pool output. The input are 1.25, 4 times. The output data in the FP 8(5,2) format is 1.25, while the mathematical output is 1.25.

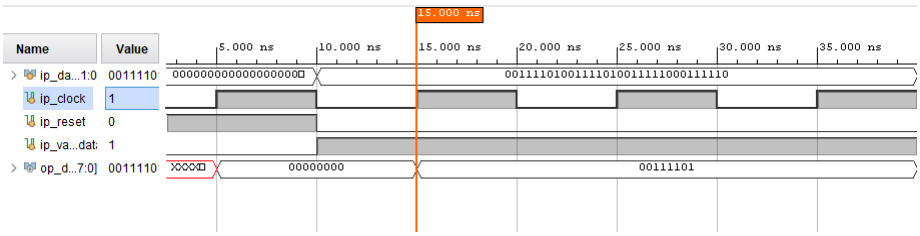

Figure 5.4: Average Pool Waveform

Table 5.4: Average Pool Reports

	Parameter	Value
Timing	Logic Delay	2.805 ns
	Net Delay	11.078 ns
	Total Logic Delay	13.883 ns
Utilization	LUTs	181
	Flip Flops	32
Power	Dynamic	5.775 W
	Static	0.456 W
	Total-on chip	6.23 W

5.5 Address Generator

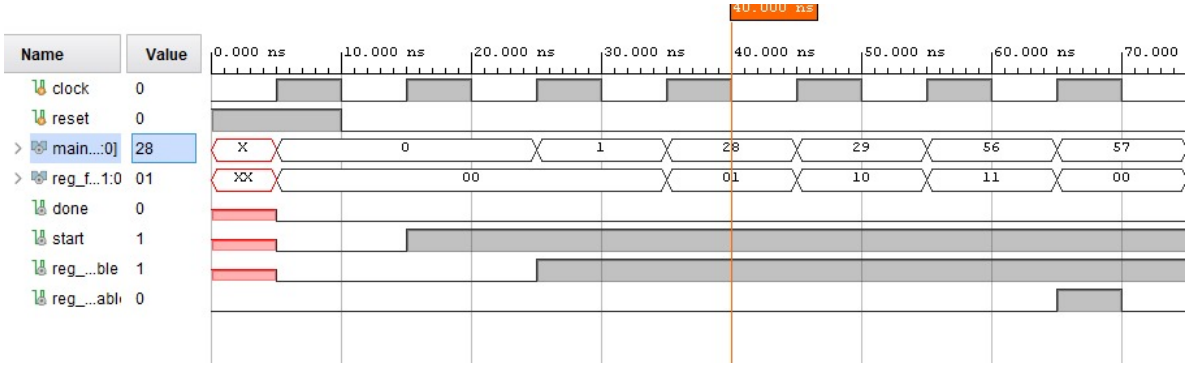

Figure 5.5: Address Generator Waveform

5.6 Complete Architecture

In this final waveform for pooling, we can analyse that the clock of double time period (clock-2) is provided to the address generator so that the generated address stays same for 2 clock cycles. This is because, the access time for the BRAM buffer is 2 cycles. The data read from the buffer is then transferred to the register file for pooling operation. We can see that that after 4 cycles between 70ns to 110 ns, 4 data points are fetched and this is transferred to the register file. After that, the input read signal is made high for the data to be read by the pooling block, after this the output is generated and stored in the output buffers. Same kind of output will be seen for pooling operation, where instead of 1 register file, 2 are used. And also the computation operation is done by the MAC core, rather than the pooling block.

In figure (5.7), the first 4 signals show the address generated by the address generator

for reading the data from input and weight buffers. This data is stored in register file 1 and register file 2 respectively. As we are simulating 5×5 filter convolution operation, we need to first bring 16 of the 25 data to the register file so that we could provide the complete data simultaneously to the MAC. After the MAC operation, data is stored in the accumulator. The remaining 9 data points are then read from the input buffers into the register file 1. While keeping the data in the register file 2 (weights), MAC operation is performed and final data is being produced after accumulation operation.

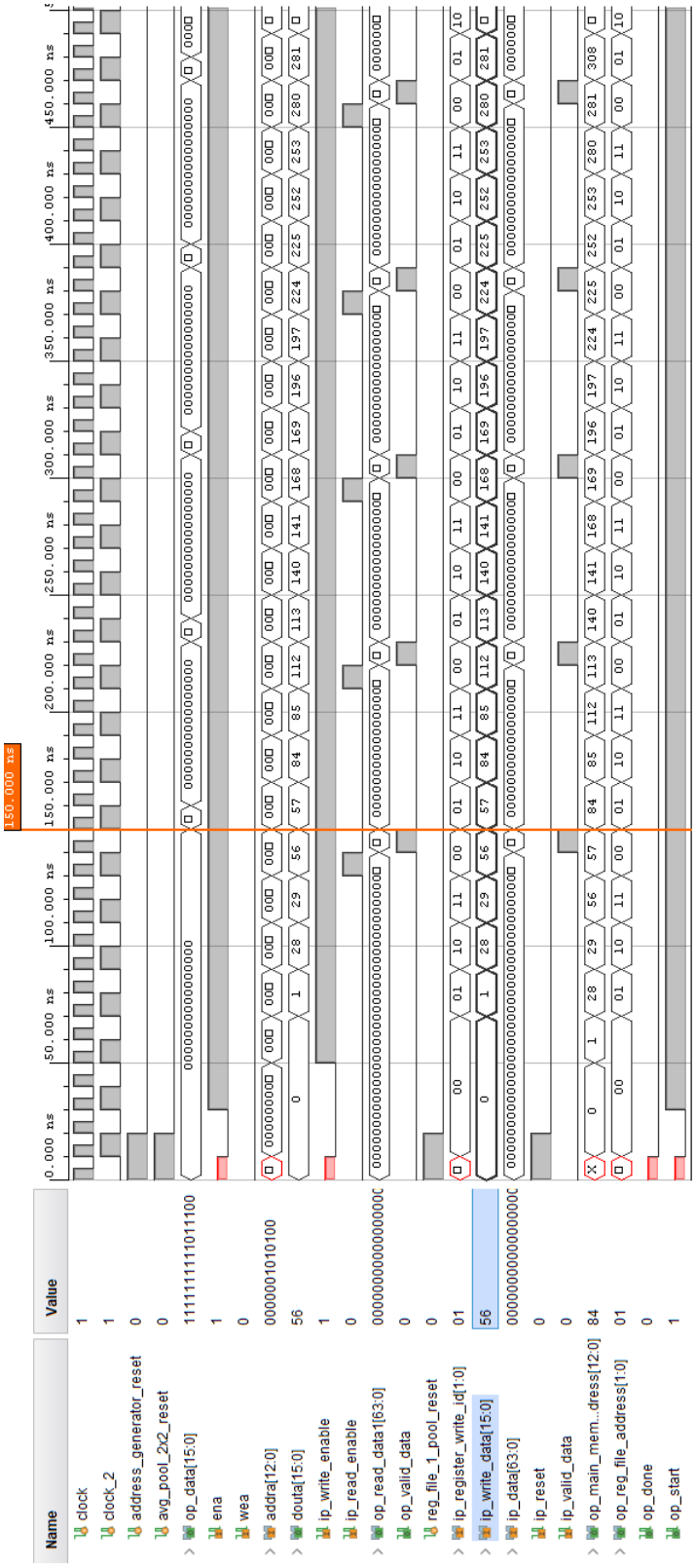

Figure 5.6: Waveform for integrated architecture for pooling operation

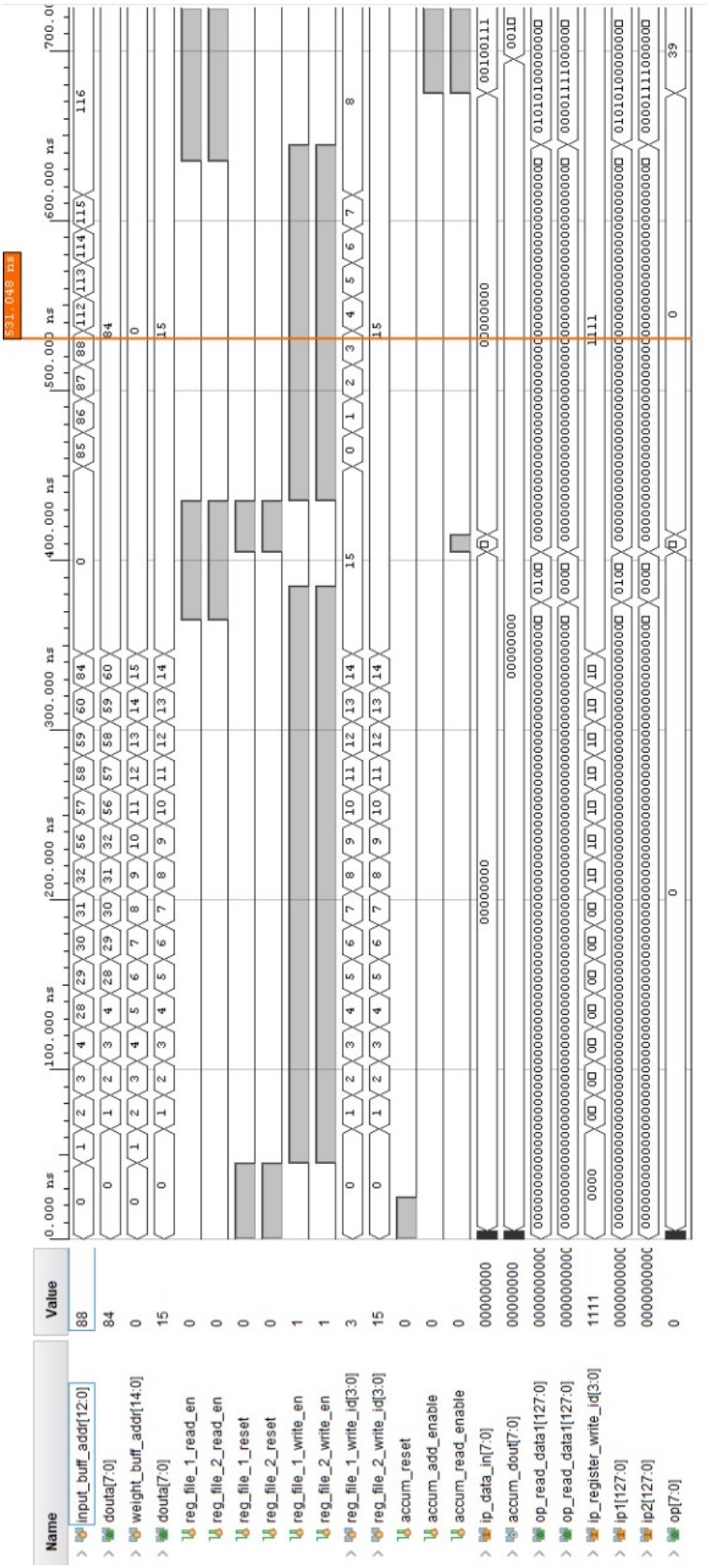

Figure 5.7: Waveform for integrated architecture for convolution operation

Chapter 6
Conclusion and Future Work

In this project, we have simulated the design of a single PE/single memory level hardware accelerator design for deep neural network architecture. In order to find the best floating point format, we simulated many floating point MAC units in Cadence Genus to find their performance. We found that FP8 (5E 2M) consumes 89.673 % less power, 76.334 % less area, and 51.364 % less clock period than its FP16 counterpart. The complete data path was designed and tested for two major operations: convolution and pooling. The discrete blocks needed for this project were designed using Xilinx Vivado 2019.2.

For future work, first, we would implement this hardware accelerator on a field programmable gate array as well as test a complete CNN on our design using hardware-software co-design. Further, we plan to include more pooling and activation functions like max pooling, sigmoid, tanh, and many more hardware implementations in our design to provide additional functionality. Also, we can modify the address generation algorithm to factor in stride in the address. In the future, for larger networks like BERT and GPT, we have to create multiple processing elements (PE) to take advantage of parallelism in the workload of neural networks. For that, we have to create a three-level memory hierarchy of caches and a network-on-chip (NoC) to facilitate the transfer of data between PEs and implement the design on a field programmable gate array. After testing the design, we can go towards the ASIC implementation of our design.

References

[1] "Tensorrent grayskull." [Online]. Available: https://fortune.com/2021/05/21/tenstorrent-a-i-chips-nvidia-keller-funding/

[2] "Cerebras wafer-scale engine." [Online]. Available: https://www.cerebras.net/product-chip/

[3] H. Genc, S. Kim, A. Amid, A. Haj-Ali, V. Iyer, P. Prakash, J. Zhao, D. Grubb, H. Liew, H. Mao *et al.*, "Gemmini: Enabling systematic deep-learning architecture evaluation via full-stack integration," in *2021 58th ACM/IEEE Design Automation Conference (DAC)*. IEEE, 2021, pp. 769–774.

[4] S. Lee, K. Kim, S. Oh, J. Park, G. Hong, D. Ka, K. Hwang, J. Park, K. Kang, J. Kim *et al.*, "A 1ynm 1.25 v 8gb, 16gb/s/pin gddr6-based accelerator-in-memory supporting 1tflops mac operation and various activation functions for deep-learning applications," in *2022 IEEE International Solid-State Circuits Conference (ISSCC)*, vol. 65. IEEE, 2022, pp. 1–3.

[5] Y.-H. Chen, T. Krishna, J. S. Emer, and V. Sze, "Eyeriss: An energy-efficient reconfigurable accelerator for deep convolutional neural networks," *IEEE journal of solid-state circuits*, vol. 52, no. 1, pp. 127–138, 2016.

[6] Y. S. Shao, J. Clemons, R. Venkatesan, B. Zimmer, M. Fojtik, N. Jiang, B. Keller, A. Klinefelter, N. Pinckney, P. Raina *et al.*, "Simba: Scaling deep-learning inference with multi-chip-module-based architecture," in *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*, 2019, pp. 14–27.

[7] "ieee 754 format." [Online]. Available: https://en.wikipedia.org/wiki/IEEE_754-1985

[8] S.-Y. Lin, K.-H. Lin, C.-K. Tsai, and P.-H. Tseng, "Reconfigurable mac systolic array architecture design for three-dimensional convolution neural network," in *2020 IEEE International Conference on Consumer Electronics-Taiwan (ICCE-Taiwan)*. IEEE, 2020, pp. 1–2.

[9] T. Glint, K. Prasad, J. Dagli, K. Gandhi, A. Gupta, V. Patel, N. Shah, and J. Mekie, "Hardware-software codesign of dnn accelerators using approximate posit multipliers," in *Proceedings of the 28th Asia and South Pacific Design Automation Conference*, 2023, pp. 469–474.