TECHNISCHE UNIVERSITÄT DRESDEN

Faculty of Computer Science Institute of Computer Engineering, Chair of Adaptive Dynamic Systems

Research Project Work

Implementation of a Configurable Neural Processing Unit with Input and Weight Stationary Dataflows

Rehan Karthik Chandralal

Born on: February 5, 1998
Course: Nano-electronic Systems
Matriculation number: 5127834

April 28, 2024

Supervisor
Ahmed Kamaleldin Atef

Supervising professor
Prof.Dr.-Ing Diana Göhringer

TECHNISCHE UNIVERSITÄT DRESDEN

Fakultät Informatik Institut für Technische Informatik, Professur für Adaptive Dynamische Systeme

Task for the preparation of a Project Work

Course:	Nanoelectronic Systems
Name:	Rehan Karthik Chandralal
Matriculation number:	5127834
Title:	Implementation of a Configurable Neural Processing Unit with Input and Weight Stationary Dataflows

Objectives of work

Typical neural processing units (NPUs) are implemented with fixed dataflow models. Based on the state-of-the-art, four types of dataflow models are supported to exploit data reuse and computing parallelism. The four types of dataflow models are input-, output-, weight-, and row-stationary. In this project work, the student has to implement a configurable NPU that can be configured during design time to support input- and weight-stationary dataflow. The NPU internal architecture should consist of an array of PEs connected to input/weight/output buffers through a configurable interconnect. The NPU implementation has to be developed using RTL targeting a Xilinx FPGA device. Also, the student has to evaluate the NPU using several convolution types with different convolution layers parameters. The evaluation should cover several factors related to execution time, resource utilization, and power consumption while using input- and weight-stationary dataflows.

Focus of work

- Studying and understanding several state-of-the-art related to flexible dataflow implementations for NPUs.
- Implementation of a configurable dataflow NPU supporting input- and weight-stationary dataflow models:
  - Development and implementation of an array of PEs, each PE implements a MAC operation. The PE array can be implemented based on a systolic array architecture.
  - Development of a configurable/flexible interconnect between NPU buffers and PE array to support both input- and weight-stationary dataflow.
  - Implementation of a control unit to manage data transfer between PE array and data buffers based on the selected dataflow model.
- Evaluation and documentation:
  - Evaluating the developed NPU in terms of execution time, resource utilization, and power consumption for the two types of dataflow models using several convolution types with different convolution layers parameters.
  - Documenting the implementation and obtained results in a form of a project work report.

Referee:	Prof. Dr.-Ing. Diana Göhringer
Supervisor:	Ahmed Kamaleldin
Issued on:	October 9, 2023
Due date for submission:	March 30, 2024

TECHNISCHE UNIVERSITÄT DRESDEN

Faculty of Computer Science Institute of Computer Engineering, Chair of Adaptive Dynamic Systems

Statement of authorship

I hereby certify that I have authored this document entitled *Implementation of a Configurable Neural Processing Unit with Input and Weight Stationary Dataflows* independently and without undue assistance from third parties. No other than the resources and references indicated in this document have been used. I have marked both literal and accordingly adopted quotations as such. During the preparation of this document I was only supported by the following persons:

Ahmed Kamaleldin Atef

Additional persons were not involved in the intellectual preparation of the present document. I am aware that violations of this declaration may lead to subsequent withdrawal of the academic degree.

Dresden, April 28, 2024

Rehan Karthik Chandralal

TECHNISCHE UNIVERSITÄT DRESDEN

Faculty of Computer Science Institute of Computer Engineering, Chair of Adaptive Dynamic Systems

Abstract

A Neural Processing Unit (NPU) is a specialized hardware component designed to accelerate neural network computations. Inspired by the structure and function of the human brain, NPUs are optimized for performing complex mathematical operations involved in training and inference tasks of artificial neural networks. Offloading neural network computations from traditional CPUs or GPUs to dedicated NPUs can yield significant improvements in performance, power efficiency, and scalability. This makes NPUs well-suited for deployment in various applications such as image and speech recognition, natural language processing, autonomous vehicles, and more.

Different neural network models (e.g., CNNs, RNNs, Transformers) have varying computational requirements and dataflow patterns. A configurable NPU allows users to tailor the hardware architecture and dataflow to match the specific characteristics of their model, optimizing performance and efficiency.

This Research project work aims to design a Configurable Neural Processing Unit that can support both input and weight stationary dataflows. The Configurable NPU is centered around a 9x9 Processing Element (PE) array, complemented by a flexible interconnect module. This setup allows for dynamic adjustment to various convolution layer parameters, such as input feature map size, weight size, and data width. The synthesis and implementation targeting the Xilinx ZCU104 evaluation board have yielded a maximum operating frequency of 140 MHz. By prioritizing flexibility in design, the configurable NPU aims to offer enhanced adaptability and customization, paving the way for optimized neural network acceleration.

Contents

List of Figures

List of Tables

List of Abbreviations

AI	Artificial Intelligence. 1
AS	Adder switch. 7
AXI	Advanced eXtensible Interface. 11, 13, III
BVPE	Bit-vector processing engines. 9
CMOS	Complementary Metal-Oxide-Semiconductor. 5
CNN	Convolutional Neural Network. 1–4, 6, 7, 13, 22, 30, III
CONV	Convolutional Layer. 4, 5, IV
CPU	Central Processing Unit. 1, D
DNN	Deep Neural Network. 1, 2, 5, 7, 8
DSP	Digital Signal Processing. 20, 24, 30
FC	Fully Connected Layer. 4, 5, IV
FPGA	Field-Programmable Gate Array. 7, 20, 23, 30
FSM	Finite State Machine. 20, 22
GPU	Graphics Processing Unit. 1, 16, D
MAC	Multiply-Accumulate. 2, 5, 18–20, 24, 30
MAERI	Microarchitecture for Accelerating Efficiently Reconfigurable Interconnects. 7
MS	Multiplier switch. 7
NPU	Neural Processing Unit. 1–3, 5, 6, 9–12, 17, 19, 20, 23, 25, 30, 31, IV, D
PB	Prefetch buffer. 7
PCM	Phase Change Memory. 5
PE	Processing Element. 2, 5, 8, 10, 13, 14, 17–20, 22, 24, 30
ReRAM	Resistive Random Access Memory. 5
SS	Simple switch. 7
TPU	Tensor Processing Unit. 16

1 Introduction

The significant progress of Artificial Intelligence (AI) in recent years can be largely attributed to the advancements made in neural networks. These networks have become a cornerstone technology across various fields, encompassing image and speech recognition, autonomous vehicles, and even medical diagnostics.

At the heart of efficient and effective neural networks lies the Neural Processing Unit (NPU). This specialized hardware accelerator is meticulously designed to execute neural network computations with unrivaled speed and energy efficiency, as referenced in [1]. While traditional processors like Central Processing Unit (CPU)s and Graphics Processing Unit (GPU)s have historically been the workhorses of computation, they struggle to keep up with the demands of deep learning algorithms, which heavily rely on parallel processing capabilities. NPUs, on the other hand, are custom-built to address these specific needs, boasting architectures that are finely tuned to exploit the inherent parallelism present in neural network operations.

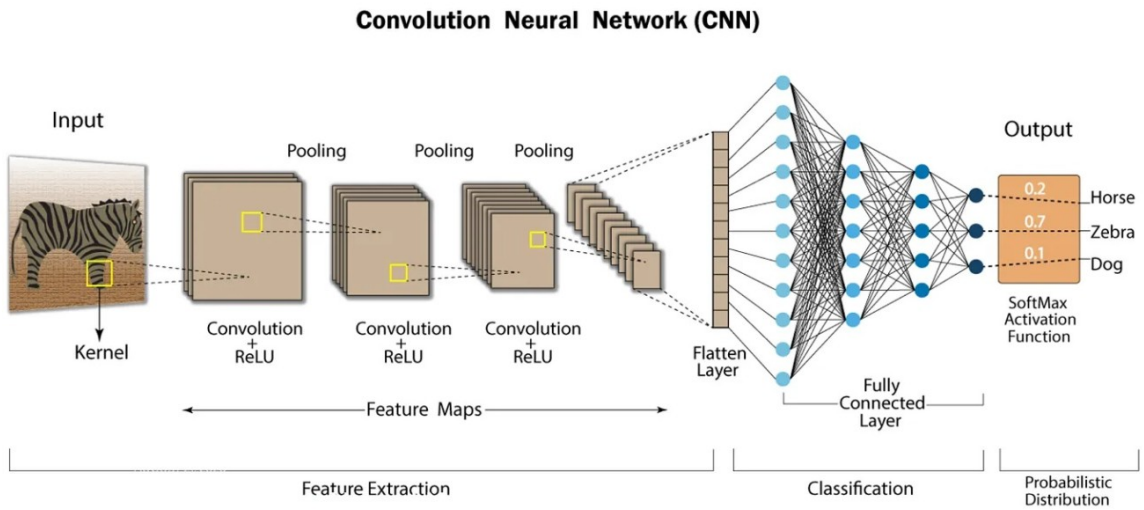


Figure 1.1: Convolutional Neural Network Schematic [2]

NPUs play a critical role in accelerating the inference phase of Convolutional Neural Network (CNN)s and Deep Neural Network (DNN)s, two prominent architectures in the field of artificial intelligence.

Figure 1.1 represents a typical CNN which performs feature extraction, classification and probability distribution calculation.CNNs are particularly effective in tasks such as image

recognition and object detection, thanks to their ability to extract features hierarchically through convolutional layers. NPUs enhance the performance of CNN by efficiently executing the numerous convolutional and pooling operations involved in the inference process [1].

A DNN comprising of multiple layers of interconnected nodes, are utilized in various applications including speech recognition and natural language processing. NPUs accelerate the computation-heavy tasks within DNN, such as matrix multiplications and activation functions, thereby improving inference speed and energy efficiency [3].

1.1 Motivation and Objective

The project is motivated by the need for a more flexible and efficient NPU. Traditional NPUs are implemented with fixed dataflow models, limiting their adaptability and performance in different scenarios.These limitations can be compensated by implementing a configurable NPU that can support different dataflow models, specifically input- and weight-stationary dataflows.

The objectives of this project include:

- Creating a Processing Element (PE) array for Multiply-Accumulate (MAC) operations, potentially utilizing a systolic array architecture.
- Designing a flexible interconnect to facilitate efficient data movement between NPU buffers and the PE array.
- Implementing a control unit to orchestrate data transfers based on the chosen dataflow model.

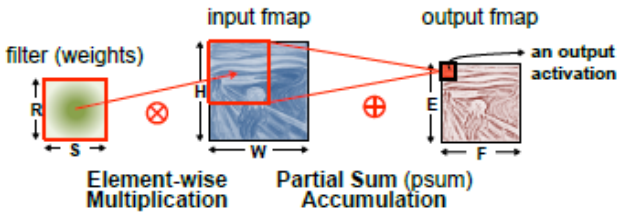
Moreover , a literature review was conducted before starting with the design of the project to understand several state-of-the-art related to flexible dataflow implementations for NPUs.

2 Background and Literature Survey

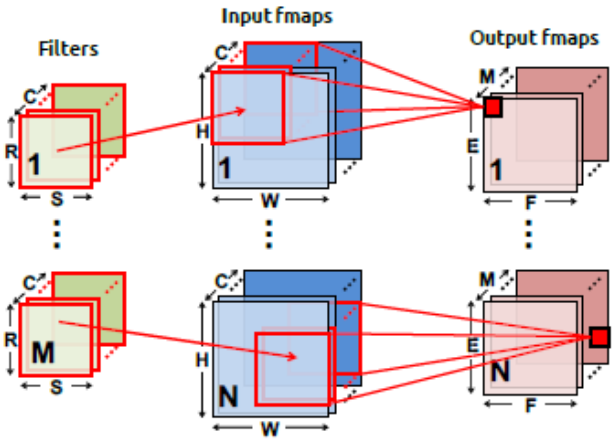
In this chapter, an introduction to convolution using NPU is provided along with some information about different kinds of dataflows and a literature survey about different state of the art flexible dataflow implementations for NPUs.

2.1 Introduction to Convolution using an NPU

Each of the convolutional layers in the CNN is primarily composed of high-dimensional convolutions as shown in Figure 2.1 .



(a) 2-D convolution in traditional image processing



(b) High dimensional convolutions in CNNs

Figure 2.1: Convolution Operation in CNN [4]

In CNNs, each layer generates a successively higher-level abstraction of the input data, called a feature map (fmap), which preserves essential yet unique information.In this computation, the input activations of a layer are structured as a set of 2-D input feature maps (ifmaps), each of which is called a channel [4]. Each channel is convolved with a distinct 2-D filter from the stack of filters, one for each channel; this stack of 2-D filters is often referred to as a single 3-D filter.

Table 2.1: Shape Parameters of a Convolutional Layer (CONV) or Fully Connected Layer (FC) Layer

Parameter	Description
N	Batch size of 3-D feature maps
M	Number of 3-D filters / Number of output feature map channels
C	Number of input feature map / filter channels
$H = W$	Input feature map plane height/width
$R = S$	Filter plane height/width (equal to H or W in FC)
$E = F$	Output feature map plane height/width (equal to 1 in FC)

The results of the convolution at each point are summed across all the channels. In addition, a 1-D bias can be added to the filtering results to form the output activations that comprise one channel of output feature map (ofmap). Additional 3-D filters can be used on the same input to create additional output channels. Finally, multiple input feature maps may be processed together as a batch to potentially improve reuse of the filter weights. Given the shape parameters in Table 2.1, the computation of a CONV layer is defined as

$$O[z][u][x][y] = B[u] + \sum_{k=0}^{C-1} \sum_{i=0}^{S_x-1} \sum_{j=0}^{R_x-1} I[z][k][Ux+i][Uy+j] \cdot W[u][k][i][j] \tag{2.1}$$

with $0 \leq z < N$, $0 \leq u < M$, $0 \leq x < F$, and $0 \leq y < E$,$E = \frac{H-R+U}{U}$ and $F = \frac{W-S+U}{U}$.

where O, I, W and B are the matrices of the ofmaps, ifmaps, filters and biases, respectively. U is a given stride size.figure 2.1 shows shows a visualization of this computation (ignoring biases).In addition to CONV and FC layers, various optional layers are also present such as the non-linearity, pooling,and normalization represented in Figure 2.2.

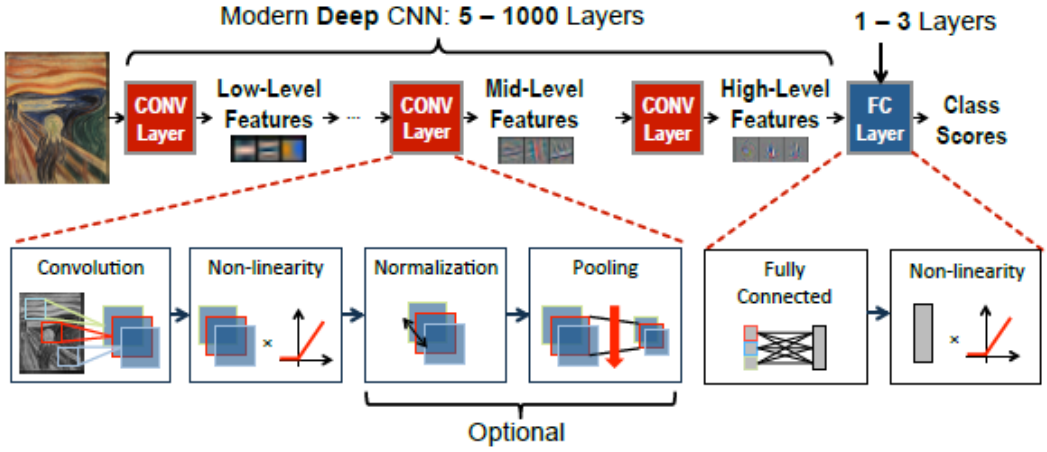


Figure 2.2: Different Layers in a CNN [4]

NPUs are designed to accommodate a reasonable amount of MAC units, which form the PEs used in the CONV and FC layers of DNNs.Each PE contains a synaptic weight buffer and the units to perform the computation of a neuron, namely, multiplication, accumulation, and an activation function (e.g., sigmoid) [5]. A PE can be realized entirely with a full-Complementary Metal-Oxide-Semiconductor (CMOS) design or by using emerging non-volatile memories such as Resistive Random Access Memory (ReRAM) and Phase Change Memory (PCM) to perform in situ matrix-vector multiplication as in the RENO chip [6].

2.2 Different kinds of data flows

Dataflow techniques play a pivotal role in optimizing the processing efficiency of DNN hardware accelerators [7].The main aim of these dataflow techniques is to reduce data movement between several parts of the architecture, which is a major consumer of energy. The four major dataflow techniques include:

- Input stationary
- Weight stationary
- Output stationary
- Row stationary

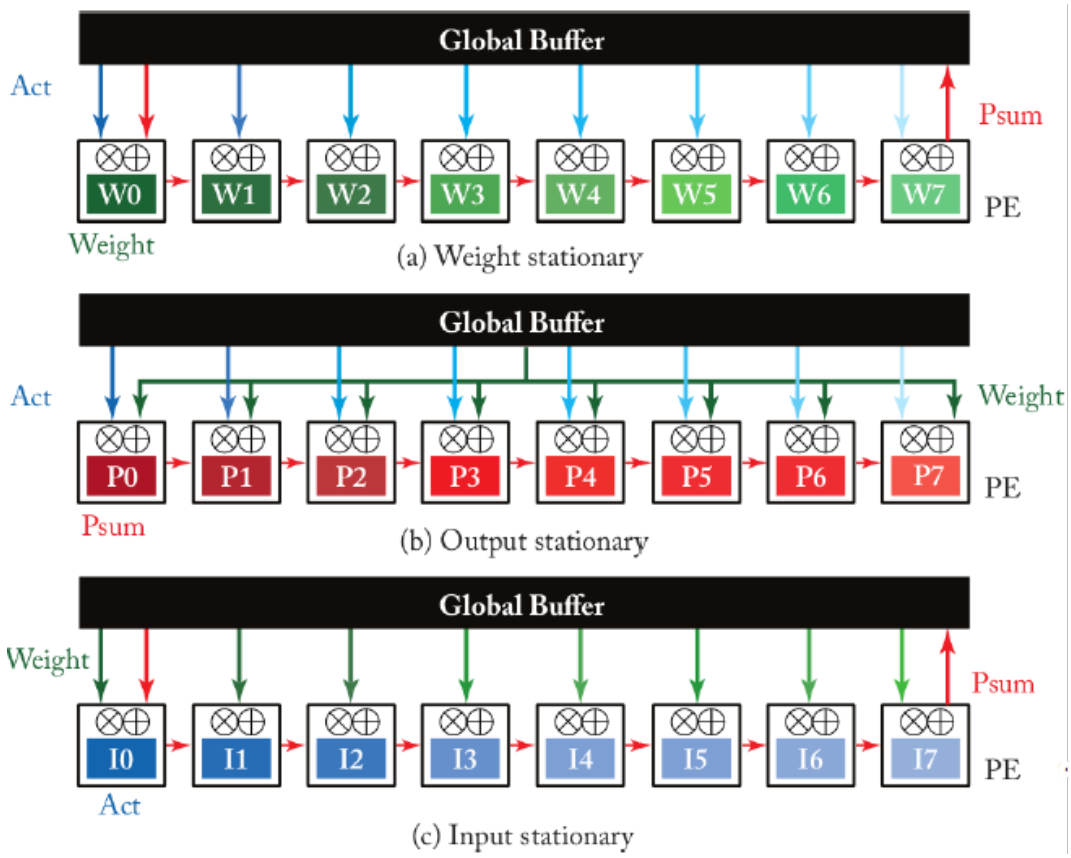


Figure 2.3: Weight,Output and Input stationary dataflows [7]

With the increasing diversity of neural networks accelerators supporting multiple dataflows are more efficient than those only supporting a specific dataflow. It helps make neural networks run faster, use less energy, and work well with different types of tasks.Thus the focus of this project is to design an NPU that can support two different kinds of data-flows (namely input and weight stationary) as seen in Figure 2.3.

2.2.1 Input stationary data flow

In Input stationary data flow, the input data remains fixed while the weights are learned during training. This setup allows for the repeated processing of the same input data (data re-use) through the network, while the weights are shared or applied uniformly across different regions of the input.

In CNNs the same filter or kernel across various spatial locations of the input to efficiently learn spatial hierarchies of features. The benefits of input stationary data flows lie in **parameter sharing**, which reduces the number of learnable parameters and promotes feature reuse across different parts of the input and is particularly advantageous for spatial data tasks, such as image recognition, where local patterns are crucial for accurate predictions.

2.2.2 Weight stationary data flow

In Weight stationary data flow, the weights remain fixed while the input data is processed during training and inference. This arrangement allows for consistent application of the same weights across different instances of the input, promoting **parameter reuse and reducing redundancy** in the network.

Particularly prevalent in architectures like CNNs, weight stationary data flows involve sharing or applying weights uniformly across various input regions. By adopting weight stationary data flows, neural networks can effectively capture patterns and relationships in the input data, facilitating tasks such as image recognition where weight sharing across different spatial locations is essential for accurate predictions.

2.3 Literature Survey

Microarchitecture for Accelerating Efficiently Reconfigurable Interconnects (MAERI) proposes a DNN accelerator architecture that tackles the challenge of supporting diverse dataflow patterns within DNN.MAERI enables dynamic adaptation of dataflow within the accelerator architecture, optimizing resource utilization and performance for diverse DNN workloads [8].

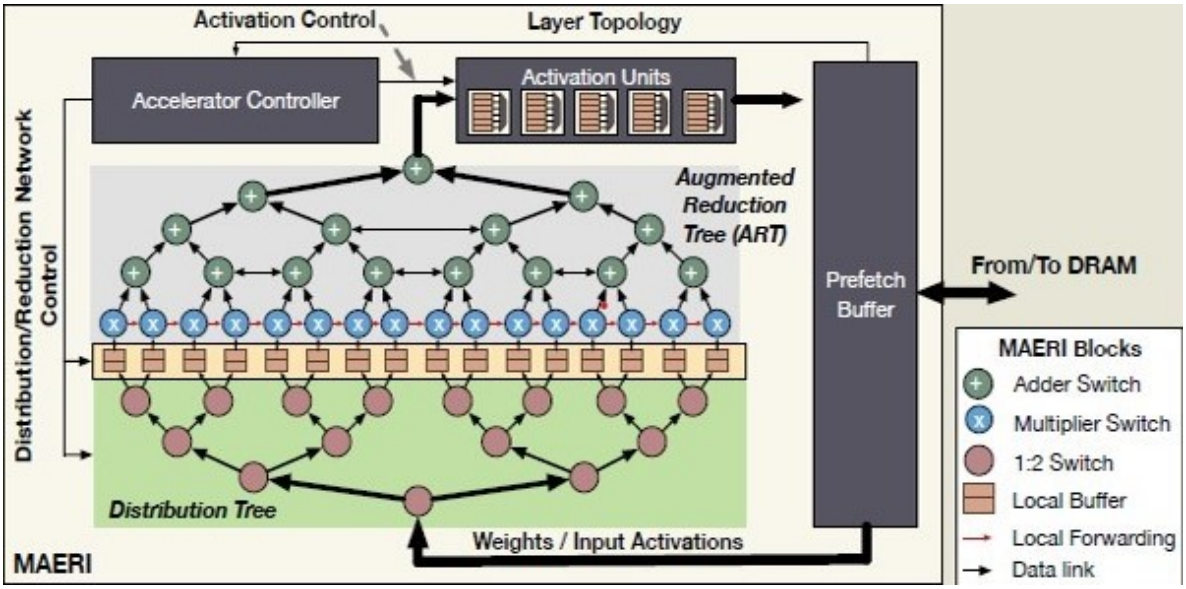


Figure 2.4: An overview of Maeri [8]

Figure 2.4 represents the MAERI architecture having 2 configurable switches - Multiplier switch (MS) and Adder switch (AS) - enabling optimization for collective communication patterns. Two configurable interconnection networks are employed: a distribution network, utilizing tiny simple switches (SS) to send inputs to MSs from the Prefetch buffer (PB), and a reduce+collect network, employing ASs to send outputs back to the PB via activation units implemented with lookup tables. A programmable controller oversees the entire accelerator, managing the reconfiguration of all three sets of switches (MS,AS, and Simple switch (SS)) for mapping the target dataflow.

A flexible dataflow CNN accelerator implemented on a Field-Programmable Gate Array (FPGA) offers a versatile and customizable solution for accelerating CNN computations. By leveraging FPGA's reconfigurability, the accelerator can dynamically adjust its dataflow to match the computational needs of different layers within the CNN, such as convolution, pooling, and fully connected layers. This adaptability not only enhances the overall efficiency and throughput of the accelerator but also facilitates rapid prototyping and experimentation with various CNN architectures [9].

In the architecture described by Figure 2.5, three distinct dataflows (DF1, DF2, and DF3) are employed to efficiently process DNN computations. DF1 facilitates parallelism in input and output channel dimensions, enhancing throughput for tasks with high data volume. DF2 extends this parallelism to feature map dimensions, particularly useful when dealing with limited output channels. DF3 targets depthwise convolution tasks by parallelizing kernel dimensions.

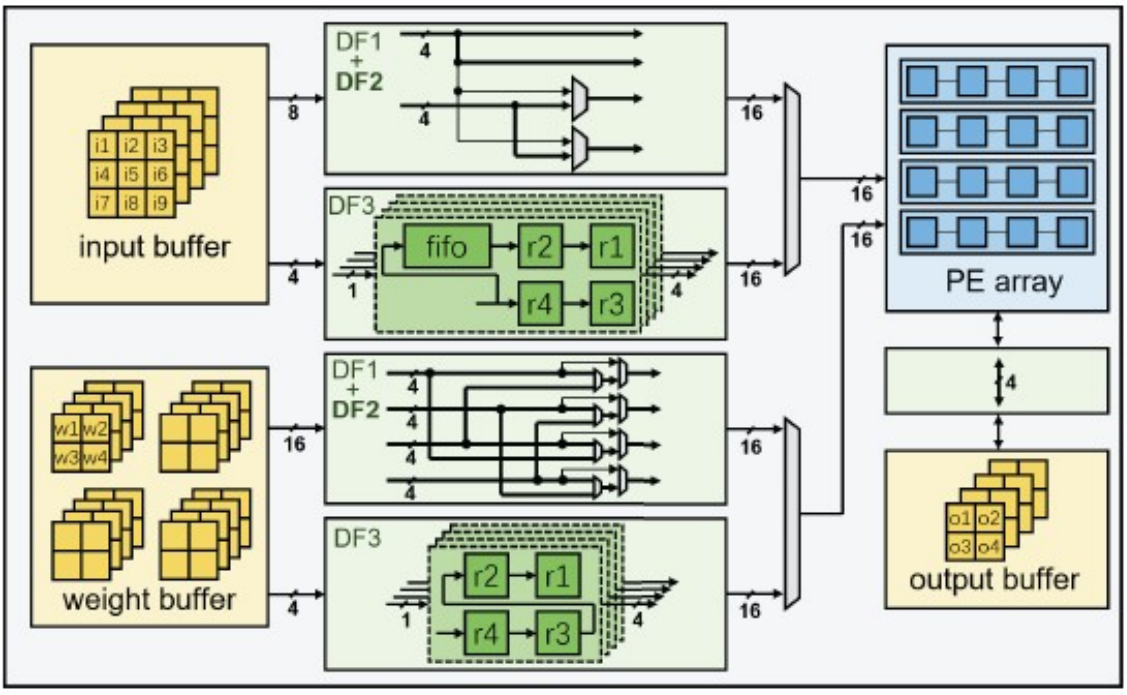


Figure 2.5: Block diagram of the accelerator with three dataflows (DF1-DF3) [9]

Adapt-Flow presents a novel approach to DNN accelerator architecture, emphasizing flexibility in dataflow implementation by offering a heterogeneous dataflow framework [10]. By accommodating varying dataflow patterns, such as input, weight, output, and row stationaries, Adapt-Flow optimizes resource utilization and energy efficiency. Its ability to seamlessly adapt to changing computational demands makes it a promising solution for accelerating deep learning workloads across a wide range of applications and hardware platforms.

The architecture of Adapt-Flow in Figure 2.6 contains a 128-PE setup, a global buffer connected to the PE array via a bi-directional Clos network. The Clos network includes diverged Clos switches and is further augmented with 2x4 and 2x2 Clos switches. Within the PE array, each PE is interconnected by a bi-directional ring at each row. Each PE features a 4x4 multiplier array for vector multiplication and a local buffer for temporary storage of weights, input activations, and output activations.

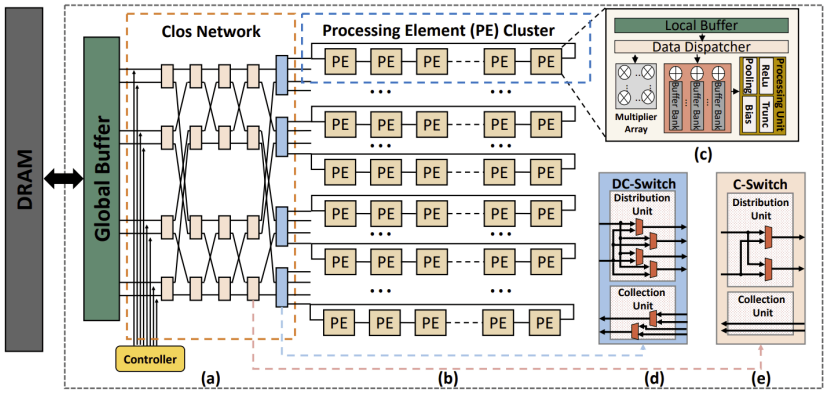


Figure 2.6: Overview of the proposed Adapt-Flow architecture [10]

BitSystolic [11] presents a cutting-edge NPU designed specifically for edge devices, offering exceptional energy efficiency and flexible dataflow configurations. The proposed BitSystolic architecture present in Figure 2.7 introduces a novel approach to neural network computation, enhancing both efficiency and performance. At its core lies a systolic array consisting of parallel systolic rows, each equipped with a chain of Bit-vector processing engines (BVPE). Three types of on-chip memory resources—input buffer, output buffer, and local register files—are integrated into the design to optimize data flow. Notably, the architecture organizes matrix operations by rows, with each systolic row performing vector-matrix multiplication. Within a systolic row,BVPEs execute partial vector products, ensuring efficient utilization of resources.

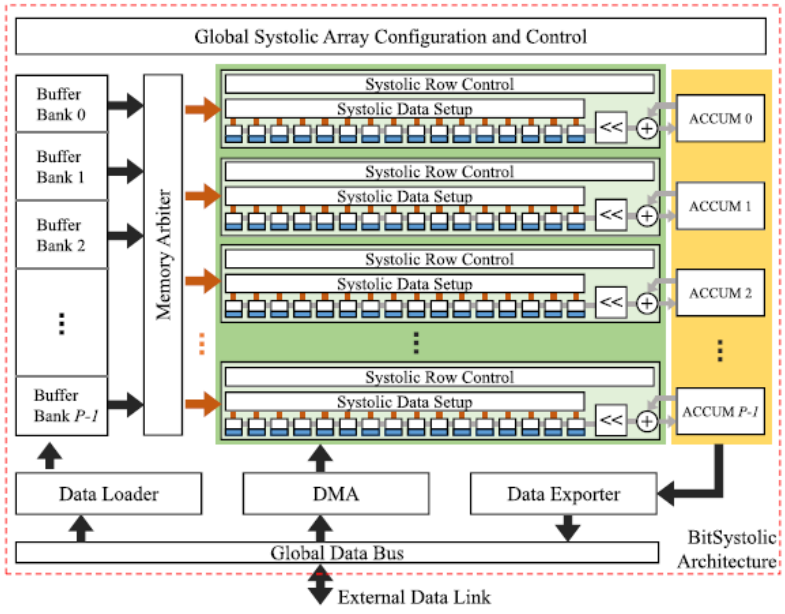


Figure 2.7: Bitsystolic Architecture [11]

3 Architecture Design - Neural Processing Unit

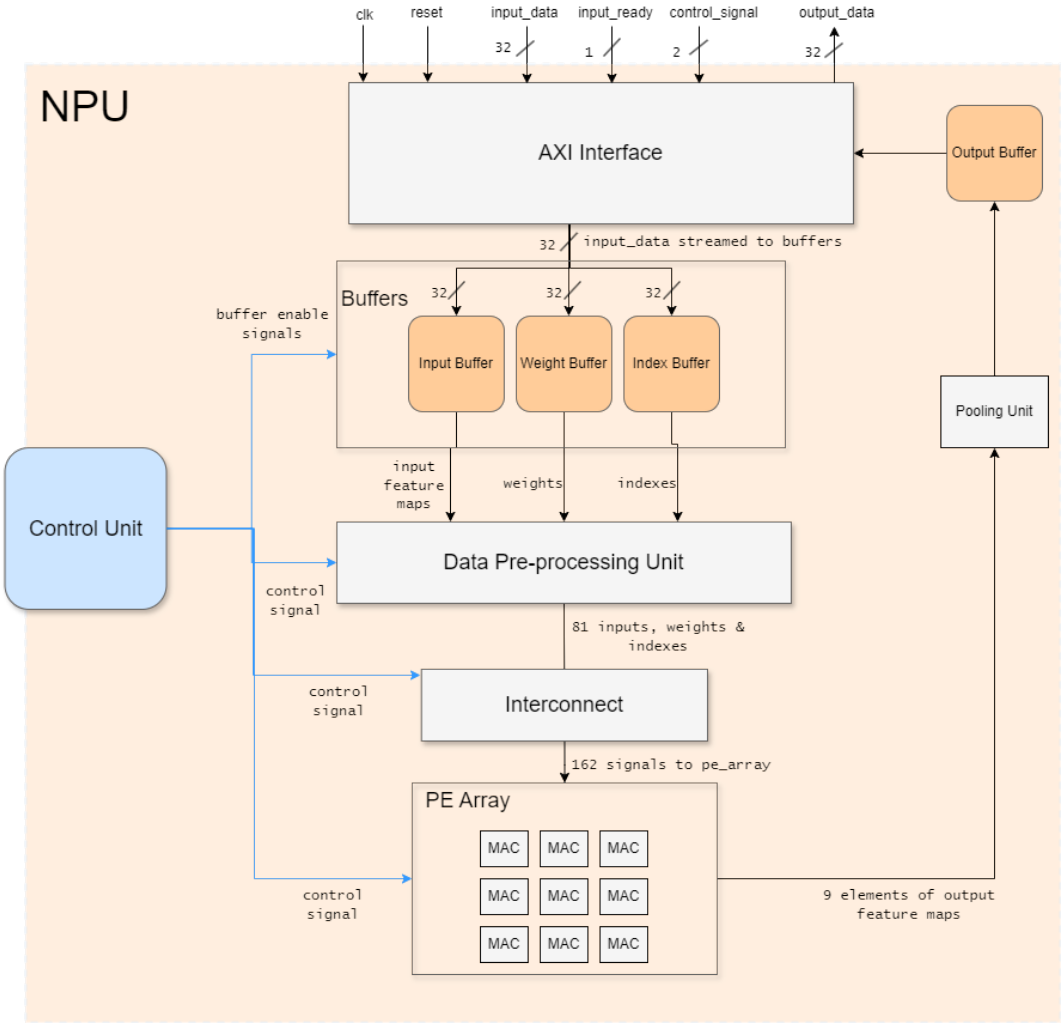


Figure 3.1: Overview of NPU Architecture

The NPU Architecture depicted in Figure 3.1 is designed with a primary focus on flexibility to support input and weight stationary dataflows, as well as to ensure compatibility for different sizes of convolutional layer parameters. It primarily consists of a PE array connected to a configurable interconnect module. Buffers are integrated for intermediate storage of

data, and preliminary processing is conducted by the data pre-processing unit before data transmission through the interconnect. An Advanced eXtensible Interface (AXI) interface is employed for data transfers with the user, while a control unit governs the operations of various modules.A Pooling unit is also provided to reduce spatial dimensions of the output feature maps.

3.1 AXI Interface

The AXI Interface represented in Figure 3.2 serves as a crucial pathway for data exchange between the user and the NPU, facilitating efficient streaming data transfers with minimal overhead. Within this interface, the TVALID signal denotes the validity of transmitted data, while TREADY indicates the receiver's readiness to accept new data. This streamlined handshake mechanism based on the AXI Lite Protocol ensures seamless data streaming between the AXI Interface and the user [12].

In the context of this research project, the AXI Interface serves two primary functions:

- It efficiently receives one data per clock cycle from the user and channels it to any of the three designated buffers (input, weight, or index buffer).
- Upon completion of processing, it streams elements of the output feature map from the output buffer back to the user, facilitating the retrieval of processed data.

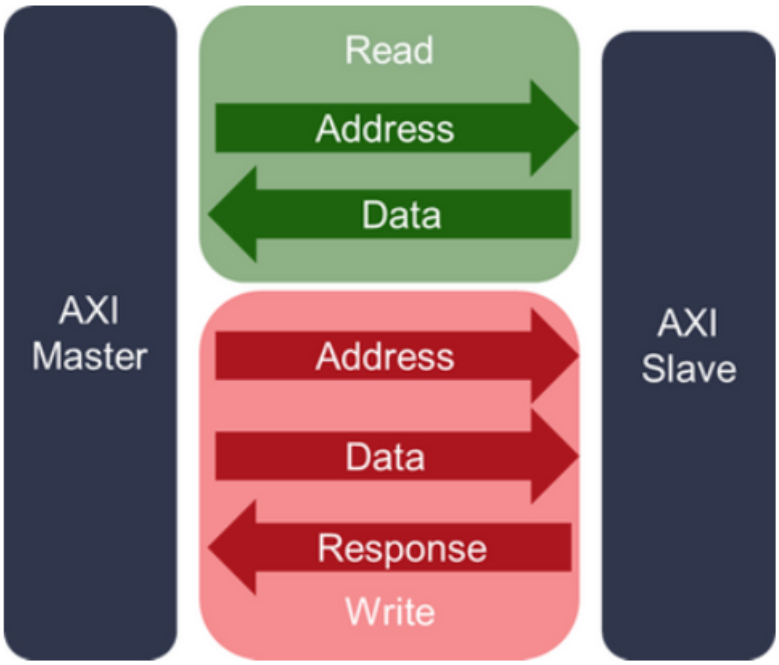


Figure 3.2: AXI Lite Interface

3.2 Buffers

Buffers are integral components of NPUs, serving as temporary storage units that facilitate efficient data flow management within the processing pipeline. By providing intermediate storage, buffers enable parallelism, latency hiding, and pipeline balancing, thereby enhancing the overall performance, flexibility, and throughput of NPUs.There are 4 buffers used for storage represented in Figure 3.3 which are all controlled by the control unit. The 4 buffers are:

- Input Buffer
- Weight Buffer
- Index Buffer
- Output Buffer

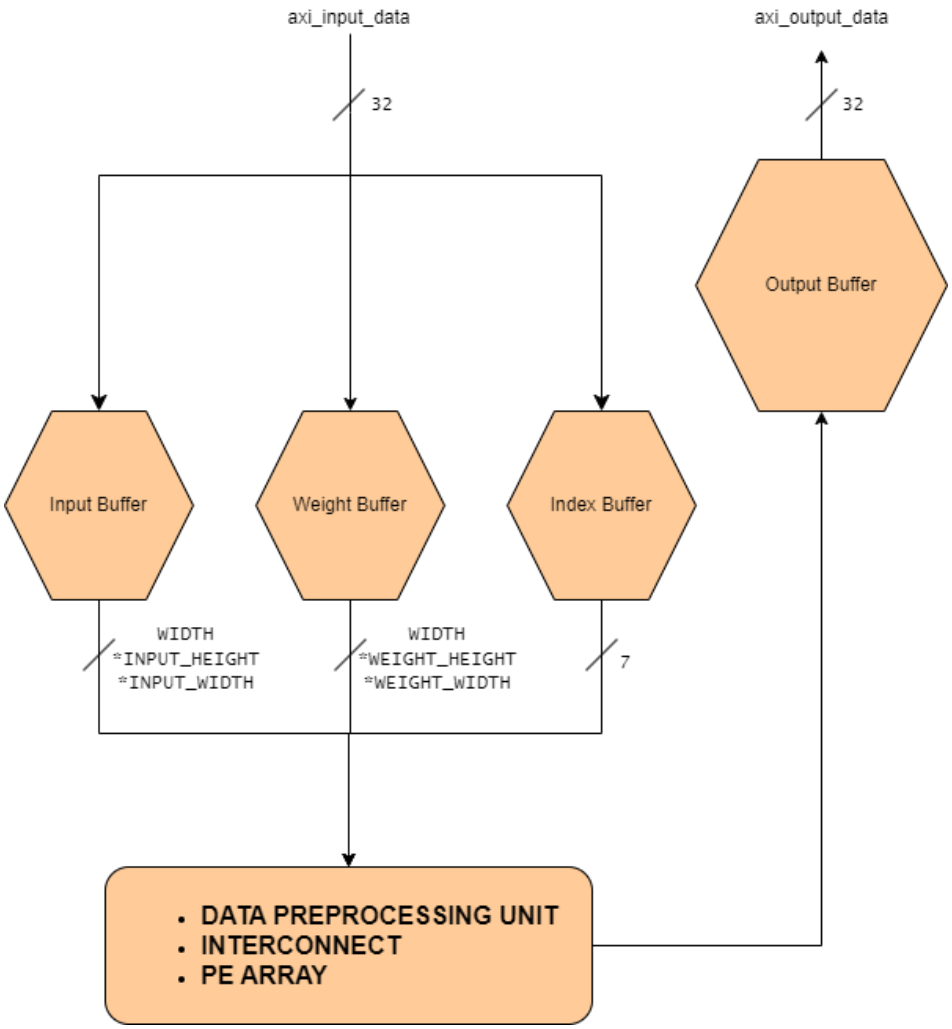


Figure 3.3: Buffers

3.2.1 Input, Weight and Index Buffer

In CNNs, the input feature map and weight parameters serve as foundational components essential for driving network operations. The input feature map embodies the raw data or input image supplied to the network, typically organized as pixel values within a grid-like structure. Conversely, the weight parameters, often referred to as filters or kernels, represent adaptable parameters that the network refines during training to discern significant features from the input data.

Initially, the input buffer accumulates elements of the input feature map, and upon reaching capacity, the control unit orchestrates the transition to filling the weight buffer. In the Multicast mode of operation, users can designate the indexes of the MAC unit to which input data should be directed—a task facilitated by the index buffer. Subsequently, processing commences once all three buffers are saturated, and data is dispatched from the buffers to the data pre-processing unit.

3.2.2 Output Buffer

In the context of the research project, the output buffer assumes the critical role of storing the conclusive results—namely, the elements constituting the output feature map subsequent to the convolution process. It accepts data from the PE array, managing the transmission of information to the AXI interface, ultimately facilitating the seamless streaming of results to the end user. Given the parameters in Table 3.1, the equations of output feature maps are

Table 3.1: 2D Convolution Parameters

Parameter	Description
N_{out}	Total Number of Output Elements
H_{out}	Height of Output feature map
W_{out}	Width of Output feature map
H_{in}	Height of Input feature map
W_{in}	Width of Input feature map
K	Kernel Size
P	Padding size
S	Stride size

$$N\text{out} = (H_{\text{out}}) \cdot (W_{\text{out}}) \tag{3.1}$$

$$H_{\text{out}} = \left(\frac{H_{\text{in}} - K + 2P}{S} + 1 \right) \tag{3.2}$$

$$W_{\text{out}} = \left(\frac{W_{\text{in}} - K + 2P}{S} + 1 \right) \tag{3.3}$$

where N,H_{out} and W_{out} are the size, height and width of the output feature maps, respectively.

3.3 Data Pre-processing Unit

The idea behind the data pre-processing unit is to simplify the processing at the PE array by modifying the data before it is sent to the interconnect and the subsequent PE array module.It has three main functions:

- Modifying Input Matrix for Zero padding and different Stride lengths
- Mapping Convolution onto Toeplitz matrix
- Batch division

3.3.1 Modifying Input Matrix for Zero Padding and Different Stride Lengths

The data pre-processing unit first adjusts the dimensions of the input data matrix to accommodate zero-padding (increase in dimensions) and varying stride lengths (decrease in dimensions) during the convolution operation.

Zero padding involves adding extra rows and columns of zeros around the borders of the input data matrix as shown in Figure 3.4. By padding the input with zeros, the output feature map can maintain the same spatial dimensions as the input, even after applying convolution operations with filter kernels.Zero padding is crucial for preserving spatial information and preventing the reduction of feature map size, which can otherwise lead to information loss and boundary effects.

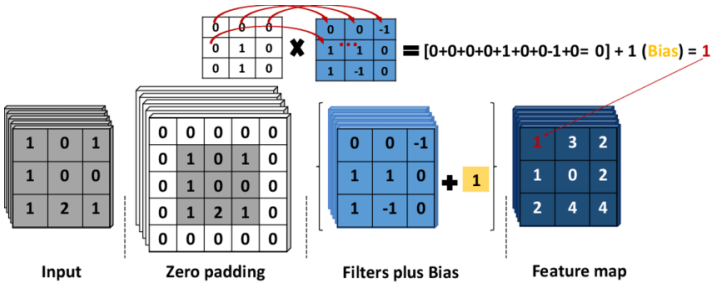


Figure 3.4: Illustration of Zero padding [13]

There are two kinds of zero padding supported by the design represented in figure 3.5 known as full padding and same padding.

- Same Padding: In same padding, the input image or feature map is padded with zeros in such a way that the output feature map has the same spatial dimensions as the input.
- Full Padding: With full padding, the filter is applied only to positions in such a way that all pixels are visited equal number of times by the filter thus resulting in an output feature map with increased spatial dimensions.

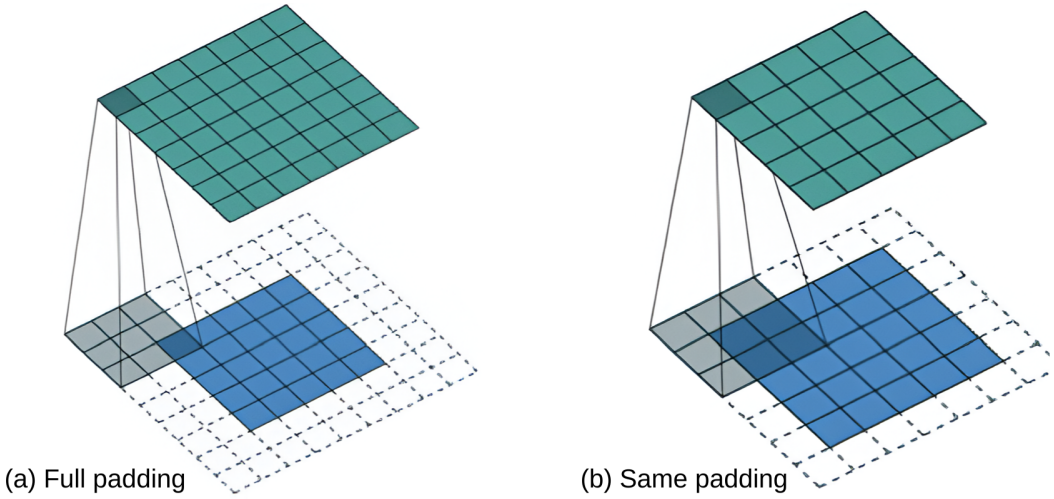


Figure 3.5: Full and Same padding [14]

Strides, as illustrated in Figure 3.6, within neural networks dictate the increment size utilized when sliding a filter (or kernel) across the input data during convolution operations. They play a crucial role in determining the displacement of the filter both horizontally and vertically at each step, thereby influencing factors such as receptive field size, computational efficiency, and feature resolution. Larger stride values correspond to a reduction in the number of output pixels or feature maps, leading to spatial down-sampling. Conversely, smaller stride values yield more output pixels and may retain a greater amount of spatial information. This architecture is designed to accommodate stride lengths ranging from 1 to 4. This flexibility empowers the network to adapt to varying requirements and optimize performance based on specific convolutional layer parameters.

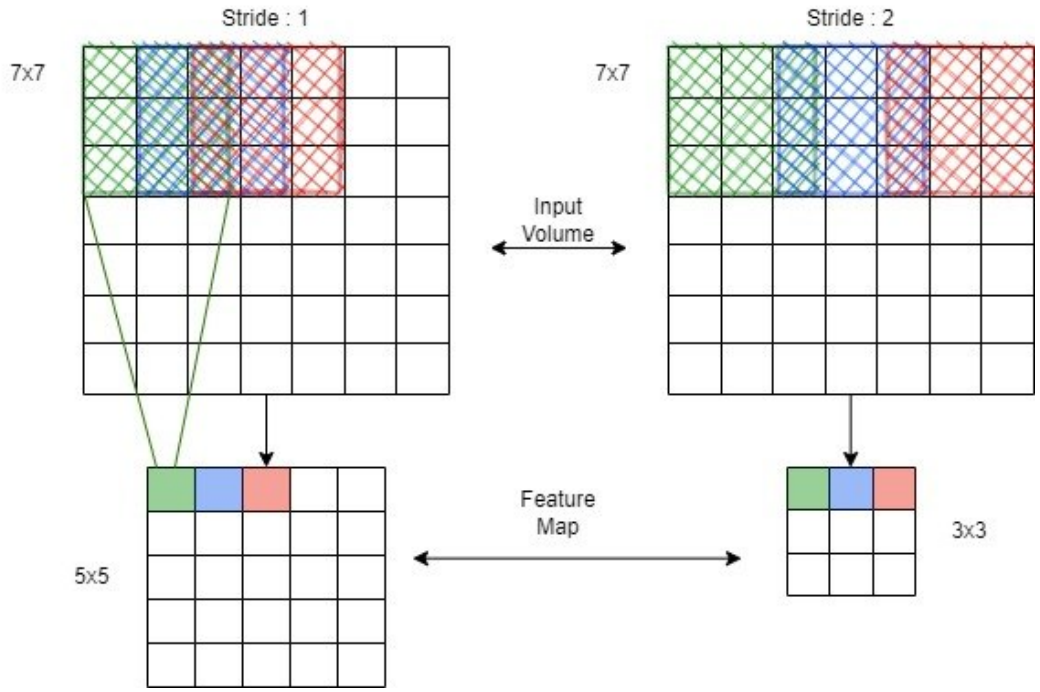
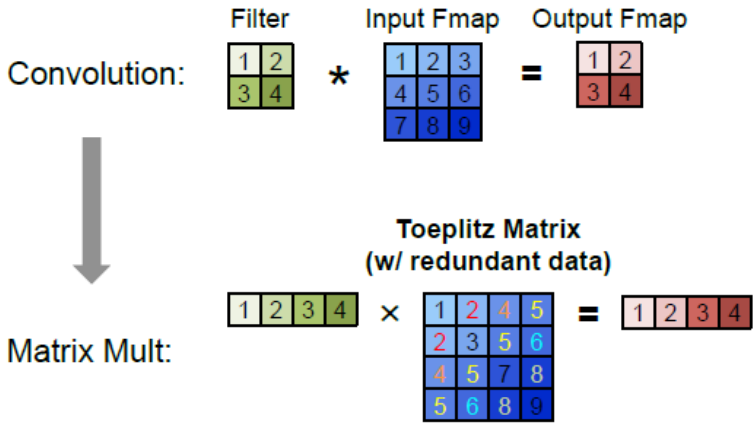


Figure 3.6: Stride =1 and Stride =2 [15]

3.3.2 Mapping Convolution onto Toeplitz Matrix

A Toeplitz matrix represents a structured matrix where each descending diagonal from left to right contains constant values. In convolutional operations, Toeplitz matrices are utilized to efficiently perform matrix-vector multiplications, facilitating the transformation of convolutional filters into structured matrices, thus optimizing the computational process.When convolutional filters are transformed into Toeplitz matrices convolution operations can be expressed as simple matrix multiplications. This transformation represented in Figure 3.7 simplifies the computational complexity of convolution, enabling faster and more efficient processing across various hardware architectures, including GPUs, Tensor Processing Unit (TPU)s etc.



(a) Mapping convolution to Toeplitz matrix

Figure 3.7: Mapping Convolution onto Toeplitz Matrix [4]

Furthermore, this approach enables the exploitation of parallelism and data locality, which are essential for efficient utilization of hardware resources and memory bandwidth.The input feature maps are converted into a toeplitz matrix and is then further divided into several batches.The size of the toeplitz matrix is given by the equation

$$N_{\text{toe}} = (H_{in} - 2) \cdot (W_{in} - 2) \cdot K^2 \tag{3.4}$$

where N_{toe} is the Total number of elements in the Toeplitz matrix,Hin is Input Feature Map Height, Win is Input Feature Map Width and K is the Kernel Size.

Overall, mapping convolution to matrix multiplication using Toeplitz matrices represents a cornerstone optimization technique in deep learning, enabling the efficient execution of convolutional neural networks across a wide range of hardware platforms and accelerating the pace of innovation in artificial intelligence research and applications.

3.3.3 Batch Division

Since the PE array has a size of 9x9, it can only process 81 inputs and weights simultaneously. Therefore, for input feature map sizes larger than this, it becomes necessary to divide the input into batches. Specifically, the Toeplitz matrix formed from the input feature maps is divided into batches and sequentially sent to further processing modules. The parameters for batch division are detailed in Table 3.2, and the corresponding equations governing this division are provided below:

$$N_{\text{toe}} = (H_{\text{in}} - 2) \cdot (W_{\text{in}} - 2) \cdot K^2 \tag{3.5}$$

$$B = \text{ceil}\,(N_{\text{toe}}/81) \tag{3.6}$$

Table 3.2: Batch Division Parameters

Parameter	Description
N_{toe}	Number of Elements in Toeplitz matrix
H_{in}	Height of Input feature map
W_{in}	Width of Input feature map
K	Kernel Size
B	Batch Size

3.4 Interconnect

The configurable interconnect illustrated in Figure 3.8 plays a pivotal role in facilitating the versatility of the Neural Processing Unit (NPU), enabling it to accommodate two distinct dataflows : input and weight stationary.

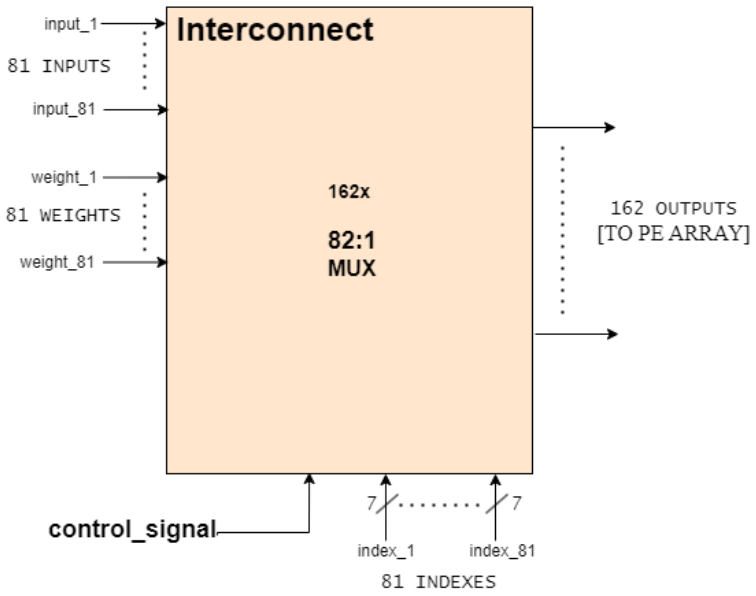


Figure 3.8: Interconnect

This interconnect structure comprises an array of 82:1 multiplexers, totaling 162 multiplexers. Within this array, 81 multiplexers channel input values to the PE array, while the remaining 81 multiplexers route weight values to the same PE array.

Each multiplexer's functionality within the interconnect is governed by a control signal. Depending on the state of this control signal, a multiplexer in the interconnect may establish a connection between an element of the input or weight matrix and a designated MAC unit within the PE array.The interconnect module also accommodates 81 7-bit index signals, which play a crucial role in facilitating various communication modes. The interconnect represented in Figure 3.9 also supports three different communication modes namely :-

- **Unicast** :In a unicast communication, a data packet is sent from one sender to one specific MAC unit. It's a one-to-one communication model where each packet has a single destination MAC unit. The destination MAC units are decided by the user given index inputs.
- **Multicast** : In a multicast communication, a data packet is sent from one sender to one specific row of MAC units. It's a one-to-many communication model where each packet has a specific row of MAC units. The destination rows are decided by the user given index inputs.
- **Broadcast** :In a broadcast communication, a data packet is sent from one sender to all of the MAC units. It's a one-to-all communication model where one packet is sent to all the MAC units.

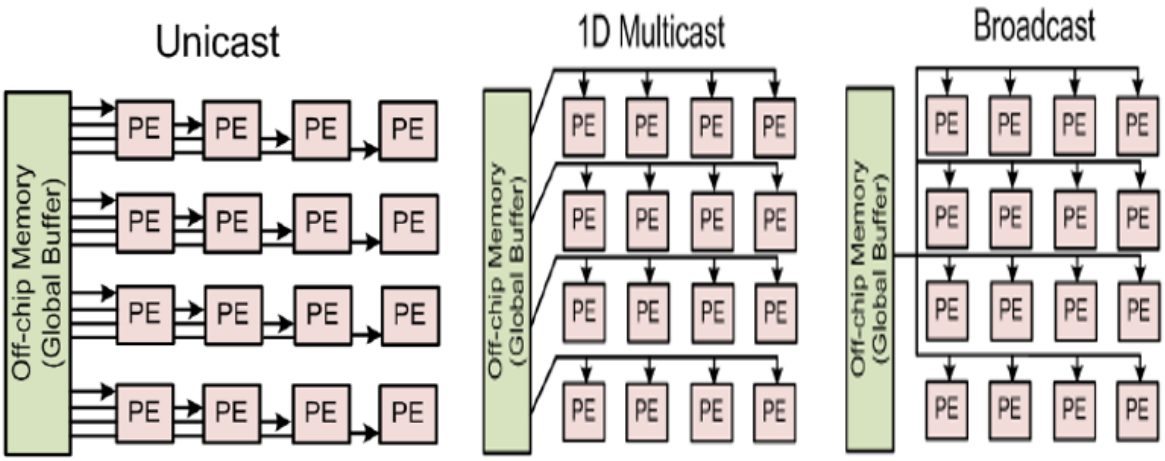


Figure 3.9: Communication Modes [4]

3.5 PE Array

The PE array plays a pivotal role in the convolutional operation within the neural processing unit (NPU). It receives inputs and weights corresponding to the selected dataflow (input or weight) and communication mode (multicast or unicast) from the interconnect module and performs convolution operations on the received data. Comprising 81 MAC units arranged in a 9x9 format, the PE array leverages the concept of a 'convolution window,' where each row of 9 MAC units represents a specific section of the input data over which the kernel slides during the convolution operation. Since the Kernel Size that the architecture supports is 3*3, the PE array was arranged to consist of 9 rows, producing 9 elements of the output feature map when provided with one batch of inputs and weights.

The processing of each element of the output feature map requires 17 clock cycles. This duration accounts for the fact that each MAC operation within the PE array involves two clock cycles for multiplication and accumulation, and there are 9 such operations per row. Additionally, the latency is reduced by one clock cycle since the first MAC unit takes 0 as its previous partial sum to be added.

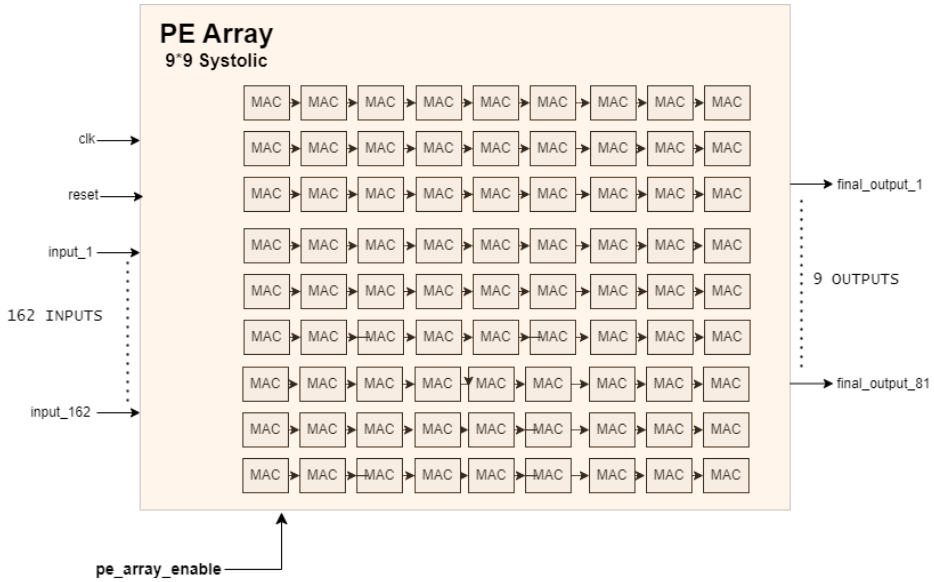


Figure 3.10: PE Array

Furthermore, the PE array is based on the systolic array architecture as shown in Figure 3.10. In a systolic array, processing elements are systematically arranged in a regular grid, establishing a pipeline through which data flows in a synchronized manner. Each processing element executes a predefined computation on the incoming data and propagates the result to neighbouring elements, facilitating high throughput and low-latency execution. This architectural design makes systolic arrays particularly suitable for tasks such as matrix multiplication, convolutional operations in neural networks, and various signal processing algorithms.

3.5.1 MAC unit

The PE array comprises 81 MAC units, each responsible for performing multiplication of input and weight values, followed by the addition of the previous partial sum to this product, resulting in the generation of a new partial sum. The basic architecture of a MAC unit is depicted in Figure figure 3.11.

As data streams through the 9 MAC units present in a row of the PE array, each MAC unit contributes to the formation of the final output. The output of the final MAC unit in the row represents one element of the output feature map. The MAC unit is implemented with the usage of Digital Signal Processing (DSP) slices present in the FPGA with support for maximum datawidth of 18 bits.The use of DSP slices also results in lower resource utilization and reduced power consumption of 0.683 W.

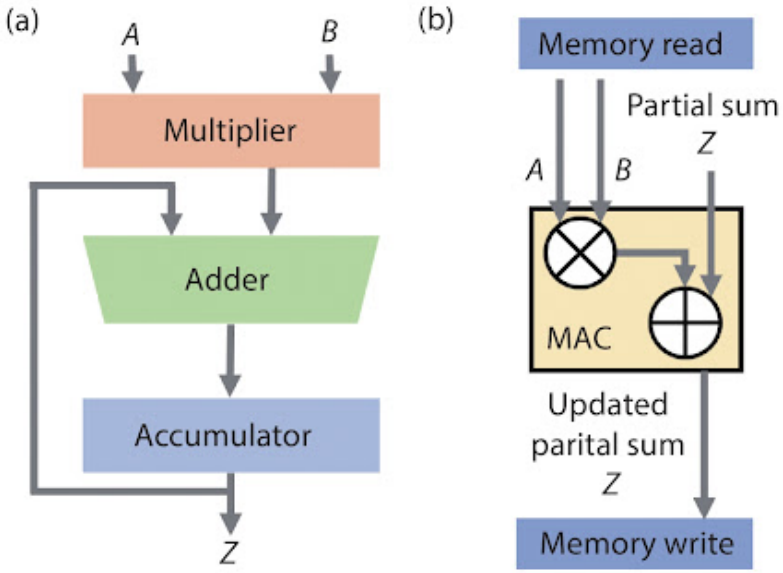


Figure 3.11: (a) Block diagram of the basic MAC unit. (b) Memory read and write for each MAC unit [16]

3.6 Control Unit

The control unit shown in Figure 3.12 acts as the central coordinator, facilitating communication and synchronization between various modules.The control unit for the NPU operates as a Moore Finite State Machine (FSM) and comprises two states: IDLE and PROCESSING. The control unit transitions between these states based on certain conditions, primarily driven by input signals such as buffer readiness and control signals from the user. Upon toggling to the PROCESSING state, the control unit activates other crucial components of the NPU, including the data pre-processing unit and the PE array, to commence processing tasks.

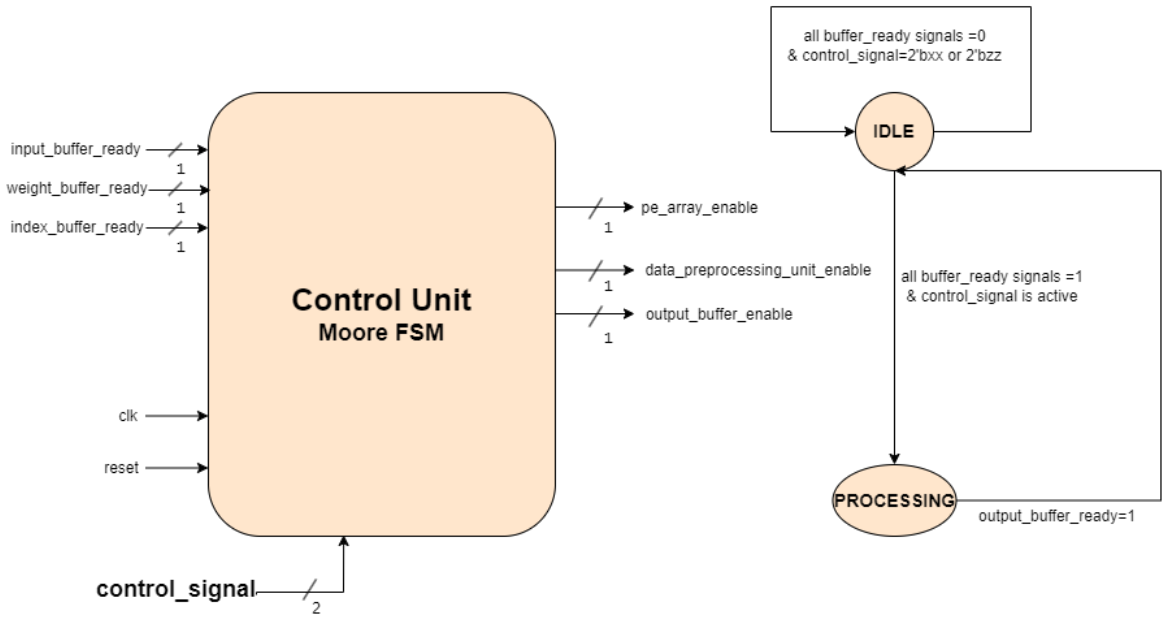


Figure 3.12: Control Unit and State Transition Diagram

Additionally, the code implements logic to manage the output buffer enable signal, facilitating the efficient flow of processed data. The control signal provided by the user determines the type of dataflow and the communication mode, with its encoding specified in Table 3.3. Additionally, the system undergoes a reset procedure at the outset to initialize the registers.

Table 3.3: Control Signal Encoding

Parameter	Description
00	Weight Stationary with Unicast
01	Weight Stationary with Multicast
10	Input Stationary with Unicast
11	Input Stationary with Multicast

State Transition Info is represented in Table 3.4 :

- IDLE State (0):

  When in the IDLE state, if the control signal is active, and all input buffers are ready, the FSM transitions to the PROCESSING state.

  Otherwise, it remains in the IDLE state.

  The PE Array Enable and Data Preprocessing Unit Enable signals are activated when transitioning to the PROCESSING state.

- PROCESSING State (1):

  In the PROCESSING state, the FSM waits till the processing is finished which is indicated by the output buffer filling up and streaming out all the outputs.

  After this, the FSM transitions back to the IDLE state.

Table 3.4: FSM Truth Table

State	Ctrl Signal	Input,Weight and Index Buffer Ready	Next State	Output Buffer Full
IDLE (0)	inactive	0	IDLE (0)	0
IDLE (0)	active	1	PROCESSING (1)	0
PROCESSING (1)	active	0	IDLE (0)	1

3.7 Pooling Unit

This architecture supports two pooling operations- max pooling and average pooling shown in figure 3.13 which are fundamental components of CNNs, essential for reducing the spatial dimensions of feature maps while extracting salient features. Max pooling selects the maximum value within a predefined window of 2*2, while average pooling computes the average value in this same window. These operations serve to reduce computational complexity and memory requirements, enabling efficient processing of input volumes. Furthermore, pooling introduces translation invariance by selecting dominant features within local regions, promoting robustness to spatial variations in the input data.

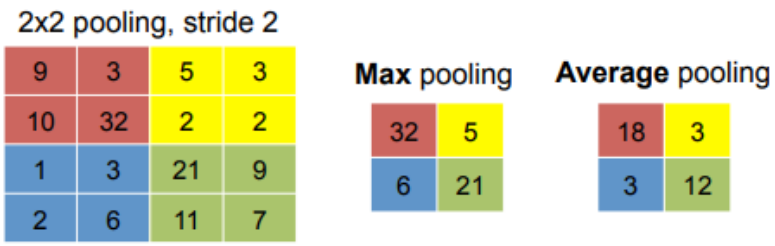


Figure 3.13: Max and Average Pooling [4]

4 Results and Evaluation

This chapter contains the evaluation of the NPU architecture and a discussion of the results.The target device for synthesis is a Xilinx Ultrascale FPGA on the ZCU104 evaluation Board.The evaluation platform used is the Xilinx Vivado Suite using Verilog .

4.1 Design Features

This NPU was designed with high flexibilty as the end goal and in the below given Table 4.1 are the ranges of the convolution layer parameters that can be used on this architecture.

Table 4.1: Convolution Layer Parameters

Convolution Layer Parameters	Maximum Range
Kernel Size (bits*bits)	3*3
Data Width	upto 18 bits
Input feature Map size (bits*bits)	from 5*5 to 256*256(non-square matrix also supported)
No of dataflows	2 (Input and weight stationary)
No of communication modes	3 (Broadcast,Multicast and Unicast)
Stride	1,2,3,4
Padding	2 types (Full and same padding)
Pooling	2 types (Max and average pooling)

4.2 Clock Summary

From the below Table 4.2, The minimum clock period observed without slack violation was obtained at 7.1 nsec. Hence the maximum clock frequency obtained for this NPU architecture is around 140 MHz.

Table 4.2: Clock and Maximum frequency for Proposed NPU

Clock Period(ns)	Frequency (MHz)
7.1	140

4.3 Resource Utilization

From the utilization report shown in Figure 4.1, We can observe that 81 DSPs are used for implementing the MAC units belonging to the PE array.

Resource	Utilization	Available	Utilization %
LUT	92947	230400	40.34
FF	8108	460800	1.76
DSP	81	1728	4.69
IO	48	360	13.33
BUFG	1	544	0.18

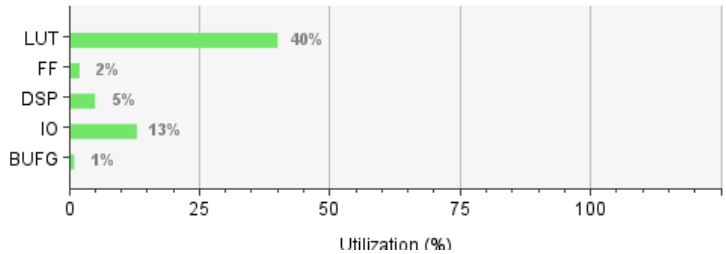


Figure 4.1: Resource Utilization Report Summary

4.4 Power Consumption

The Total On-chip power consumption is 0.683 W.

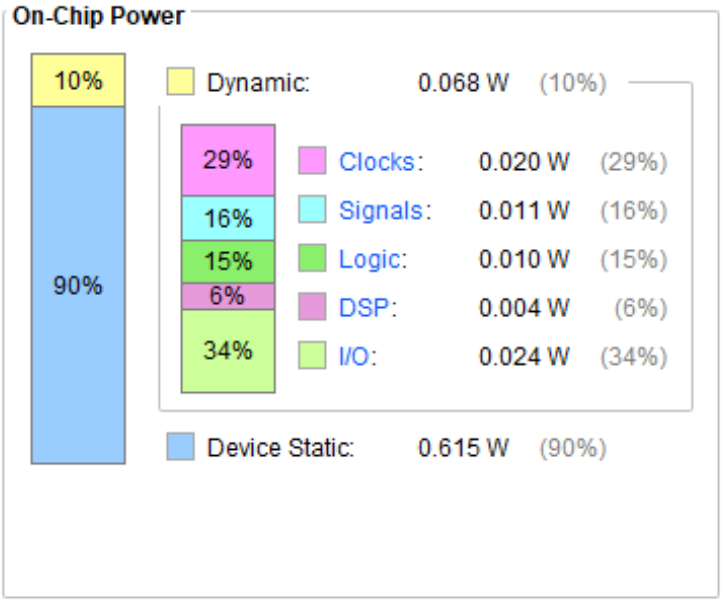


Figure 4.2: Power Analysis Report Summary

4.5 Simulation

The design of the NPU is configurable by either changing the parameters present in the module port declaration or by changing the control signal.The below given Table 4.3 categorizes how to change any one of the convolutional layer parameters.

Table 4.3: Method of Configuration

Convolution Layers Parameter	Method of Configuration
Kernel Size Data Width (bits) Input feature Map size (bits*bits) Weights Size (bits*bits) Stride Padding Pooling	Parameter Declaration
Dataflows Communication modes	Control Signal

4.5.1 Testcase 1 = Input Stationary with Zero padding

A Simulation performed on an Input feature map of size =7*7 represented on Figure 4.4 with **control signal = 10 (Input stationary with Unicast)** finished with an execution time of 130 µs.

The parameters used for Testcase 1 is given in the table Table 4.4 .

Parameter	Value
DATA WIDTH	4
INPUT_WIDTH	7
INPUT_HEIGHT	7
KERNEL_SIZE	3
WEIGHT_HEIGHT	21
WEIGHT_WIDTH	21
STRIDE	1
PADDING	1

Table 4.4: Parameters for Testcase 1

```
Displaying Input Feature Maps:
 0   0   0   0   0   0   0   0   0
 0   0   1   2   3   4   5   6   0
 0   7   8   4   5  10   3   4   0
 0   9  13   7   8  14   1   2   0
 0   3   8   5   6  10   8   8   0
 0   7   6   5   4   3   2   1   0
 0   0   4   6   4   4   9   4   0
 0   4  10  11   5   5   5   5   0
 0   0   0   0   0   0   0   0   0
```

(a) Input feature Map (7*7) with Zero padding

```
Displaying Weights Matrix:
 0   1   1   1   1   1   1   1   1
 2   2   2   2   2   2   2   2   2
 3   3   3   3   3   3   3   3   3
 4   4   4   4   4   4   4   4   4
 5   5   5   5   5   5   5   5   5
 6   6   6   6   6   6   6   6   6
 7   7   7   7   7   7   7   7   7
 8   8   8   8   8   8   8   8   8
 9   9   9   9   9   9   9   9   9
 2   2   2   2   2   2   2   2   2
 2   2   2   2   2   2   2   2   2
 3   3   3   3   3   3   3   3   3
 4   4   4   4   4   4   4   4   4
 5   5   5   5   5   5   5   5   5
 6   6   6   6   6   6   6   6   6
 7   7   7   7   7   7   7   7   7
 8   8   8   8   8   8   8   8   8
 9   9   9   9   9   9   9   9   9
 3   3   3   3   3   3   3   3   3
```

(b) First 21 Weights used (Total = 441)

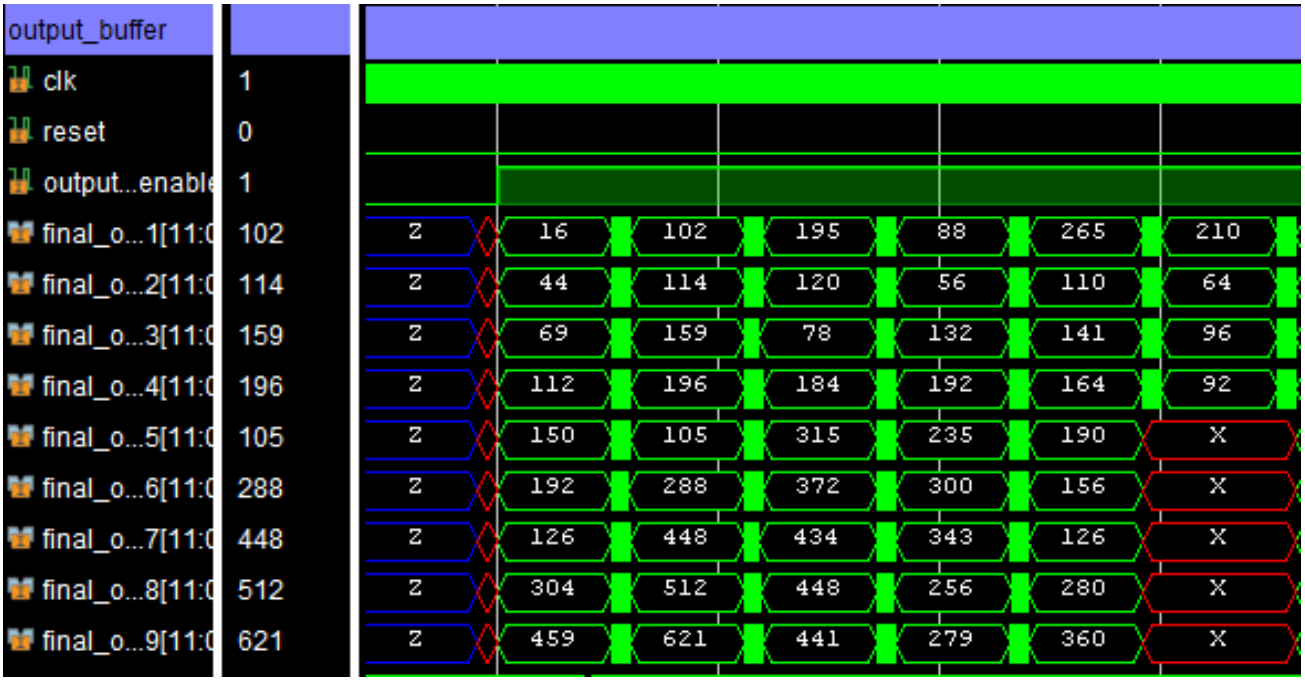


Figure 4.4: Final Output feature Map (7*7)

4.5.2 Testcase 2 = Weight Stationary with Stride = 2

A Simulation performed on an Input feature map of size =32*32 represented on Figure 4.5 with **control signal = 00 (Weight stationary with Unicast)** finished with an execution time of 410 µs.

The parameters used for Testcase 2 is given in the table Table 4.5 .

Parameter	Value
DATA WIDTH	8
INPUT_WIDTH	32
INPUT_HEIGHT	32
KERNEL_SIZE	3
WEIGHT_HEIGHT	3
WEIGHT_WIDTH	3
STRIDE	2
PADDING	0

Table 4.5: Parameters for Testcase 2

```
Displaying Input Feature Maps:
  0   1   2   7  12   5   6   7  21  12   1   2   3  13   5   6   7  56   2   1   2  13   4   5   6  72   8   8   7  10   5   4
  3   5   1   0   4  34  12   4   3   4   2   4   0   5   7   5   5   2   5  34   5  87   6   6  90   6   6   9   6  10   6   7
  4   7   7   6   7  10   7   7   8   6   8   8   4   5  34   8   8   0   1   2   3   4   5   6   7   8   0   1   2   3   4   5
  6   7   8   0   1   2   3   4   5   6   7   8   8   7   6   5   4   3   2   1   0   4   4   4   4   4   4   4   4   4   5   5
  5   5   5   5   5   5   5   6   6   6   6   6   6   6   6   6   7   7   7   7   7   7   7   7   8   8   8   8   8   8   8
  8   8   0   1   2   3   4   5   6   7   8   0   1   2   3   4   5   6   7   8   0   1   2   3   4   5   6   7   8   8   7   6
  5   4   3   2   1   0   4   4   4   4   4   4   4   4   4   5   5   5   5   5   5   5   5   5   6   6   6   6   6   6   6   6
  6   7   7   7   7   7   7   7   7   7   8   8   8   8   8   8   8   8   8   0   1   2   3   4   5   6   7   8   0   1   2   3
  4   5   6   7   8   0   1   2   3   4   5   6   7   8   8   7   6   5   4   3   2   1   0   4   4   4   4   4   4   4   4   4
  5   5   5   5   5   5   5   5   5   6   6   6   6   6   6   6   6   6   7   7   7   7   7   7   7   7   8   8   8   8   8   8
  8   8   8   8   0   1   2   3   4   5   6   7   8   0   1   2   3   4   5   6   7   8   0   1   2   3   4   5   6   7   8   8
  7   6   5   4   3   2   1   0   4   4   4   4   4   4   4   4   4   5   5   5   5   5   5   5   5   5   6   6   6   6   6   6
  6   6   6   7   7   7   7   7   7   7   7   7   8   8   8   8   8   8   8   8   0   1   2   3   4   5   6   7   8   0   1
  2   3   4   5   6   7   8   0   1   2   3   4   5   6   7   8   8   7   6   5   4   3   2   1   0   4   4   4   4   4   4   4
  4   4   5   5   5   5   5   5   5   5   5   6   6   6   6   6   6   6   6   6   7   7   7   7   7   7   7   7   8   8   8
  8   8   8   8   8   8   0   1   2   3   4   5   6   7   8   0   1   2   3   4   5   6   7   8   0   1   2   3   4   5   6   7
  8   8   7   6   5   4   3   2   1   0   4   4   4   4   4   4   4   4   4   5   5   5   5   5   5   5   5   5   6   6   6   6
  6   6   6   6   6   7   7   7   7   7   7   7   7   8   8   8   8   8   8   8   8   0   1   2   3   4   5   6   7   8
  0   1   2   3   4   5   6   7   8   0   1   2   3   4   5   6   7   8   8   7   6   5   4   3   2   1   0   4   4   4   4   4
  4   4   4   4   5   5   5   5   5   5   5   5   5   6   6   6   6   6   6   6   6   6   7   7   7   7   7   7   7   7   8
  8   8   8   8   8   8   8   8   0   1   2   3   4   5   6   7   8   0   1   2   3   4   5   6   7   8   0   1   2   3   4   5
  6   7   8   8   7   6   5   4   3   2   1   0   4   4   4   4   4   4   4   4   5   5   5   5   5   5   5   5   5   6   6
  6   6   6   6   6   6   6   7   7   7   7   7   7   7   7   7   8   8   8   8   8   8   8   8   8   0   1   2   3   4   5   6
  7   8   0   1   2   3   4   5   6   7   8   0   1   2   3   4   5   6   7   8   8   7   6   5   4   3   2   1   0   4   4   4
  4   4   4   4   4   4   5   5   5   5   5   5   5   5   5   5   6   6   6   6   6   6   6   6   6   7   7   7   7   7   7   7
  7   8   8   8   8   8   8   8   8   8   0   1   2   3   4   5   6   7   8   0   1   2   3   4   5   6   7   8   0   1   2   3
  4   5   6   7   8   8   7   6   5   4   3   2   1   0   4   4   4   4   4   4   4   4   5   5   5   5   5   5   5   5   5   5
  6   6   6   6   6   6   6   6   6   7   7   7   7   7   7   7   7   8   8   8   8   8   8   8   8   8   0   1   2   3   4   5
  5   6   7   8   0   1   2   3   4   5   6   7   8   0   1   2   3   4   5   6   7   8   8   7   6   5   4   3   2   1   0   4
  4   4   4   4   4   4   4   4   5   5   5   5   5   5   5   5   5   6   6   6   6   6   6   6   6   6   7   7   7   7   7   7
  7   7   7   8   8   8   8   8   8   8   8   8   0   1   2   3   4   5   6   7   8   0   1   2   3   4   5   6   7   8   0   1
  2   3   4   5   6   7   8   8   7   6   5   4   3   2   1   0   4   4   4   4   4   4   4   4   4   5   5   5   5   5   5   5
Displaying Weights Matrix:
  0   1   2   3   4   5   6   7   8
```

Figure 4.5: Input feature Map (32*32) and Weights (3*3)

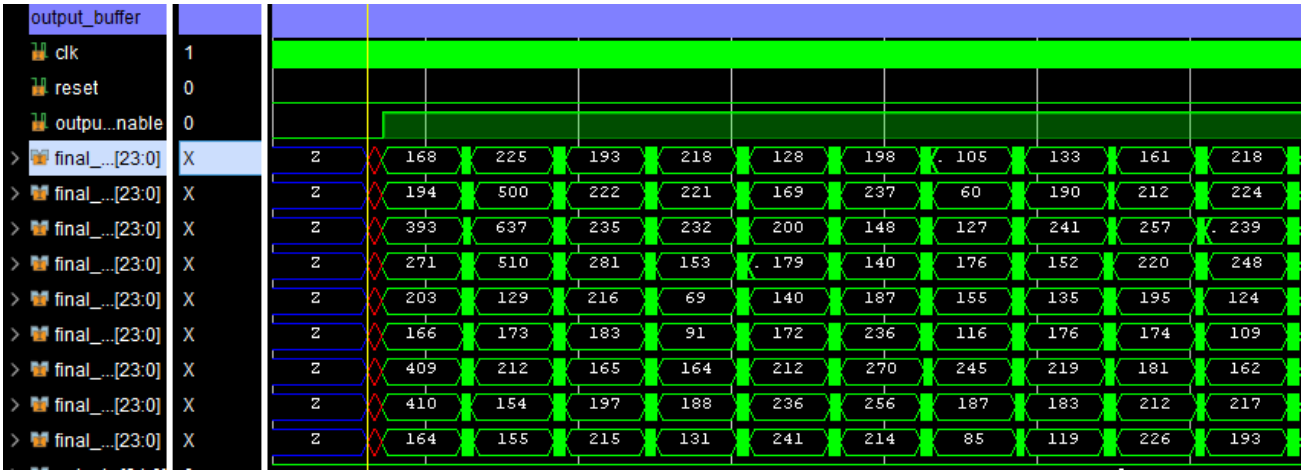


Figure 4.6: Final Output feature Map (15*15)

4.5.3 Testcase 3 = Weight Stationary for Input Size =1024*2

A Simulation performed on an Input feature map of size =1024*2 represented on Figure 4.8 with **control signal = 00 (Weight stationary with Unicast)** finished with an execution time of 2940 µs.

The parameters used for Testcase 3 is given in the table Table 4.6 .

Parameter	Value
DATA WIDTH	8
INPUT_WIDTH	2
INPUT_HEIGHT	1024
KERNEL_SIZE	3
WEIGHT_HEIGHT	3
WEIGHT_WIDTH	3
STRIDE	1
PADDING	1

Table 4.6: Parameters for Testcase 3

```
Displaying Input Feature Maps:
   0    0    0    0
   0    0    1    0
   0    2    7    0
   0   12    5    0
   0    6    7    0
   0   21   12    0
   0    1    2    0
   0    3   13    0
   0    5    6    0
   0    7   56    0
   0    2    1    0
   0    2   13    0
   0    4    5    0
   0    6   72    0
   0    8    8    0
   0    7   10    0
   0    5    4    0
   0    3    5    0
   0    1    0    0
   0    4   34    0
   0   12    4    0
```

(a) First 21 elements of Input feature Map (1024*2) with Zero padding

```
Displaying Weights Matrix:
   0    1    2    3    4    5    6    7    8
```

(b) Weights(3*3)

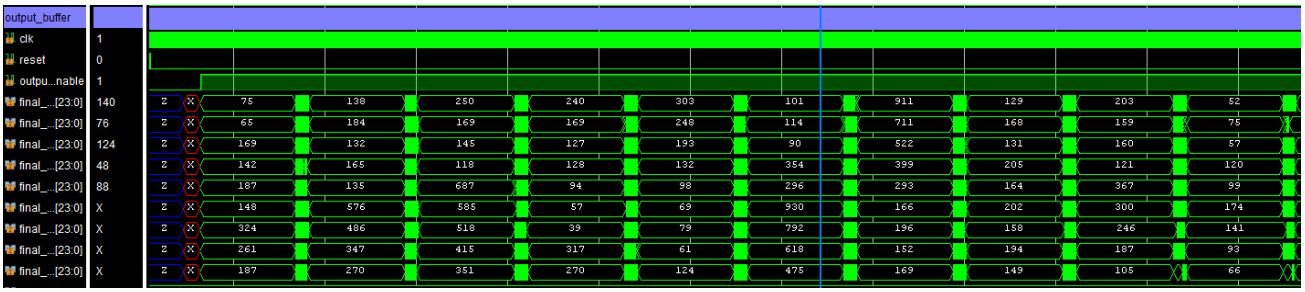


Figure 4.8: Final Output feature Map (1024*2)

5 Conclusion and Future Scope

5.1 Conclusion

This research project is based on the implementation and assessment of a versatile NPU, designed to support input- and weight-stationary dataflow models. Different layers within a larger neural network may benefit from different dataflows based on the nature of the computations involved. The flexibility of the NPU architecture allows for optimizing performance based on specific requirements.Input and weight stationary dataflows affect memory access patterns reducing memory bandwidth requirements and providing more efficient memory utilization, leading to higher performance and lower power consumption.

The NPU architecture, prioritizing adaptability, incorporates an array of PEs, each dedicated to executing MAC operations. These PEs are interconnected via a configurable interconnect module, facilitating seamless adaptation to both input- and weight-stationary dataflows. Furthermore, the architecture includes dedicated buffers for intermediate data storage and a data pre-processing unit, ensuring efficient data handling and transmission within the NPU. The incorporation of features such as varying stride, padding options, and pooling types further amplifies the adaptability of the NPU, rendering it well-suited for a wide array of CNN architectures and applications.

Moreover, the evaluation phase of the project encompassed a comprehensive analysis of the NPU's performance metrics, including execution time, resource utilization, and power consumption, across varying convolution layer parameters.The maximum frequency clock frequency achieved using this proposed design is 140 MHz and DSP slices on the FPGA were used to implement the multiply and accumulate operation for convolution to reduce power consumption.

5.2 Future Scope

Expanding this project could involve exploring additional dataflow models, like output- and row-stationary, to optimize memory access patterns and resource utilization within the NPU architecture. Investigating these models will deepen our understanding of their impact on NPU performance, informing future design decisions and optimizations.

Another avenue for future work could be enabling floating-point support in the current architecture, broadening the NPU's capability to handle a wider range of numerical values with greater precision. This enhancement would be particularly beneficial for tasks requiring higher precision computations, offering increased dynamic range and flexibility compared to fixed-point numbers. Additionally, incorporating floating-point representations could open doors to more sophisticated neural network architectures and algorithms, expanding the scope of applications where the NPU can be deployed.

Bibliography

[1] L. Zeng, X. Liu, and X. Zhang. *Neural Processing Unit: A Scalable and Programmable Accelerator for Neural Network Inference*. 2020. DOI: 10.1109/ACCESS.2020.3006418.

[2] Zakaria Rguibi et al. *CXAI: Explaining Convolutional Neural Networks for Medical Imaging Diagnostic*.

[3] H. Yu et al. *Understanding and Designing Neural Network Accelerators*. 2019. DOI: 10.1145/3299879.

[4] Vivienne Sze et al. *Efficient Processing of Deep Neural Networks: A Tutorial and Survey*. 2020. DOI: 10.1109/JSTSP.2020.3009164.

[5] Cristina Silvano et al. *A Survey on Deep Learning Hardware Accelerators for Heterogeneous HPC Platforms*. June 2023. eprint: 2306.15552v1.

[6] X. Liu et al. *RENO: A high-efficient reconfigurable neuromorphic computing accelerator design*. 2015.

[7] Y.-H. Chen, J. Emer, and V. Sze. *Eyeriss: A Spatial Architecture for Energy-Efficient Dataflow for Convolutional Neural Networks*. 2016.

[8] Hyoukjun Kwon, Ananda Samajdar, and Tushar Krishna. *MAERI: Enabling Flexible Dataflow Mapping over DNN Accelerators via Reconfigurable Interconnects*. 2018.

[9] Haoran Li et al. *A Flexible Dataflow CNN Accelerator on FPGA*. 2023.

[10] Jiaqi Yang, Hao Zheng, and Ahmed Louri. *Adapt-Flow: A Flexible DNN Accelerator Architecture for Heterogeneous Dataflow Implementation*. 2022.

[11] Qing Yang and Hai Li. *BitSystolic: A 26.7 TOPS/W 2b–8b NPU With Configurable Data Flows for Edge Devices*. Mar. 2021. DOI: 10.1109/TCSI.2021.XXXXXXX.

[12] Xilinx. *AXI Basics 6 - Introduction to AXI4-Lite in Vitis HLS*. 2023.

[13] Serveh Kamrava, Pejman Tahmasebi, and Muhammad Sahimi. *Linking Morphology of Porous Media to Their Macroscopic Permeability by Deep Learning*. 2019. DOI: 10.1007/s11242-019-01352-5.

[14] Avinash Kumar, Sobhangi Sarkar, and Chittaranjan Pradhan. *Malaria Disease Detection Using CNN Technique with SGD, RMSprop and ADAM Optimizers*. 2023.

[15] Baeldung. *Convolutional Neural Network Channels*.

[16] Jia Chen et al. *Multiply accumulate operations in memristor crossbar arrays for analog computing*. 2021. DOI: 10.1088/1674-4926/42/1/013104.

A Appendix

A.1 Testcase 4 = Weight Stationary for Input size=9*9

A Simulation performed on an Input feature map of size =9*9 represented on Figure A.2 with **control signal = 00 (Weight stationary with Unicast)** finished with an execution time of 90 µs.

The parameters used for Testcase 4 is given in the table Table A.1 .

Parameter	Value
DATA WIDTH	4
INPUT_WIDTH	9
INPUT_HEIGHT	9
KERNEL_SIZE	3
WEIGHT_HEIGHT	3
WEIGHT_WIDTH	3
STRIDE	1
PADDING	0

Table A.1: Parameters for Testcase 4

```
input_matrix = np.array([[0, 1, 2, 3, 4, 5, 6, 7, 8],
                         [0, 1, 2, 3, 4, 5, 6, 7, 8],
                         [0, 1, 2, 3, 4, 5, 6, 7, 8],
                         [8, 7, 6, 5, 4, 3, 2, 1, 0],
                         [4, 4, 4, 4, 4, 4, 4, 4, 4],
                         [5, 5, 5, 5, 5, 5, 5, 5, 5],
                         [6, 6, 6, 6, 6, 6, 6, 6, 6],
                         [7, 7, 7, 7, 7, 7, 7, 7, 7],
                         [8, 8, 8, 8, 8, 8, 8, 8, 8]])

weight_matrix = np.array([[0, 1, 2],
                          [3, 4, 5],
                          [6, 7, 8]])
```

Figure A.1: Input feature Map (9*9) and Weights (3*3)

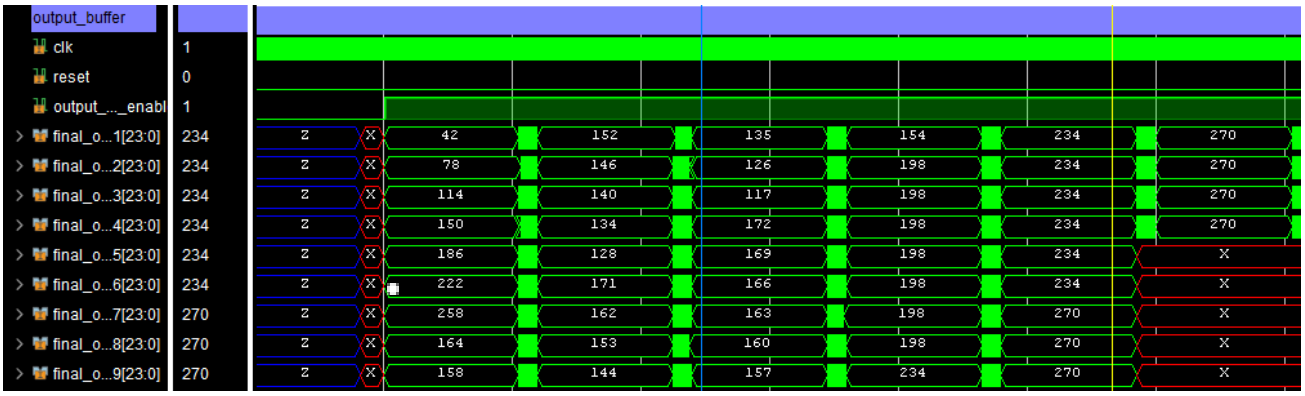


Figure A.2: Final Output feature Map (7*7)

```
Results of Max Pooling are
 150   258   158   171   153   172   150   258   158
 171   153   172   150   258   158   171   153   172
 150   258   158   171   153   172   150   258   158
 171   153   172   150   258   158   171   153
```

Figure A.3: Max Pooling Results

```
Average Pooling was selected
Results of Pooling are
  96   207   149   148   139   156    96   207   149
 148   139   156    96   207   149   148   139   156
  96   207   149   148   139   156    96   207   149
 148   139   156    96   207   149   148   139
```

Figure A.4: Average Pooling Results

A.2 Testcase 5 = Weight Stationary Multicast for Input Size =512*4

A Simulation performed on an Input feature map of size =512*4 represented on Figure A.6 with **control signal = 01 (Weight stationary with Multicast)** finished with an execution time of 1360 µs.

The parameters used for Testcase 5 is given in the table Table A.2 .

Parameter	Value
DATA WIDTH	8
INPUT_WIDTH	4
INPUT_HEIGHT	512
KERNEL_SIZE	3
WEIGHT_HEIGHT	3
WEIGHT_WIDTH	3
STRIDE	1
PADDING	0

Table A.2: Parameters for Testcase 5

```
Displaying Input Feature Maps:
  0   1   2   7
 12   5   6   7
 21  12   1   2
  3  13   5   6
  7  56   2   1
  2  13   4   5
  6  72   8   8
  7  10   5   4
  3   5   1   0
  4  34  12   4
  3   4   2   4
  0   5   7   5
  5   2   5  34
  5  87   6   6
 90   6   6   9
  6  10   6   7
  4   7   7   6
  7  10   7   7
  8   6   8   8
  4   5  34   8
  8   0   1   2
```

(a) First 21 elements of Input feature Map (512*4)

```
Displaying Weights Matrix:
  0   1   2   3   4   5   6   7   8
```

(b) Weights(3*3)

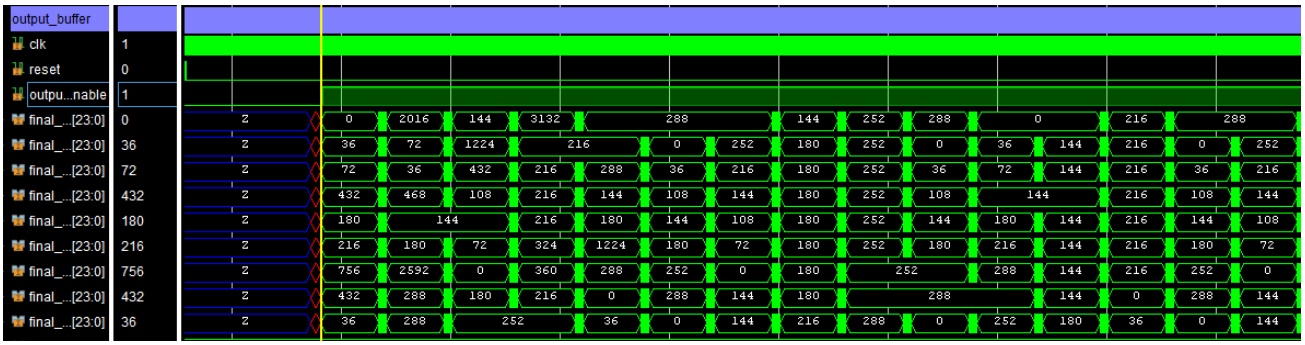


Figure A.6: Final Output feature Map (510*2)