

CALIFORNIA STATE UNIVERSITY, FRESNO

LYLES COLLEGE OF ENGINEERING

Electrical and Computer Engineering

SPRING 2025



ECE 298 PROJECT

**Title: High-Performance Design and Verification of a Hardware-Optimized CNN Accelerator
for Real-Time Object Detection Using YOLOv3 with Darknet-19 Architecture**

A Project submitted in partial fulfillment of the requirements for the degree of

Master of Science in Electrical and Computer Engineering

in the Lyles College of Engineering

California State University, Fresno

May 2025

SUBMITTED BY: MANASA KUNAPAREDDY (301828294)

ADVISOR: DR. NAN WANG Ph.D.

APPROVED

For the Department of Electrical and Computer Engineering

We, the undersigned, certify that the project of the following student meets the required standards of scholarship, format, and style of the university and the student's graduate degree program for the awarding of the master's degree.

Manasa Kunapareddy

Project Author

Nan Wang

Electrical and Computer Engineering

Zoulikha Mouffak

Electrical and Computer Engineering

Nusrat Jahan

Electrical and Computer Engineering

For the University Graduate Committee:

Dean, Division of Graduate Studies

AUTHORIZATION FOR REPRODUCTION OF MASTER'S PROJECT

Yes

I grant permission for the reproduction of this project in part or in its entirety without further authorization from me, on the condition that the person or agency requesting reproduction absorbs the cost and provides proper acknowledgment of authorship.

Permission to reproduce this project in part or in its entirety must be obtained from me.

Signature of project author: _____ **Manasa Kunapareddy** _____

ACKNOWLEDGEMENT

I would like to extend my sincere appreciation to my project advisor, Dr. Nan Wang, for his invaluable support, motivation, and thoughtful guidance during every phase of this project. His deep knowledge and dedicated mentoring significantly enhanced my project's outcomes and guided me in crafting a solution aligned with professional standards.

I am equally thankful to Dr. Reza Raeisi, Chair of the Electrical and Computer Engineering Department, whose consistent academic support has been instrumental during my graduate studies. His course on VLSI Digital System Testing, particularly the concepts of Design for Testability, ATPG, and fault coverage, laid a strong foundation in digital logic design that will undoubtedly benefit my future in the field of ASIC design and verification.

My sincere appreciation also goes to Dr. Mouffak for her continuous academic mentorship and encouragement. Her support significantly contributed to strengthening both my technical and professional skills and helped me gain confidence throughout my internship and academic journey.

I also want to sincerely thank Dr. Aaron Stillmaker for generously sharing his deep expertise and enthusiasm for Physical Design. His insightful advice and practical support were instrumental in shaping my project and inspired me to consider a career path in ASIC Physical Design. I deeply appreciate the time and valuable guidance he offered throughout.

A special note of thanks to Dr. N. Jahan for serving as a member of my project evaluation panel. Her expertise in Analog Circuit Design and Cryogenic CMOS Electronics for Quantum Computing has been both insightful and inspiring. Her valuable input during evaluations added great depth to my academic experience.

Finally, I would like to convey my heartfelt appreciation to my parents, Mastan Rao Kunapareddy and Vijayalakshmi Kunapareddy, for their constant love, encouragement, and unwavering faith in my abilities. Their endless support and sacrifices have profoundly shaped my academic path and have been essential to my successful completion of this master's degree.

Manasa Kunapareddy

May 2025

ABSTRACT

This research presents a new, powerful hardware system designed to speed up convolutional neural networks (CNNs) using Verilog HDL. It is specially created for the DARKNET-19 model, which is the main part of the YOLO v3 tiny algorithm, widely used for fast object detection. The CNN architecture was implemented using Verilog HDL and synthesized through Synopsys Design Compiler, focusing on achieving both functional accuracy and resource-efficient hardware design. The important purpose of this hardware is to perform key CNN tasks, such as convolution, pooling, and activation functions, quickly and efficiently, making real-time object detection possible and much faster than traditional methods. The hardware design uses advanced methods of parallel processing, meaning it can perform many calculations at the same time. This reduces delays and uses less power, providing a quicker and more energy-efficient solution for CNN tasks. The convolutional layers of DARKNET-19 are carefully organized onto this hardware, ensuring efficient storage and quick access to data, which helps improve processing speed and accuracy. A crucial part of this design is the preparation of images before they enter the CNN. This preparation includes resizing images, normalizing brightness, and adjusting color information to make sure the CNN can clearly understand and analyze them effectively. After preprocessing, images pass through several layers in the CNN. Convolutional layers first identify important features in the images, followed by pooling and activation layers, which further refine these features for better object detection. The processed data from the CNN is then analyzed by the YOLO v3 tiny algorithm, which accurately identifies and locates objects by drawing boxes around them and labeling each one correctly. Compared to standard CNN methods run on software, this hardware approach significantly reduces common issues like slow processing speeds and high energy usage. The outcome is a high-speed, dependable, and resource-efficient system, making it well-suited for applications including automation systems, security surveillance, and intelligent devices.

Keywords: Convolution Neural Networks (CNN), DARKNET-19, YOLO v3

TABLE OF CONTENTS

AUTHORIZATION FOR REPRODUCTION OF MASTER'S PROJECT	iii
ACKNOWLEDGEMENT	iv
ABSTRACT	v
LIST OF FIGURES	ix
LIST OF TABLES	x
1. INTRODUCTION	1
1.1 BACKGROUND:	1
1.2 MOTIVATION OF THE PROJECT	3
1.3 PROBLEM STATEMENT	4
1.4 OBJECTIVE OF PROJECT	4
1.5 PROJECT CONTRIBUTION	5
1.6 MATHEMATICAL BACKGROUND	6
1.6.1 CONVOLUTION OPERATION	6
1.6.2 POOLING OPERATION	7
1.6.3 FULLY CONNECTED LAYER	8
1.6.4 BATCH NORMALIZATION	8
1.7 LITERATURE REVIEW	8
1.8 ORGANIZATION OF THE REPORT	11
2. DESIGN AND ARCHITECTURE	12
2.1 INTRODUCTION	12
2.2 STANDARD CNN STRUCTURE	12
2.2.1 CONVOLUTION LAYER	12
2.2.2 CONVOLUTION LAYER SLIDING OPERATION	14
2.2.3 POOLING LAYER	15

2.2.4 ACTIVATION LAYER	16
2.2.5 FULLY CONNECTED LAYER	17
2.3 PADDING LAYER	18
2.4 DARKNET-19 STRUCTURE	18
2.5 BATCH NORMALIZATION	20
2.6 YOLO v3.....	22
2.7 PROPOSED ARCHITECTURE	24
2.8 BOOTH MULTIPLIER	26
3. TOOLS AND LIBRARY USED.....	28
3.1 XILINX VIVADO.....	28
3.2 SYNOPSYS DESIGN COMPILER	28
3.3 CADENCE INNOVUS.....	29
3.4 MATLAB	29
3.5 NANGATE OPEN CELL LIBRARY	30
4. ASIC IMPLEMENTATION.....	31
4.1 INTRODUCTION	31
4.2 FUNCTIONAL SIMULATION.....	32
4.3.1 DIAGNOSIS	44
4.4 LOGIC SYNTHESIS.....	44
4.4.1 AREA OPTIMIZATION	44
4.4.2 POWER OPTIMIZATION	45
4.4.3 TIMING ANALYSIS	46
4.5 PHYSICAL DESIGN	47
4.5.1 DESIGN IMPORTING	49
4.5.2 FLOOR PLANNING	50

4.5.3 ADDING GLOBAL NETS	51
4.5.4 ADDING POWER RINGS	52
4.5.5 SPECIAL ROUTING	52
4.5.6 ADDING POWER STRIPES	53
4.5.7 PLACEMENT.....	54
4.5.8 CLOCK TREE SYNTHESIS	55
4.5.9 INSERTING I/O FILLERS.....	56
4.5.10 VERIFYING DESIGN RULE CHECK	56
4.5.11 EARLY GLOBAL ROUTING AND NANO ROUTING.....	57
4.5.12 FINAL CHIP LAYOUT	58
4.5.13 GDS II STREAMOUT	59
5. RESULTS AND ANALYSIS	61
5.1 VERIFICATION OF FUNCTIONALITY OF CNN ACCELERATOR	61
5.2 POWER AND AREA ANALYSIS OF PROCESSING ELEMENT	62
5.2.1 POWER ANALYSIS	63
5.2.2 AREA ANALYSIS	63
6.CONCLUSION.....	65

LIST OF FIGURES

Figure 1: Structure of Standard CNN [11].....	12
Figure 2: Convolution layer operation [10]	13
Figure 3: Convolution layer sliding window operation [11].....	14
Figure 4: Structure of Pooling layer [10].....	15
Figure 5: ReLU Activation Function [11].....	17
Figure 6: Darknet-19 structure [14]	19
Figure 7: Batch Normalization [15].....	22
Figure 8: Proposed Architecture	24
Figure 9: Flowchart of the booth algorithm [24]	27
Figure 10: ASIC Implementation [25]	31
Figure 11: Functional Simulation output	32
Figure 12: 8-bit pixel values for blue channel	34
Figure 13: Green channel 8-bit pixel values	34
Figure 14: 8-bit pixel values of red channel	35
Figure 15: RGB row pixel data.....	35
Figure 16:Top-level module RTL Schematic	36
Figure 17: RTL schematic of internal architecture of conv_layer_0	36
Figure 18: RTL schematic of the systolic array	37
Figure 19: RTL schematic of processing element (PE)	38
Figure 20: RTL schematic of the conv_g_channel block with PE.....	39
Figure 21: RTL schematic of Booth multiplier module	40
Figure 22: RTL schematic of internal structure of a single Booth submultiplier	41
Figure 23: RTL schematic of 8-bit adder subtractor module	42
Figure 24: RTL schematic of 1-bit full adder(fa) module.....	43
Figure 25: Area optimization report.....	45
Figure 26: Power Optimization report	46
Figure 27: Timing Analysis report	47
Figure 28: Physical design flow [25]	48
Figure 29: Importing NETLIST and LEF file.....	49

Figure 30: Core Utilization Setup and Detailed Floorplan	50
Figure 31: Connecting Global nets	51
Figure 32: Power rings Configuration and Power rings	52
Figure 33: Special routing setup and Nets after routing	53
Figure 34: Power stripes	53
Figure 35: Standard cell placement.....	54
Figure 36: Clock Tree Synthesis	55
Figure 37: Filler Cells added on layout.....	56
Figure 38: DRC Violations after placement.....	57
Figure 39: post-routing violations check	58
Figure 40: Final Chip Layout.....	59
Figure 41: GDSII stream out.....	60
Figure 42: YOLOv3 detection results.....	61
Figure 43 Power comparison of different models.....	63
Figure 44 Area Comparison of different models	63

LIST OF TABLES

Table 1:Comparison of area and power between existing approaches and proposed model.....	58
--	----

1. INTRODUCTION

1.1 BACKGROUND:

Convolutional Neural Networks (CNNs) are increasingly important in analyzing visual data and understanding images. Taking inspiration from how the human brain processes visual inputs, CNNs use multiple layers that initially identify basic features such as edges and patterns, progressing to more complex characteristics including distinct shapes and identifiable objects. They have shown impressive performance in various visual applications, ranging from image classification and segmentation to tracking and particularly object detection. A significant advantage of CNNs is their capacity to autonomously identify relevant features from raw pixel data, thus removing the dependence on manually designed feature extraction methods [1].

Image processing is a vital step in the CNN pipeline and creates an important role in enhancing the efficiency and accuracy of visual recognition tasks. It includes techniques such as image resizing, grayscale conversion, histogram equalization, noise reduction, and normalization. These steps ensure that input images are consistent in size, scale, and intensity distribution, making it easier for the CNN to extract meaningful features [2]. For example, normalization converts pixel intensities into a standardized interval (such as between 0 and 1), enhancing training effectiveness. Advanced preprocessing methods frequently involve techniques including horizontal or vertical flips, rotations, or scaling, all of which significantly strengthen the network's robustness to varied input conditions [2]. Essentially, preprocessing connects initial image capturing and subsequent interpretation through deep neural models, guaranteeing optimized and uniform input for effective deep learning evaluation.

Identifying objects represents an essential function in visual analysis that detects which elements exist within an image and accurately determines their positions by employing rectangular boundary indicators. Unlike straightforward classification, which allocates a single category to a complete image, object detection algorithms must recognize and precisely localize numerous items of different dimensions and orientations within the same image frame.

. Earlier approaches, such as the R-CNN series (including Fast and Faster R-CNN), followed a two-step method—first generating potential object regions and then classifying each one. While these methods achieved high accuracy, they were computationally expensive and slower. The YOLO (You

Only Look Once) series revolutionized this process by treating object detection as a unified regression task, enabling the model to detect bounding box coordinates and class labels in one pass. This greatly enhances processing speed, making YOLO an ideal solution for real-time systems like autonomous drones, robotic vision, and smart surveillance [1].

YOLOv3 Tiny is a streamlined version of the YOLO architecture, specifically tailored for devices with limited computational power. It relies on a lightweight backbone network known as Darknet-19, composed of 5 max-pooling layers and 19 convolution layers, enabling faster processing while still maintaining acceptable detection performance. Though shallower than Darknet-53 used in the full YOLOv3 model, Darknet-19 still captures essential visual patterns with a much lower computational cost. This makes it ideal for use in embedded systems, FPGAs, and other low-power platforms where computational resources are limited but real-time performance is still required. YOLOv3 Tiny with Darknet-19 retains much of the detection capability of its larger counterparts, while significantly reducing memory usage and latency [2].

Despite the improvements in algorithm design, CNN-based object detection still demands substantial computational power due to the repetitive and complex operations involved, especially in convolutional layers. These layers perform numerous matrix multiplications and accumulation operations, which demand significant computational effort and become slow when carried out sequentially on standard processors. Furthermore, running these models on CPUs or GPUs in portable or embedded devices poses challenges such as increased power consumption, longer processing times, and heat generation. These limitations become significant barriers when deploying CNNs in real-time edge applications like autonomous navigation, where decisions must be made within milliseconds, and power budgets are tight [1].

To address these challenges, hardware accelerators especially Field-Programmable Gate Arrays (FPGAs) have emerged as a practical and efficient platform for deploying CNNs. FPGAs offer a balance between performance and flexibility. They can be programmed to perform specific tasks in deeply parallel pipelines, allowing multiple operations (e.g., multiplications in convolution) to be executed simultaneously. This parallelism drastically improves throughput and reduces latency. In addition, FPGAs offer significant power savings compared to GPUs and allow custom memory management and data reuse strategies that further enhance efficiency. Unlike ASICs (Application-

Specific Integrated Circuits), which are fixed at the time of manufacturing, FPGAs can be reprogrammed and optimized for different CNN architectures, making them ideal for research and prototyping.

In this project, a hardware accelerator is developed on an FPGA platform, tailored for the Darknet-19 backbone of YOLOv3 Tiny. The design includes optimized logic blocks for major CNN operations such as convolution, pooling, and ReLU activation, as well as components for image preprocessing. The goal is to enable real-time image analysis with minimal power consumption. The system first processes input images through preprocessing steps like resizing, normalization, and channel reordering to make them suitable for hardware input. These images are then passed through the CNN layers, where the hardware computes the required features in parallel. Finally, the generated feature maps are forwarded to the detection module, which completes the object classification and localization.

This approach not only improves speed and efficiency but also enables deployment of advanced object detection capabilities on compact, energy-constrained devices. Applications such as autonomous drones, smart surveillance cameras, medical imaging tools, and industrial robots stand to benefit significantly from this kind of embedded intelligence. By offloading the computation to a specialized hardware accelerator, the system can deliver consistent performance even under tight energy and space constraints.

1.2 MOTIVATION OF THE PROJECT

Convolutional Neural Networks (CNNs) are essential components within contemporary technological advancements, powering a wide range of applications such as self-driving cars, surveillance systems, and intelligent IoT devices. CNNs have proven highly effective at recognizing and classifying objects in images, making them valuable tools for real-time applications. However, CNN models require considerable computing power, leading to high energy use and slow processing when run on conventional hardware such as CPUs and GPUs. These challenges limit the effectiveness of CNNs, especially in portable or embedded devices where energy efficiency and quick performance are essential.

This project addresses these limitations by developing a specialized hardware accelerator using Field-Programmable Gate Arrays (FPGAs), designed specifically for the DARKNET-19 model in the YOLO v3 tiny algorithm. Although hardware accelerators exist, this project focuses on a unique implementation using DARKNET-19, optimizing the CNN operations for improved real-time performance.

1.3 PROBLEM STATEMENT

The widespread adoption of artificial intelligence in everyday technologies has created an increasing demand for rapid and energy-efficient processing capabilities. CNNs are powerful models mostly used for tasks like object detection and classification of images. However, their complex computations make them resource-heavy, leading to increased processing time and high energy consumption when run on conventional hardware like CPUs and GPUs. These performance demands create major obstacles for using CNNs in real-time scenarios, particularly on embedded systems where computational resources and power availability are limited. Therefore, there is a critical need for developing a specialized hardware accelerator optimized specifically for the DARKNET-19 model within the YOLO v3 tiny algorithm. Such an accelerator would improve real-time CNN performance by reducing latency and power consumption, thus enhancing their usability in critical applications like autonomous driving, intelligent security systems, and IoT-based solutions.

1.4 OBJECTIVE OF PROJECT

This project primarily aims to design and implement a specialized hardware accelerator using Field-Programmable Gate Arrays (FPGAs) specifically optimized for the DARKNET-19 model, which is the core of the YOLO v3 tiny algorithm. The goal is to significantly enhance the speed and efficiency of convolutional neural networks (CNNs) using DARKNET-19 to achieve real-time object detection performance. This project aims to effectively handle key CNN tasks, including convolution, pooling, and activation functions, through parallel processing techniques that reduce latency and minimize power consumption. Additionally, the project emphasizes proper image preprocessing methods, such as resizing, normalization, and color adjustments, to ensure accurate and reliable CNN analysis. CNN architecture is built in Verilog and synthesized this architecture to compare area and power and finally implementing physical design of the architecture.

1.5 PROJECT CONTRIBUTION

This project covers the comprehensive architectural development and assessment of a specialized hardware accelerator designed specifically for Convolutional Neural Networks (CNNs), optimized for the DARKNET-19 backbone used in the YOLO v3 tiny algorithm. The design was developed using Verilog HDL and simulated using Xilinx Vivado to verify its functionality. Key CNN operations such as convolution, pooling, and activation functions were implemented and tested for correctness. Parallel processing techniques were applied to improve computational speed, while clock gating methods were explored to reduce dynamic power consumption.

Image preprocessing modules were also implemented and tested to ensure that input images were properly resized, normalized, and formatted for efficient CNN processing. The complete design was synthesized using Synopsys Design Compiler, utilizing a 45nm CMOS technology library, providing accurate insights into area, power, and timing characteristics. This synthesis step helped evaluate how well the architecture performs regarding energy consumption and efficient usage of hardware resources.

Following synthesis, I conducted functional simulation and verification to observe the behavior of CNN layers and confirm their expected outputs. Special attention was given to analyzing the hardware mapping of the convolution layers and how data is managed across the accelerator pipeline. Additionally, testing and debugging were carried out at the Register Transfer Level (RTL) using Xilinx Vivado tool to confirm that the logic structure adhered to the design specifications.

To check how efficient and scalable the design is in terms of power, I performed RTL synthesis using Synopsys Design Compiler. For this process, I used a 45 nm CMOS NAND library to obtain an accurate estimation of the design's power and area characteristics. Synopsys tool converted the RTL design into a gate-level netlist and performed optimizations for better area, power consumption, and speed. To ensure the robustness and performance of the accelerator in real-time scenarios, timing analysis and clock tree synthesis were performed using Cadence Innovus tool. The complete hardware design flow included placement, routing, and static timing checks. Finally, a GDSII file was generated for tape-out preparation, representing a fully integrated, manufacturable design.

1.6 MATHEMATICAL BACKGROUND

1.6.1 CONVOLUTION OPERATION

At the heart of CNNs lies the convolution operation. For an input feature map $C[m,x,y]$, the output feature map $G[n, x, y]$ using kernel weights $K[n,i,j]$ is given by:

$$G[n, x, y] = \sum_{i=-\frac{W_k}{2}}^{\frac{W_k}{2}} \sum_{j=-\frac{H_k}{2}}^{\frac{H_k}{2}} \sum_{m=0}^{M-1} C[m, x + i, y + j] \cdot K[n, i, j] \quad [10] \dots \dots \dots \quad (1)$$

This equation performs weighted sums over spatial neighborhoods and channels, allowing the network to detect features such as edges, textures, or object parts. The general form of CNN can be derived in 4 steps.

Step 1: Understanding 2D Convolution

In image processing, convolution involves sliding a small kernel (also called a filter) across an input image to produce an output feature map.

Assume:

- Input image: $I(x, y)$
- Kernel: $K(i, j)$
- Output (convolved feature map): $O(x, y)$

The basic 2D convolution without channels is:

$$O(x, y) = \sum_{i=0}^{k-1} \sum_{j=0}^{k-1} K(i, j) \cdot I(x + i, y + j) \quad [1] \dots \dots \dots \quad (2)$$

Step 2: Include Multiple Channels (e.g., RGB)

In CNNs, images have depth for example, RGB images have 3 channels. So, for a single output pixel, the convolution includes contributions from all channels.

Let:

- C_{in} : Number of input channels

- $K(c,i,j)$: Kernel weight at position (i,j) for channel c .
 - $I(c,x+i,y+j)$: Input at channel c , shifted by (i,j)

Now, the convolution becomes:

$$O(x, y) = \sum_{c=0}^{Cin-1} \sum_{i=0}^{k-1} \sum_{j=0}^{k-1} K(c, i, j) \cdot I(c, x+i, y+j) [1] \dots \dots \dots \quad (3)$$

Step 3: Add Multiple Output Channels

CNNs learn multiple filters (kernels) in a layer, each producing a separate output feature map (channel).

Let:

- Cout: Number of output channels
 - Each output channel n has its own kernel K_n

So, the complete convolution equation is:

$$O(n, x, y) = \sum_{c=0}^{Cin-1} \sum_{i=0}^{k-1} \sum_{j=0}^{k-1} K(n, c, i, j) \cdot I(c, x+i, y+j) [1] \dots \dots \dots \quad (4)$$

Step 4-Optional-Adding bias b_n

Often, a bias term b_n is added per output channel:

$$O(n, x, y) = bn + \sum_{c=0}^{Cin-1} \sum_{i=0}^{k-1} \sum_{j=0}^{k-1} K(n, c, i, j) \cdot I(c, x+i, y+j) [1] \dots (5)$$

1.6.2 POOLING OPERATION

Pooling layers, particularly max pooling and average pooling, perform feature down sampling. The average pooling is mathematically defined as:

$$Output = \frac{1}{h \times w} \sum_{i=1}^h \sum_{j=1}^w x_{i,j} [10] \dots \dots \dots \quad (6)$$

For max pooling, the output is simply:

Where $x_{i,j}$ represents the elements within the pooling window, and h and w represent the pooling window height and width respectively.

1.6.3 FULLY CONNECTED LAYER

The fully connected layer operation can be mathematically represented as:

$$Y[n] = \sum_{m=0}^{M-1} X[m].W[m, n] + B[n] \quad [12] \dots \dots \dots \quad (8)$$

- $X[m]$ is the flattened input vector.
 - $W[m,n]$ are the weights connecting input neuron m to output neuron n .
 - $B[n]$ is the bias term added to neuron n .
 - $Y[n]$ is the output from neuron n

1.6.4 BATCH NORMALIZATION

Batch normalization helps stabilize the training process by normalizing the outputs from layers. The operation is mathematically defined as:

$$\hat{x} = \frac{x - \mu}{\sqrt{\sigma^2 + \epsilon}} \quad \dots \dots \dots (9)$$

Where:

- x is the input value.
 - μ is the mean calculated as:

1.7 LITERATURE REVIEW

Zhao et al. propose a method to accelerate convolutional neural networks (CNNs) using FPGAs, focusing on the LeNet-5 model [3]. They begin by pruning the network using Dynamic Network Surgery (DNS) to reduce the number of parameters. Next, the remaining parameters are quantized to

8-bit precision and then further encoded into 5-bit representations, allowing the use of shift operations in place of traditional multipliers. This technique enables the development of a parallel, LUT-based accelerator architecture that avoids the use of DSP blocks. The accelerator is implemented using Verilog and achieves a performance of 33.6 GMACS while consuming only 1.758 W of power on a Xilinx ZC702 FPGA. Despite the reduced precision, it maintains a high accuracy of 98.9% on the MNIST dataset [3].

Veena et al. propose a resource-optimized hardware implementation of a CNN useful for image processing applications, utilizing the VGG16 architecture [4]. Recognizing the difficulties of deploying deep learning models on resource-limited devices like IoT and edge systems, they focus on optimizing the CNN for FPGA platforms using Verilog. Their work involves reducing the size of feature maps and applying image compression techniques that are aware of hardware limitations. These optimizations help to significantly decrease memory usage and computational load, making the model suitable for real-time applications. Additionally, MATLAB is used to preprocess input images and convert them into a binary format compatible with Verilog. This preprocessing ensures smooth integration between software and hardware components. The result is a compact, faster, and energy-efficient CNN that performs effectively on devices with limited processing power and storage [4].

Karapurkar et al. design an energy-efficient CNN accelerator by focusing on optimizing the processing element (PE) and memory system. They propose a row-stationary dataflow architecture that reduces data movement and improves PE utilization [5]. The system uses scratchpads and local buffers to reduce energy per memory access. Implemented in System Verilog, their design is validated on FPGA and tested with ASIC tools. The architecture is designed to perform 2D convolution efficiently, and each PE uses pipelined MAC units. Their solution offers an effective balance between power and Computational efficiency, suitable for real-time CNN processing.

Crumley et al. focus on rehosting the YOLOv2 framework onto an FPGA to enable low-power object detection. They convert CNN models, written in high-level languages like C++ and Python, into Verilog using Vivado HLS [6]. This hardware accelerator is then deployed on a ZYNQ PYNQ-Z1 FPGA board. Their system uses quantized data and integrates with a System on Chip (SoC) platform. Testing on the COCO dataset reveals that execution time for convolution layers scales with image

resolution and layer complexity. This work demonstrates the feasibility of porting complex CNN frameworks like YOLO to energy-efficient FPGA platforms.

Lee et al. introduce a reconfigurable and scalable AI acceleration hardware platform for real-time seizure detection using EEG signals [7]. The system includes a RISC-V controller and a CNN coprocessor, built using System Verilog. The CNN model is simplified for energy efficiency and implemented with a reconfigurable architecture, allowing it to support various deep learning models. The system achieves 99.06% accuracy with just 0.108 W power consumption at 1 MHz frequency on a PYNQ-Z2 FPGA [7]. This makes it highly suitable for wearable biomedical devices that require continuous monitoring with low power use.

Song et al. design a lightweight CNN accelerator for IoT devices using Verilog. They focus on reducing the computational cost by simplifying the Multiply and Accumulate (MAC) units and quantizing 32-bit floating-point weights and biases to 8-bits [8]. This quantization greatly reduces power and memory requirements, making it ideal for IoT devices. The system is tested using the MNIST dataset, achieving 95% accuracy despite the lower bit width. Their design maintains both efficiency and accuracy, and it is implemented using Cadence tools and tested on an FPGA board [8].

Thomas et al. propose an improved processing element (PE) unit for CNN accelerators, using Modified Booth Encoded (MBE) multipliers and Wallace Tree Adders [9]. These optimizations are implemented on a Winograd-based architecture called UniWiG, which is designed to handle small kernel convolutions efficiently. The use of MBE and Wallace trees helps reduce hardware complexity and power consumption. The design is implemented in Verilog and tested on FPGA, showing better performance compared to traditional PE units. This work is particularly useful for real-time image processing tasks in power-sensitive environments.

Yulan Shen addresses the difficulty of running CNNs on limited-resource systems by using MobileNet, a lightweight model, and implementing it on an FPGA [10]. MobileNet is known for using depthwise separable convolutions instead of standard convolutions, leading to a substantial reduction in the total number of parameters and multiply-accumulate operations. In the paper, Shen designs a hardware accelerator consisting of 64 parallel processing units capable of handling depthwise and pointwise convolutions, pooling, and ReLU operations [10]. The system is implemented on a Xilinx UltraScale+ ZU104 FPGA and applied to a real-world gesture classification task. The implemented

network processes 128×128 grayscale images and includes compressed MobileNet layers optimized for embedded applications. The accelerator achieves a $28.4 \times$ speed-up compared to CPU and a $6.5 \times$ speed-up compared to GPU while maintaining a power consumption of only 4.07W. Despite being simplified, the model demonstrates strong performance in terms of both efficiency and energy use, processing up to 1250 frames per second with 304.9 fps/W efficiency. The work highlights the power of FPGAs in providing high-speed, low-power solutions for deploying CNNs in real-time edge applications.

1.8 ORGANIZATION OF THE REPORT

The structure of this project report is outlined as follows:

Chapter 1 Introduction- Provides a brief overview background, Problem statement, Objectives and literature survey on previous models.

Chapter 2 Design and Architecture-Describes about the CNN architecture design and shows the working and functionality of all required modules for CNN accelerator implementation in hardware.

Chapter 3 Tools used-This chapter talks about all the tools used in this project like Xilinx vivado, Cadence Synopsys, Cadence Innovus for physical design and finally MATLAB for verification of CNN accelerator for object detection

Chapter 4 ASIC Implementation-This chapter talks about the all the steps used for ASIC implementation and the talks about outputs at each step.

Chapter 5 Results and analysis-Presents about performance metrics such as area and power reports with previous models.

2. DESIGN AND ARCHITECTURE

2.1 INTRODUCTION

This chapter provides an introduction to key concepts like CNN design and architecture and components included in the CNN architecture like pooling layer, Activation function and RELU. This chapter also includes how these layers are useful in designing CNN accelerator for object detection and the algorithm of Darknet-19.

2.2 STANDARD CNN STRUCTURE

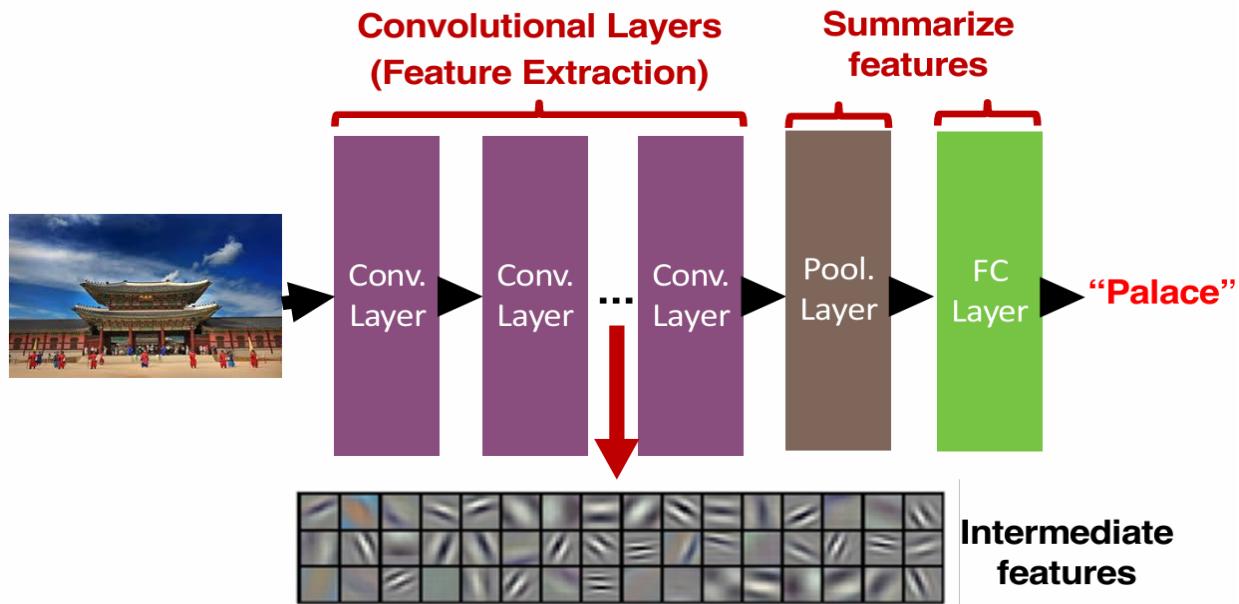


Figure 1: Structure of Standard CNN [11]

CNN is usually structured with different layers as shown in the above figure. We consider an image as an input layer and the values at the output representing the different classes or different pixel values of the images by extracting the feature maps. Starting from input to output, the layers in between them include convolutional layers, pooling layers, fully connected layers, and other essential components.

2.2.1 CONVOLUTION LAYER

A convolution layer serves as a fundamental component of a CNN. Its main job is to find useful features in an image, like edges or shapes. It uses small grids called filters (or kernels) that slide over

the input. At each position, the filter performs element-wise multiplication with the corresponding input values it covers and adds them together to get one output number. This process is known as the convolution operation, and the result is stored in a new image called the output feature map. Each output feature map highlights certain patterns, such as edges, corners, or textures, that the network learns during training [10].

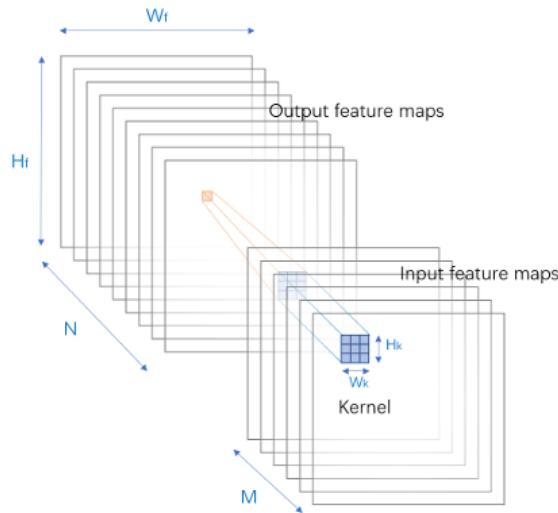


Figure 2: Convolution layer operation [10]

The figure visually explains how this convolution process works. On the right side, we see M input feature maps, which represent the data going into the convolution layer. These could be the original image or feature maps from a previous layer. A small kernel (shown as a cube) is put to a little portion of these input maps. Kernel has a width (W_k) and height (H_k), and it operates across all input channels. As the kernel slides over width (W_f) and height (H_f) of the input, it performs the multiply-and-add operations to produce a single value at each step [10]. The formal expression for this convolution operation is already provided in **Equation (1)** in Section 1.6.1. This equation captures how kernels interact with spatial and channel dimensions to extract local patterns like edges or textures. By applying multiple such kernels across the input feature maps, CNNs generate corresponding output maps capable of learning various features at different layers of abstraction.

These computed values are placed into new images called N output feature maps, shown on the left of the image. Each output feature map is generated using a different set of kernel weights, allowing

the network to learn multiple types of features. So, for example, one output map might detect vertical lines, another might detect corners, and so on. The more output channels (N) we have, the more feature types the network can learn. This process helps the CNN understand complex patterns in data, which is useful for tasks like image classification or object detection.

2.2.2 CONVOLUTION LAYER SLIDING OPERATION

In a standard color image, each pixel contains three separate color channels Red, Green, and Blue (RGB). These three channels represent the color intensity at each pixel location, and together they form the full image. When a convolutional neural network processes such an image. The convolution process is individually performed on each channel using the same or different filters (kernels).

In the below figure 3, we see three colored grids labeled as partial sums for the B-Channel (blue), G-Channel (green), and R-Channel (red). These represent the result of sliding a kernel over each individual color channel and performing multiply-and-add operations. Each element in these grids is the outcome of a convolution at that location for that specific channel. These are called "partial sums" because they are not yet combined into the result [11].

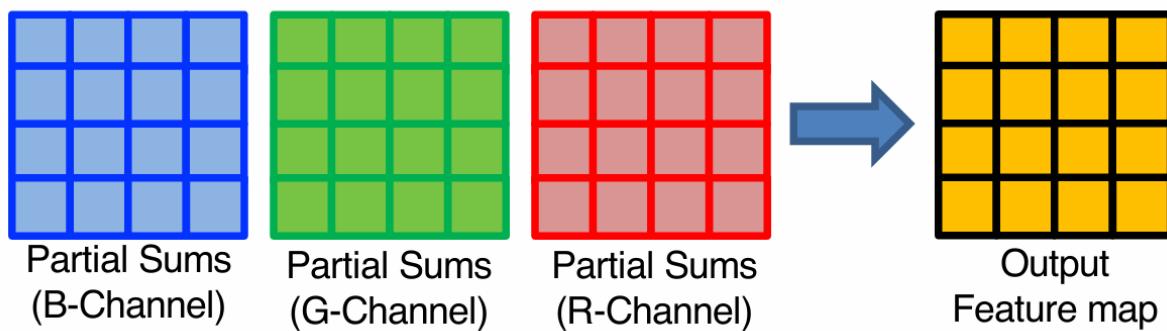


Figure 3: Convolution layer sliding window operation [11]

Once the convolution is completed for all three channels, the partial sums are added together elementwise. This means that at each (x, y) position, the value from the blue channel, green channel, and red channel are summed up to get a single value. This one value becomes a pixel in the feature map generated at output (highlighted in yellow) and then passed to the next layer of the CNN, and it contains the combined features extracted from all three-color channels. This process helps the network learn useful patterns like edges, textures, or shapes from the input image. We get these channels as

text files by preprocessing the image and these channels are now computed with different other layers of CNN [11].

2.2.3 POOLING LAYER

In a CNN, the pooling layer plays a key role in reducing the dimensions of feature maps. This reduction lowers computational demands, conserves memory, and retains the most significant features. Additionally, pooling enhances the model's robustness to minor variations or movements in the input image, such as slight shifts in object position. As a result, it improves generalization on new data and helps minimize the risk of overfitting [10].

The most widely used pooling methods are max pooling and average pooling. In max pooling, the feature map is divided into smaller regions (often 2×2), and from each region, we take the maximum value. This operation is replicated over the whole feature map, sliding 2×2 window with a step size called **stride** (usually stride = 2). If you apply this over a 4×4 feature map, the result will be a 2×2 output [10].

Pooling layers simplify the feature maps by reducing their spatial dimensions while retaining essential information. The most common pooling methods include average pooling and max pooling, both of which are fixed operations. Average pooling, as shown in **Equation (6)** from Section 1.6.2, computes the mean value across a defined region. Max pooling, on the other hand, selects the maximum value within the region, as described in **Equation (7)**. These pooling operations reduce the number of parameters and enhance the model's robustness to positional variations in input data. This makes them simple but very useful for making the network smaller, faster, and more focused on key features.

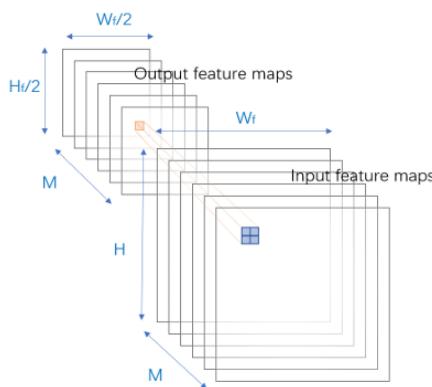


Figure 4: Structure of Pooling layer [10]

As shown in the above figure 2.4 the input image is classified into some feature maps by the stride. Pooling layer separated feature maps into more 2×2 non-overlapping rectangles and takes a max value in each rectangle by applying the max pooling operation. The output size is reduced to $1/4^{\text{th}}$ of input size as a result.

2.2.4 ACTIVATION LAYER

An activation function is a math-based rule used in neural networks to decide if a neuron should be turned on or stay off, helping the network learn complex patterns [10]. It adds non-linearity to the network, which means the model can learn complex patterns, like curves or edges in images. Without activation functions, a neural network would behave like a simple linear equation and would not be able to solve difficult problems like image recognition or language understanding [10].

The activation function is used right after a convolution or fully connected layer. If the previous layer gives a value called x , the activation function changes it to $f(x)$, where f is the function used. This new value is passed onto the following layer within the network.

Imagine if we are trying to teach a network to recognize cats and dogs. If the network could only use straight-line calculations (linear), it wouldn't be able to understand complex shapes like ears, eyes, or tails. The activation function allows it to learn from curved, detailed patterns, making it smarter and more flexible. Activation function is of three types. They are Rectified Linear Units (ReLU), Sigmoid, Scaled Exponential Linear Unit (SELU) etc. For my architecture I have used ReLU as activation function. It is applied after each convolution step to activate the feature maps effectively. If the input 'a' is less than zero, the output is set to 0, If input 'a' is positive output is unchanged--This is the required condition for ReLU.

$$ReLU(x) = \begin{cases} x & x > 0 \\ 0 & x \leq 0 \end{cases} \dots \quad [1]$$

The figure 5 shows the graph of the ReLU (Rectified Linear Unit) activation function, which is stated as $R(z)=\max(0, z)$. This means that if the input value z is greater than zero, the output is z itself; otherwise, if z is zero or negative, the output becomes zero. On the graph, you can see that for all negative input values, the line stays flat at 0, and for positive input values, it increases linearly with a slope of 1. This creates a sharp bend at zero, often called a "hinge." ReLU is widely used in neural

networks because it is simple, fast to compute, and introduces nonlinearity into the model which helps the network learn complex data. It also reduces the chances of vanishing gradients and makes the network more efficient by allowing only positive signals to pass through.

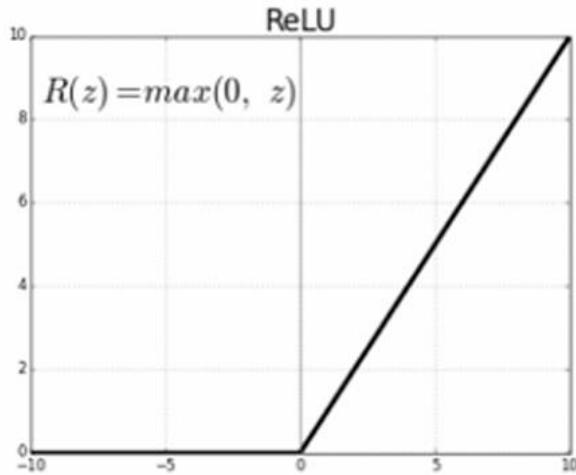


Figure 5: ReLU Activation Function [11]

2.2.5 FULLY CONNECTED LAYER

A dense layer, commonly referred to as a fully connected layer, is an important part of neural networks where each neuron is connected to every neuron in the preceding layer. This setup allows the network to learn complex patterns by combining all the extracted features. In CNNs, fully connected layers usually come after the convolution and pooling layers. Their role is to take the features found in different parts of the image and bring them together to make final decisions, like classifying what the image shows.

Before the fully connected layers can process the data, the output produced by the convolutional layers is initially reshaped into a single-dimensional vector. This transformation allows the network to take all the extracted features and use them for final decision-making. In the fully connected layer, each value in the flattened vector is scaled by a corresponding weight, followed by the addition of a bias term. This method helps the model learn detailed patterns and relationships within the data, making it a key step in many neural network architectures. This process is defined in **Equation (8)** of Section 1.6.3. This allows the network to form associations between spatial features and output categories based on learned weights.

2.3 PADDING LAYER

In Convolutional Neural Networks (CNNs), a padding is an important technique in inserting additional rows and columns, typically containing zeros, along the borders of the input image or feature map. Padding is typically applied before the convolution operation. Its main purpose is to preserve important information near the borders of image and control the output feature maps size. Without padding, every convolution operation decreases the size of feature map. For instance, using a 3×3 kernel on a 5×5 input without padding would give a smaller 3×3 output, causing the image to shrink after each layer and potentially losing important edge features.

There are different types of padding, but the most common one is called "same padding" or "zero padding," where zeros are added so that the size of the output keeps same as the input. Another type is "valid padding," which means no padding is applied at all, and the output becomes smaller. Some networks also use "replicate" or "reflect" padding, where the border values of the image are repeated instead of using zeros, although this is less common [13].

In CNN accelerators especially those implemented on hardware like FPGAs padding becomes even more critical. Since these accelerators are designed to process data in parallel and in fixed-size blocks, padding helps ensure that the data fits the expected shape and size. It also simplifies memory alignment and makes it easier to handle edge pixels during convolution. Efficient padding logic is often built directly into the hardware pipeline to keep processing fast and smooth. Without proper padding, the accelerator may miss key features at the borders or require extra logic to handle special cases, which can slow down the system or increase its complexity.

2.4 DARKNET-19 STRUCTURE

Darknet-19 is a compact and fast CNN architecture mainly used as the backbone for the YOLOv2 real-time object detection system. It includes 19 convolutional layers and 5 max-pooling layers, designed to provide a good balance between accuracy and processing speed. This makes it an ideal choice for applications that require real-time image processing with limited computational resources. Darknet-19 mainly uses small convolutional filters (like 1×1 and 3×3), which makes it easier to implement in hardware-based CNN accelerators like those on FPGAs. Additionally, it applies batch

normalization and ReLU activations throughout the network to improve convergence and performance [14].

In the context of CNN accelerators, Darknet-19 is particularly useful because of its simple and repetitive structure. The consistent use of 3×3 filters and the predictable layout of convolutional and pooling layers make it easier to map the network onto hardware logic blocks as shown in figure 6. This regular pattern allows for efficient pipelining, parallelism, and reuse of computation, which are all essential features for achieving real-time performance in FPGA and ASIC designs. Since CNN accelerators are typically constrained by memory and computation resources, the relatively compact size of Darknet-19 makes it an ideal choice for embedded and low-power systems without sacrificing too much accuracy.

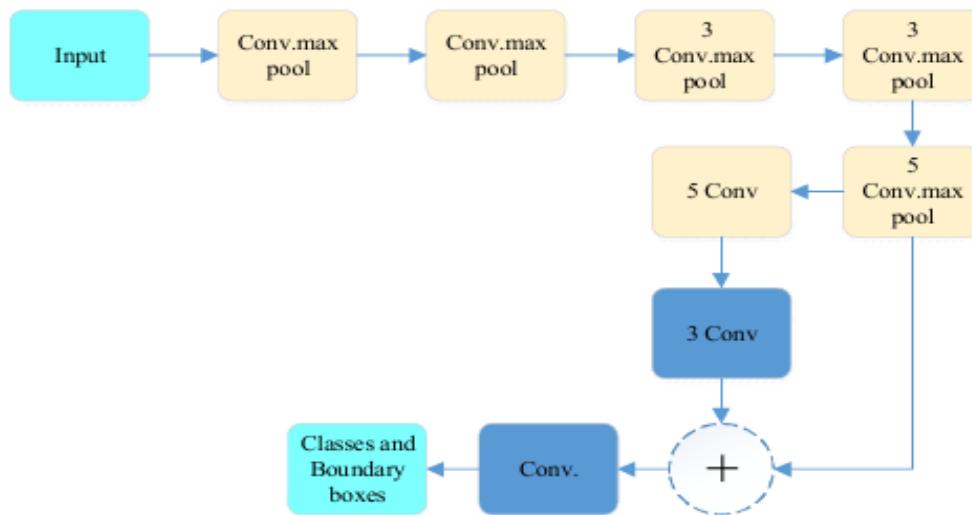


Figure 6: Darknet-19 structure [14]

The architecture shown in the figure 6 represents Darknet-19, which we are using as the backbone network for YOLOv3. Darknet-19 was originally designed for YOLOv2 and consists of 19 convolutional layers and 5 max pooling layers [6,14]. However, using it in YOLOv3 offers certain advantages, especially when targeting hardware-friendly implementations like FPGA-based CNN accelerators. Unlike Darknet-53, which is deeper and heavier, Darknet-19 is faster, smaller, and more efficient, makes it well-suited for real-time object detection on edge devices and platforms with limited resources [14].

In this architecture, the image given as input is sent to a sequence of Conv + Max Pooling layers. These early layers help extract basic features like edges and textures. As the network goes deeper, it uses multiple 3×3 and 5×5 convolutions. A key aspect of this design is the branching and merging mechanism: two paths extract features at different depths and scales, and their outputs are later combined (via the "+" symbol) before the final prediction. Combining low-level and high-level features allows YOLOv3 to accurately detect objects of various sizes, including both small and large ones.

When using Darknet-19 with YOLOv3, some modifications are typically made to adapt the backbone. YOLOv3 requires multi-scale predictions, so feature maps from different depths of Darknet-19 can be reused for prediction layers. In addition, a few extra convolution layers may be added on top of Darknet-19 to make it more compatible with YOLOv3's output requirements, which include finding class probabilities, objectness scores, and bounding boxes.

2.5 BATCH NORMALIZATION

When designing YOLO-specialized CNN accelerators on FPGA, whether to implement Batch Normalization (Batch Norm) can depend on several factors. Batch normalization plays an essential role in stabilizing neural network training and accelerating training speed [15]. However, it is not strictly necessary during inference. The pros and cons of each stage are described below.

Batch Normalization in Training Phase

Advantages:

- Faster Training Speed: Batch normalization helps neural networks converge faster during training.
- Enhanced Stability: Makes the training process more robust to variations in input data distribution.
- Deeper Network Capability: Effectively supports training of deeper neural networks.

Batch Normalization in Inference Phase

Disadvantages:

- Additional Computation Required: At inference, additional computation is needed due to the calculation of moving averages and variances for batch normalization.
- Increased Hardware Resource Usage: Implementing batch normalization on FPGA consumes additional computational resources and memory.

Deciding Whether to Use Batch Normalization

When to Use Batch Normalization:

- Computational Accuracy: Higher accuracy can be maintained due to the advantages of batch normalization.
- Network Compatibility: If the previously trained network already includes batch normalization, retaining it simplifies implementation.

When Not to Use Batch Normalization:

- Simplified Computation: Omitting batch normalization reduces computation, simplifying hardware design and potentially increasing inference speed.
- Memory Savings: Omitting batch normalization reduces the parameters that need to be stored in BRAM, thereby saving memory resources.
- Pre-computation: Pre-computing batch normalization's moving averages and variances during training and using them as fixed values during inference can achieve similar effects without additional runtime computation [15].

During inference, batch normalization improves the stability and convergence of the network by normalizing feature values. The operation standardizes the output distribution with zero mean and unit variance, as shown in **Equation (9)** of Section 1.6.4. Additionally, the batch mean used in normalization is computed using **Equation (10)**, allowing the hardware implementation to utilize pre-computed statistics for faster performance during deployment.

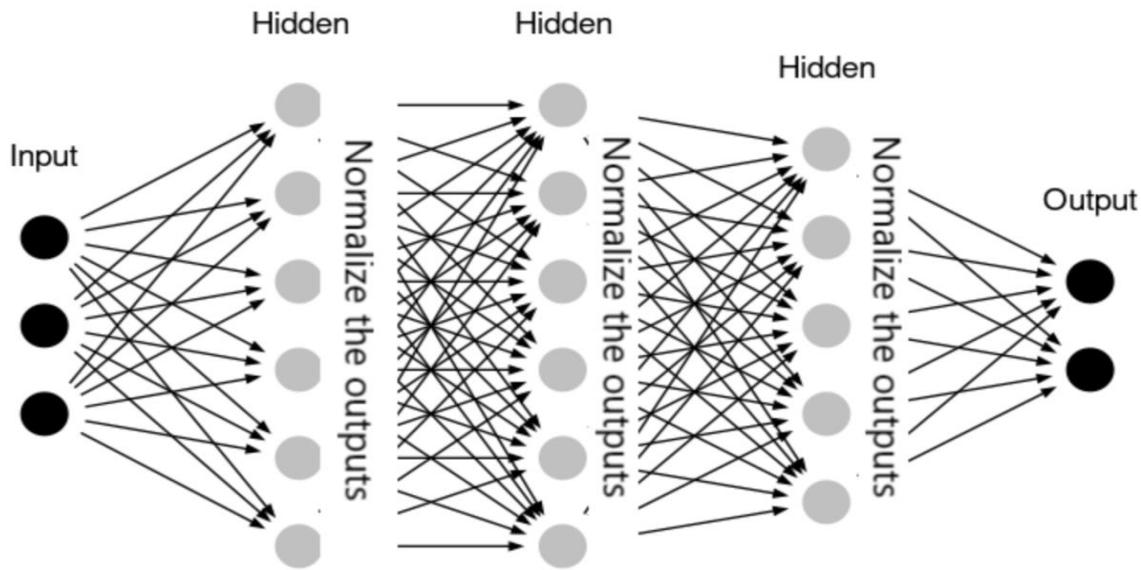


Figure 7: Batch Normalization [15]

2.6 YOLO v3

Object detection has advanced significantly in computer vision, especially with the rise of deep learning models capable of real-time performance. Among these, **YOLOv3 (You Only Look Once version 3)** stands out for offering an effective trade-off between detection speed and accuracy. Unlike earlier multi-stage approaches that first generate region proposals and then classify them, YOLOv3 uses a **single-stage architecture** that performs both localization and classification in one pass. This streamlined structure allows for faster, end-to-end training and real-time detection, making it highly suitable for applications where speed is crucial [16]. This makes YOLOv3 highly suitable for applications like autonomous vehicles, surveillance, and embedded systems.

The original YOLOv3 architecture uses Darknet-53 as its feature extractor, a deep CNN containing 53 convolutional layers with extra connections [16]. However, for hardware-constrained environments, such as FPGA-based CNN accelerators, this can be computationally expensive. To address this limitation, we propose the use of Darknet-19, a more lightweight and efficient architecture originally introduced in YOLOv2 [17]. Darknet-19 architecture contains of 19 convolutional layers and 5 max pooling layers, using 3×3 and 1×1 filters, along with batch normalization and Leaky ReLU

activations. Its simpler structure makes it ideal for hardware implementation while still retaining sufficient capacity for feature extraction.

When adapted into YOLOv3, Darknet-19 serves as the backbone network that processes the input image and produces intermediate feature maps. All these feature maps are then sent into the multi-scale detection head of YOLOv3, where predictions are made at three different scales (typically 13×13 , 26×26 , and 52×52). This approach helps the model for object detection of various sizes more effectively, using high-resolution maps for small objects and low-resolution maps for larger ones [18]. To enhance prediction accuracy, YOLOv3 uses anchor boxes predefined bounding box templates that help the model estimate objects with different aspect ratios and scales.

In YOLOv3, each grid cell in the output predicts several bounding boxes, along with details like the objectness score, the bounding box coordinates (x , y , width, height), and class probabilities [16]. The model is trained using a mix of cross-entropy loss for classification and mean squared error for predicting box locations. To improve the accuracy during inference, a technique called Non-Maximum Suppression (NMS) is applied to remove overlapping boxes, keeping only the most confident predictions [19].

The integration of YOLOv3 with Darknet-19 brings several benefits. First, it significantly decreases the model's computational load, allowing it suitable for real-time inference on edge devices like FPGAs and embedded platforms [20]. Second, it retains much of YOLOv3's detection capability due to its multi-scale structure, despite the lighter backbone. Lastly, the regular and compact structure of Darknet-19 simplifies hardware mapping, allowing for easier implementation of pipelined and parallel processing units in CNN accelerators.

In conclusion, the use of Darknet-19 as a replacement for Darknet-53 in YOLOv3 represents a strategic trade-off between model complexity and practical performance. It makes the detection framework more accessible to real-time and resource-constrained environments without a significant drop in accuracy. This modified architecture is especially valuable for systems requiring high-speed, low-power, and on-device intelligence.

2.7 PROPOSED ARCHITECTURE

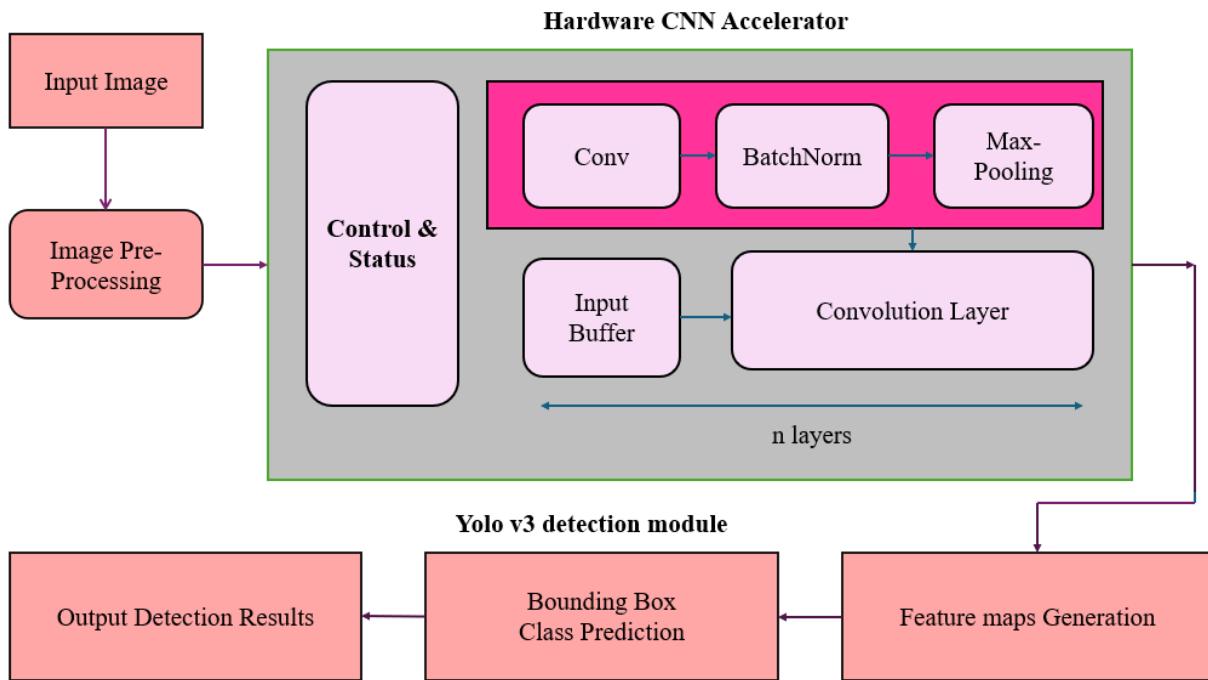


Figure 8: Proposed Architecture

Figure 8 illustrates the proposed architecture high performance hardware CNN accelerator using darknet-19 yolov3. The architecture begins with the Input Image, which could be captured from a real-time source like a camera or loaded from a dataset or image file. This image is in its raw form and cannot be directly used by the hardware CNN accelerator. Therefore, it goes through the Image Pre-Processing stage. This particular step is important because it prepares in a way that image matches the input requirements of the accelerator. During pre-processing, the image is resized to a standard dimension (such as 224×224), and the pixel intensities are scaled through normalization to bring them within a specific range, such as 0–1 or -1 to 1. In some cases, this stage may also involve converting the image format, changing the color channel order (like RGB to BGR), or performing quantization to convert floating-point values to fixed-point representation. These steps ensure that the image is lightweight, hardware-compatible, and optimized for fast processing.

After pre-processing, the formatted image is sent to the Hardware CNN Accelerator, which is the core of the architecture. This accelerator is implemented on an FPGA or ASIC platform and is responsible for performing all the heavy computation related to feature extraction. Inside the accelerator, a Control

and Status Unit oversees the entire operation. It manages timing, data flow between modules, and system coordination. It acts like the brain of the hardware module, ensuring that every part—like convolution units, buffers, and memory—works in harmony and without delay or conflict.

At the heart of the accelerator is the Input Buffer, which temporarily stores the preprocessed image data. This buffer ensures smooth data feeding into the convolutional layers. Instead of sending the entire image at once, it feeds pixel blocks row by row or tile by tile. This allows the CNN to work on smaller, manageable sections and supports pipelined and parallel processing.

Next comes the Convolution Layer, which contains multiple convolution blocks stacked as per the Darknet-19 architecture. Each block applies several 3×3 and 1×1 convolution operations using learned filters. These filters scan the image region and detect important visual features. The outputs from each convolution go through a Batch Normalization layer, which stabilizes the activations and improves convergence speed. Subsequently, a ReLU (Rectified Linear Unit) activation function is utilized to introduce non-linearity, enabling the network to capture more intricate patterns. This is followed by a Max-Pooling operation, which reduces the dimensionality of the feature maps while preserving the most significant information. This combination of Convolution → Batch Normalization → Max Pooling is repeated over multiple layers, helping the network extract both simple features like edges and textures, and more complex ones like shapes and object parts.

Once all layers in the Darknet-19 structure have been processed, the CNN accelerator outputs a rich feature maps. These represent the visual content of the image in a compressed and abstract form. They are passed from the hardware module to the YOLOv3 Detection Module, which operates in software (usually in MATLAB). This separation between feature extraction (hardware) and object detection (software) offers flexibility and efficiency.

The YOLOv3 detection module begins with Feature Map Generation, where it receives the extracted features and prepares them for the detection logic. YOLOv3 make the image divide into grid and applies prediction logic to each grid cell. The next step is Bounding Box and Class Prediction, where the network predicts several values per grid cell: the coordinates of bounding boxes (x , y , width, height), objectness scores, and class probabilities (what kind of object it might be like a person, car, or dog). YOLOv3 is known for using anchor boxes, which helps in detecting objects of different sizes and aspect ratios more effectively.

After prediction, the results are processed in the Output Detection Results stage. Here, post-processing is applied—most notably, Non-Maximum Suppression (NMS) is employed to eliminate redundant or overlapping bounding boxes by retaining only those with the highest confidence levels. Once NMS is applied, the resulting output presents the image annotated with bounding boxes and corresponding class labels, clearly showing the location and type of each detected object. This result is displayed to the user, marking the completion of the detection pipeline.

2.8 BOOTH MULTIPLIER

In Convolutional Neural Networks (CNNs), the multiply-and-accumulate (MAC) operation is one of the most essential and commonly executed computations. This operation is heavily used in convolution layers where input features are multiplied by corresponding kernel weights and summed to produce the output. Since these multiplications occur thousands of times across multiple layers, they become the performance bottleneck in hardware accelerators [21].

To tackle this challenge, the Booth multiplication algorithm is commonly used as an efficient solution. The Booth multiplier is a hardware-friendly algorithm designed to multiply signed binary numbers efficiently, especially in two's complement form. Instead of performing straightforward bit-by-bit multiplication, the Booth algorithm uses Booth encoding to decrease the no of partial products, especially when the multiplier contains a series of 1s. This results in fewer addition operations and less switching activity, which directly translates to faster computation and lower power consumption in digital hardware [22,24].

In fixed-point CNN accelerators, both the feature values and weights are typically represented as signed numbers, making the Booth multiplier a natural choice for handling these operations. The ability of Booth multipliers to directly handle signed arithmetic eliminates the need for additional hardware modules or conversion logic, further simplifying the design and saving on area and power consumption [23].

Furthermore, using a Modified Booth Multiplier in your CNN accelerator helps reduce the critical path delay, allowing system to operate at a higher clock speed. It also helps reduce area and power usage, which is crucial for real-time and embedded systems with limited hardware resources. This

makes the Booth multiplier not just a performance enhancer, but a practical necessity for efficient CNN hardware design. [24]

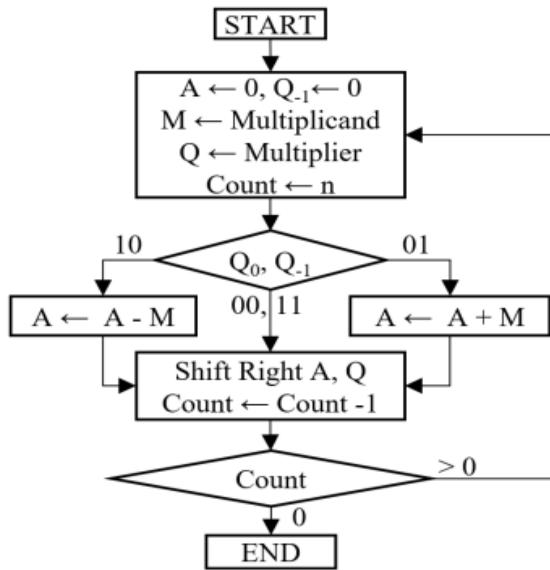


Figure 9: Flowchart of the booth algorithm [24]

The above figure 9 shows the algorithm of booth multiplier. The algorithm works based on the bits. If it's 00 or 11 then it directly shifts to right. If it's 10 or 01 then multiplicand will subtract or add from accumulator and then right shift occurs.

3. TOOLS AND LIBRARY USED

3.1 XILINX VIVADO

Xilinx Vivado Design Suite is a development environment specifically created by Xilinx for designing digital circuits on FPGA (Field-Programmable Gate Array) platforms. It supports both Verilog and VHDL, allowing users to write and simulate RTL code. In this project, Vivado was primarily used for logic simulation and RTL schematic generation. The simulation feature in Vivado helped verify the correctness of the design logic, especially for the Booth multiplier and CNN processing elements. The built-in waveform viewer was used to check input-output relationships and to ensure the design behaved as expected under different conditions.

Another important feature of Vivado is its ability to generate the RTL schematic, which gives a visual representation of the hardware structure based on the written Verilog code. This schematic includes all the internal connections, logic blocks, and data paths, and is useful for debugging and understanding the architecture of the design. Vivado also includes synthesis tools and support for IP (Intellectual Property) cores, but in this project, it was used mainly during the pre-synthesis phase for verification and visualization.

3.2 SYNOPSYS DESIGN COMPILER

Once the design is verified in simulation, it moves to the synthesis stage, which was performed using Synopsys Design Compiler. This tool is a powerful and widely used synthesis engine that converts high-level RTL code (written in Verilog) into a technology-mapped netlist. The netlist consists of standard cells from a selected technology library, which are used to build the final digital circuit. Design Compiler also applies optimizations during synthesis, such as logic minimization, pipelining, and balancing logic paths.

In this project, Design Compiler played an important role in generating power, area, and timing reports, which are essential for evaluating hardware performance. These reports provide information about how many gates are used (area), how much power the circuit will consume during operation, and whether it meets the required clock timing (setup and hold times). The tool supports constraint files, which allow designers to define target clock frequency, maximum area, and power budgets. Using

these capabilities, the Verilog-based CNN architecture was analyzed for efficiency and readiness for physical implementation.

3.3 CADENCE INNOVUS

After synthesis, the gate-level netlist is sent to physical design stage, which was handled using Cadence Innovus. Innovus is a physical design automation tool that performs place and route (P&R). During this phase, the synthesized logic cells are put on the chip area, and metal interconnects are being routed between them to form a working circuit. Innovus manages a range of essential tasks including floor planning, cell placement, clock tree synthesis, signal routing, and achieving timing closure.. These steps ensure that the physical layout of the chip follows all design rules and operates reliably at the target clock frequency.

In this project, Innovus was used to carry out physical implementation of the synthesized CNN accelerator. The layout process includes creating power grids, optimizing clock distribution, and minimizing routing congestion. The tool also generates detailed post-layout reports, which include the final area and wirelength, along with timing analysis that checks for violations like hold and setup failures. Moreover, Innovus performs verification checks for Design Rule Violations (DRC) and identifies any discrepancies through Layout Versus Schematic (LVS) comparison, ensuring that the final layout is ready for fabrication or further backend processes.

3.4 MATLAB

MATLAB (short for Matrix Laboratory) is a high-level environment commonly used for simulation, mathematical modeling, algorithm development, image processing, and data visualization. It offers a rich set of toolboxes and built-in functions that make it ideal for working with matrix operations, signals, and images. MATLAB is especially popular in academic and research domains for tasks, as it allows users to prototype and test complex systems quickly and accurately. Its support for graphical outputs, easy debugging, and integration with neural network libraries makes it a powerful tool for both software simulation and hardware co-verification.

In this project, MATLAB was used to verify the behavior and accuracy of the hardware CNN accelerator designed for object detection. The tool was used to preprocess input images (resizing,

normalization), simulate the expected output from convolution layers, and compare those outputs with the results produced by the RTL hardware implementation. MATLAB also helped visualize feature maps and object detection boundaries to ensure the CNN correctly detected and classified objects. This step was critical in confirming the functional correctness of the accelerator before moving to hardware synthesis and layout, ensuring that the design aligned with the expected CNN model behavior in practical object detection scenarios.

3.5 NANGATE OPEN CELL LIBRARY

The Nangate Open Cell Library is an open-source digital standard cell library that is publicly accessible and intended primarily for educational and research applications. In this project, the 45nm Nangate Open Cell Library was used during synthesis and physical design to map the RTL design to gate-level logic. The library includes a wide range of standard cells like logic gates, flip-flops, buffers, and multiplexers, each characterized for timing, power, and area. It provides essential files like Liberty (.lib) for synthesis, LEF for physical design, and GDSII for layout generation, making it highly suitable for ASIC flows. The availability of this library enabled accurate estimation of performance metrics such as cell power consumption, area, and timing delays, ensuring a realistic and practical design implementation throughout the backend process.

4. ASIC IMPLEMENTATION

4.1 INTRODUCTION

This chapter outlines the implementation of the proposed CNN accelerator design following the ASIC (Application-Specific Integrated Circuit) design flow. The ASIC implementation is an important step in verifying how the hardware design would perform if it were manufactured as a real chip. In this chapter, the design is synthesized using Synopsys Design Compiler to generate gate-level netlists and estimate area, power, and timing. Cadence Innovus is used to perform key physical design steps including cell placement, clock tree synthesis, and routing.

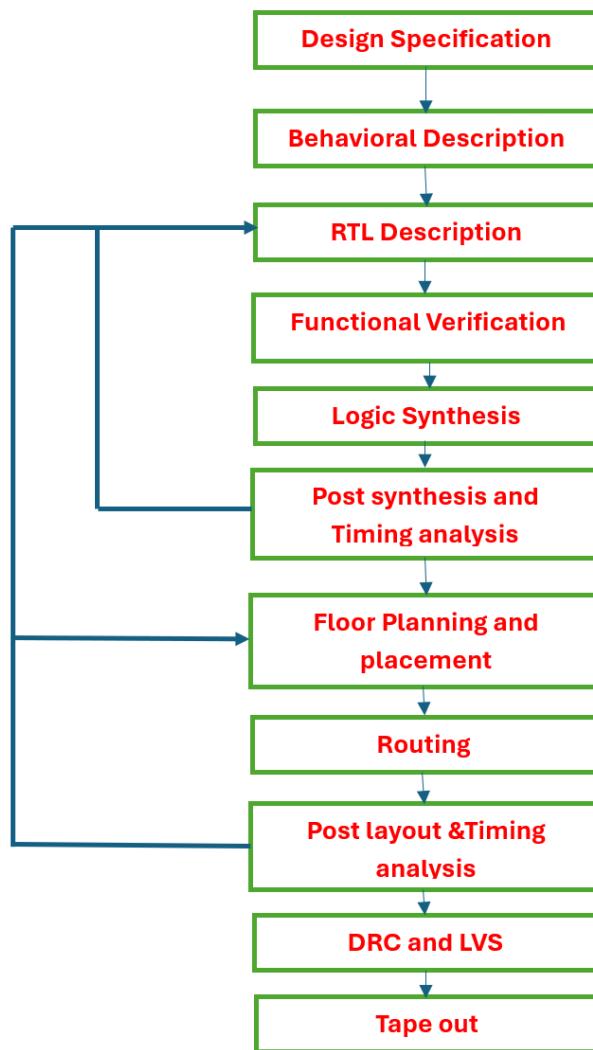


Figure 10: ASIC Implementation [25]

4.2 FUNCTIONAL SIMULATION

Functional simulation is an essential step in the digital design process to verify that the written RTL (Register Transfer Level) code behaves as expected before moving on to synthesis or physical design. In this project, I have performed using Xilinx Vivado, a widely used FPGA design tool. The simulation was carried out by applying different test inputs to the CNN accelerator modules, such as the Booth multiplier and convolution units, and observing the outputs to ensure correctness. Vivado's integrated simulation environment allowed for waveform analysis, signal tracing, and debugging of the logic design at the RTL level. This helped identify and fix any functional errors early in the design flow, ensuring that the Verilog code accurately represented the intended behavior of the hardware.

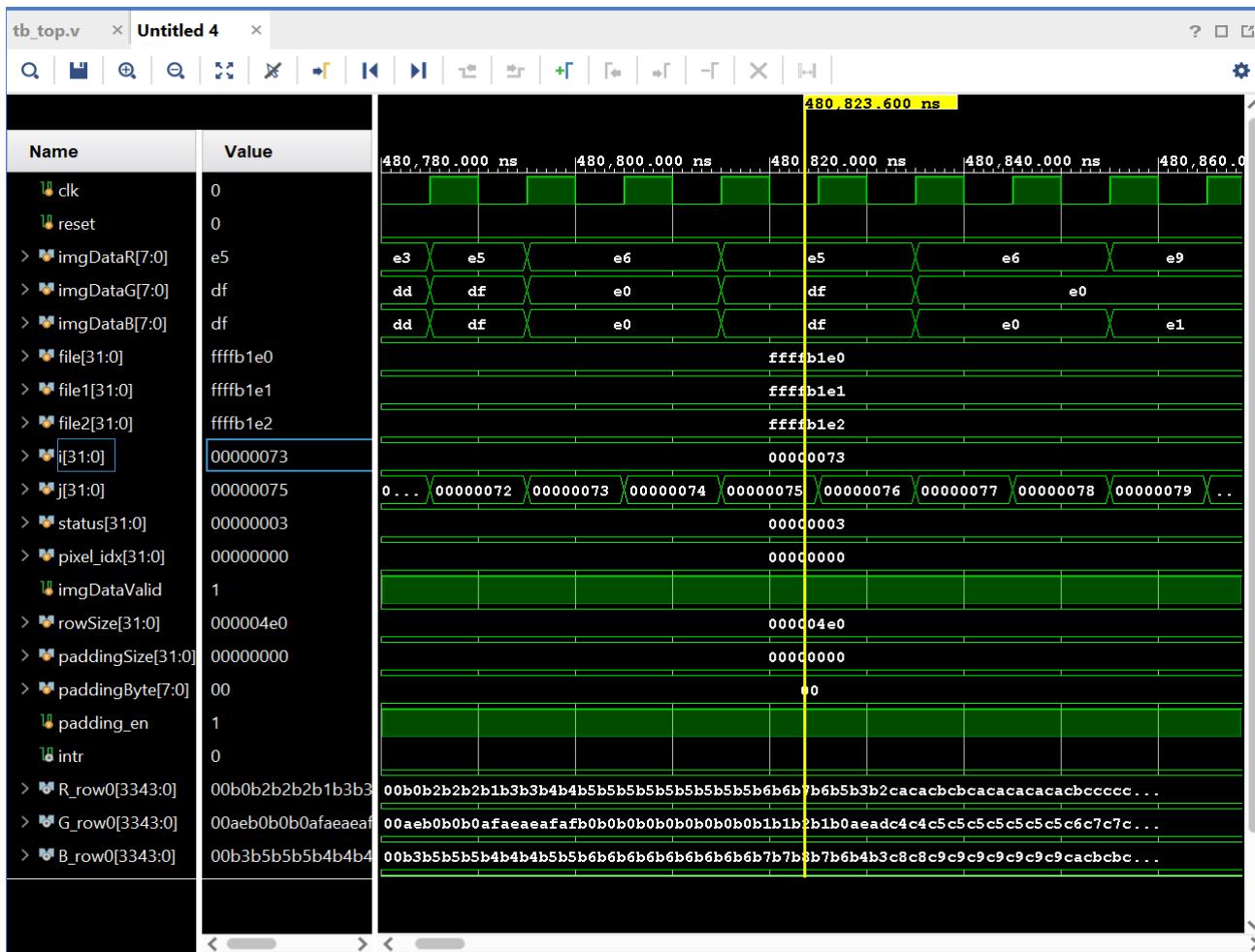


Figure 11: Functional Simulation output

The waveform shown in the above figure 11 represents the functional simulation output of the proposed CNN accelerator's image padding module. This simulation was performed using Xilinx Vivado, and it demonstrates how the internal signals behave over time when input image data is passed through the system. The simulation confirms that the logic is working correctly and that the data flows through the padding unit as expected.

At the top of the waveform, the clock signal (clk) is toggling at regular intervals, which is essential for driving all sequential operations in the design. The reset signal (reset) remains low (0), indicating that the module is not in reset state and is operating normally. This allows the internal processes, such as data reading, pixel indexing, and padding, to proceed.

We also observe signals like file, file1, and file2, each with a 32-bit width, holding values such as ffffb1e0. These likely represent memory addresses or indices used for simulation reference, ensuring that the pixel values are being read from the correct location in the simulated image array. This setup mimics real-world hardware reading image data from memory during live operation.

The loop control variables, such as i and j, are critical for navigating through image rows and columns. In the waveform, values like 00000073 and 00000075 indicate the current row and column positions of the pixel being processed. These indices ensure that pixels are padded and stored at the correct location in the internal memory structure. The pixel_idx signal likely tracks the one-dimensional index used when reading or writing pixel data sequentially.

The control signal imgDataValid becomes high (1) when the pixel data is valid and ready to be processed by the module. Similarly, padding_en is asserted (1) when padding is enabled, meaning the system is actively inserting padding bytes into the image rows. The status signal, holding a value like 00000003, could indicate the current finite state machine (FSM) state, guiding the operation through its various phases such as idle, read, write, or pad.

Padding-related configuration signals such as rowSize and paddingSize are also visible in the waveform. The rowSize signal, with a value like 000004e0 (1248 in decimal), defines the no of pixels in each row of the image. The paddingSize and paddingByte signals define the amount and value of the padding to be added at the image edges—typically zeros or constants to align the data for convolution operations.

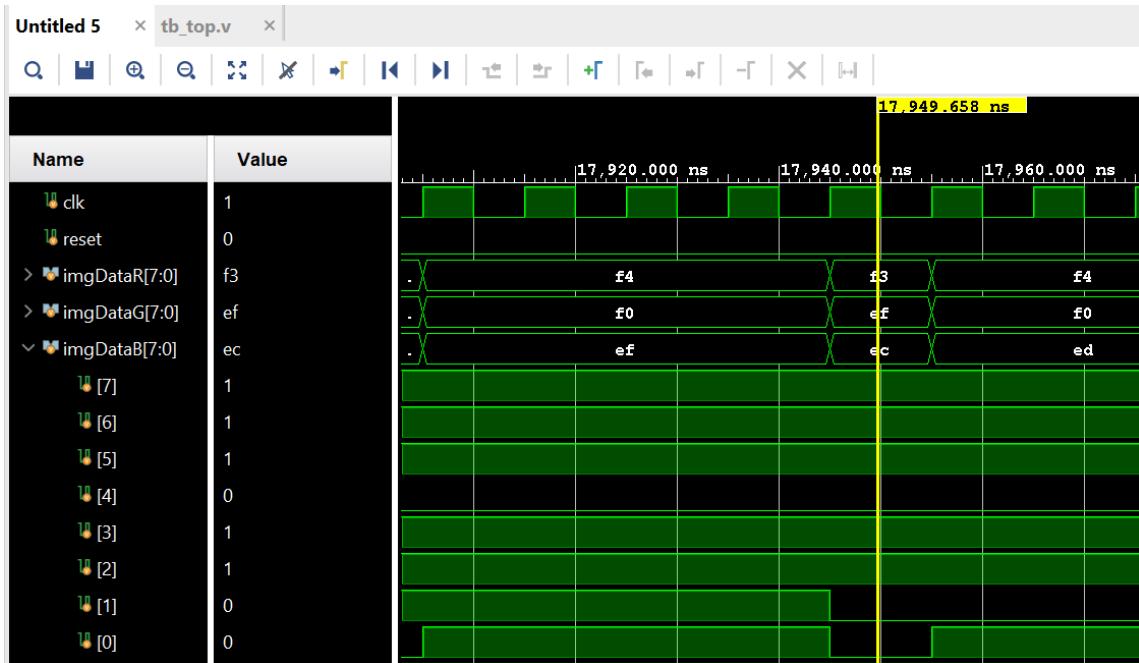


Figure 12: 8-bit pixel values for blue channel

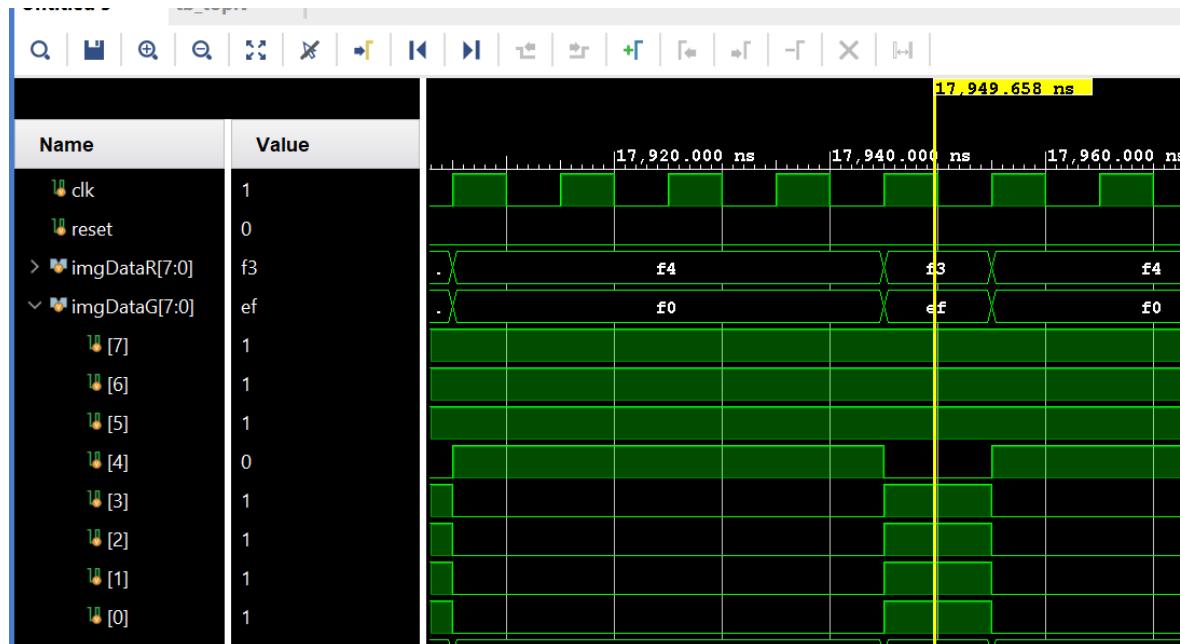


Figure 13: Green channel 8-bit pixel values

The signals imgDataR, imgDataG, and imgDataB represent the incoming 8-bit pixel values for the RGB channels of image as shown in the above figures 12, 13, 14. These values are shown in hexadecimal format, such as e5, df, e0, etc., and they change over time to simulate the streaming of

image data through the system. These inputs are handled by the padding logic and then written into internal row buffers for further convolutional processing.

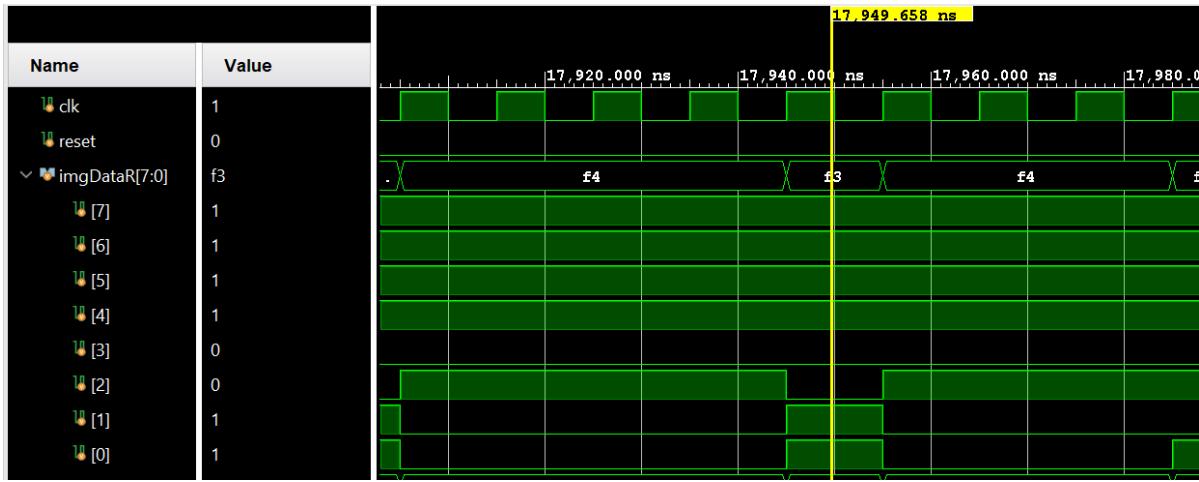


Figure 14: 8-bit pixel values of red channel

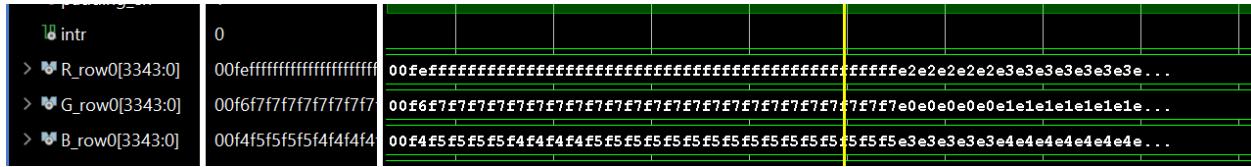


Figure 15: RGB row pixel data

Finally, the R_row, G_row, and B_row signals represent internal buffers that hold entire rows of the red, green, and blue pixel data respectively as shown in the above figure 15. These signals display hexadecimal strings such as 0b0b2b2b2b1b3b3b..., showing how the image data is being stored after padding. These buffers are essential for feeding the pixel data into convolution layers in a structured way.

4.3 RTL SCHEMATIC

The figure 16 represents the top-level RTL schematic of the proposed CNN accelerator, generated using Xilinx Vivado after elaboration of the Verilog code. This schematic shows the primary module named conv_layer_0, which acts as the central processing block for the convolution operation in the design. Two main input ports are visible: clk_i, the clock input that drives the sequential logic. These control signals are essential for coordinating the internal timing and operation of the CNN accelerator.

The schematic confirms that the top-level module is correctly instantiated and that all input signals are properly connected.

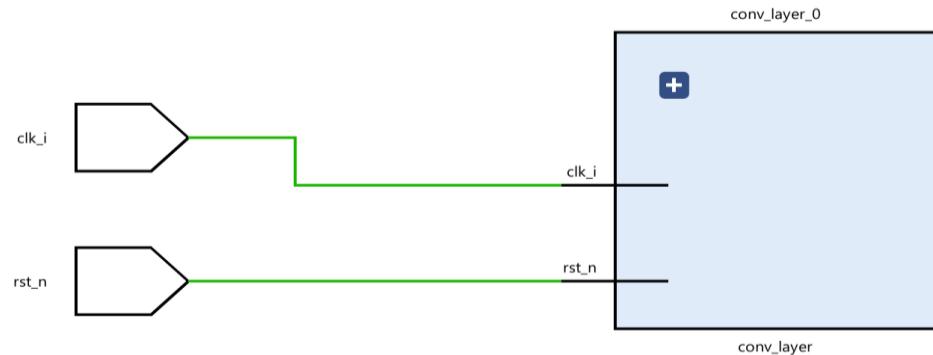


Figure 16:Top-level module RTL Schematic

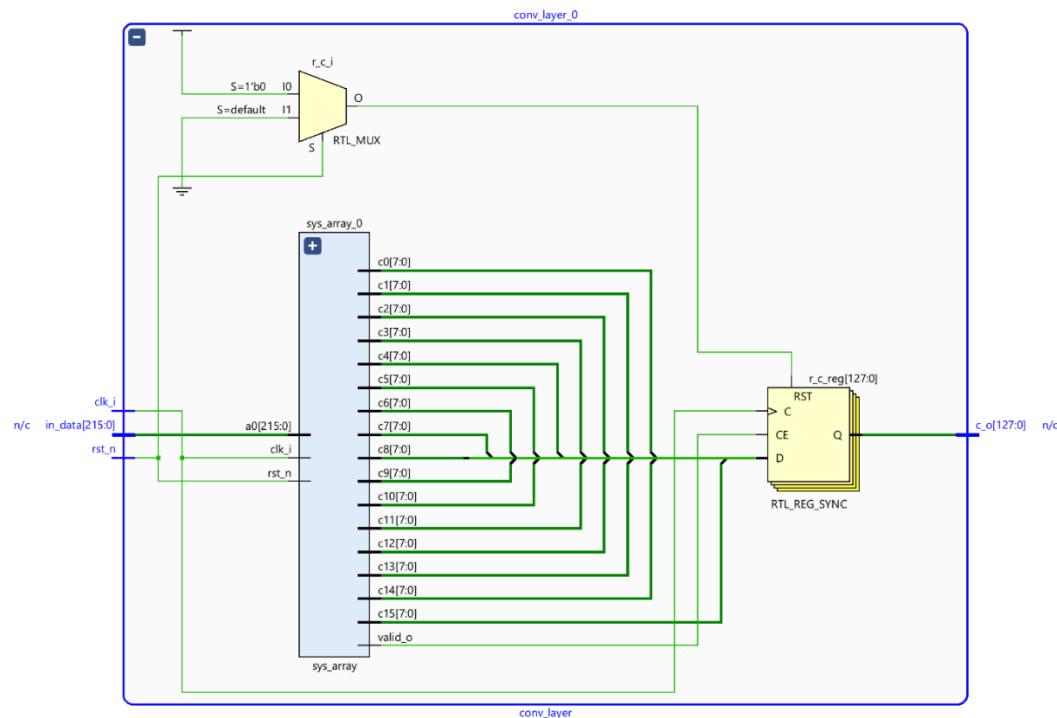


Figure 17: RTL schematic of internal architecture of conv_layer_0

The above figure 17 RTL schematic represents the internal architecture of the conv_layer_0 module, which is a key component of the proposed CNN accelerator. This module performs the convolution operation by receiving input data (in_data [215:0]) and distributing it to a systolic array block labeled sys_array_0. The systolic array processes the input data in a structured and parallel manner, producing

intermediate outputs through columns labeled c0 to c15. These outputs are combined and directed into a 128-bit register block (`r_c_reg [127:0]`) for synchronization and storage. The result is then forwarded to the output port `c_o [127:0]`. The design also includes control logic such as a resettable multiplexer (`RTL_MUX`) and a synchronous register (`RTL_REG_SYNC`) to manage data flow and ensure timing correctness. Clock (`clk_i`) and reset (`rst_n`) signals are globally distributed, maintaining proper sequencing and initialization. This schematic highlights the organized and parallel nature of the convolution layer, emphasizing efficiency and high throughput within the CNN hardware accelerator.

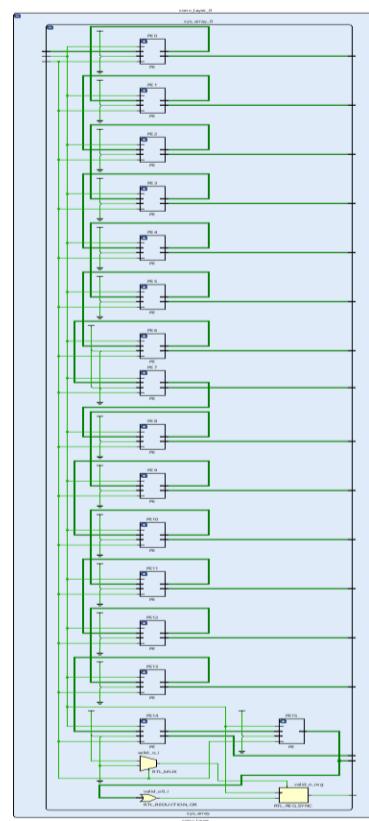


Figure 18: RTL schematic of the systolic array

The above figure 18 illustrates the RTL schematic of the systolic array block (`sys_array_0`) within the `conv_layer_0` module of the CNN accelerator. This array consists of 16 Processing Elements (PEs), each labeled from PE 0 to PE 15, arranged in a vertically cascading structure. These PEs are responsible for performing the core convolution computations by receiving input data and weights, processing them through MAC operations, and forwarding the results through interconnected data

paths. The green lines represent signal interconnections between PEs, showing how data flows sequentially and concurrently across the array. Toward the bottom, additional logic elements such as a multiplexer (RTL_MUX), control register (RTL_REG_SYNC), and a validation unit (RTL_REDUCTION_OR) manage the control signals and ensure that the outputs are synchronized and flagged as valid when processing is complete. This highly structured layout enables parallel processing, efficient data reuse, and high-speed computation, which are critical for accelerating convolution operations in CNNs on hardware platforms like FPGAs or ASICs.

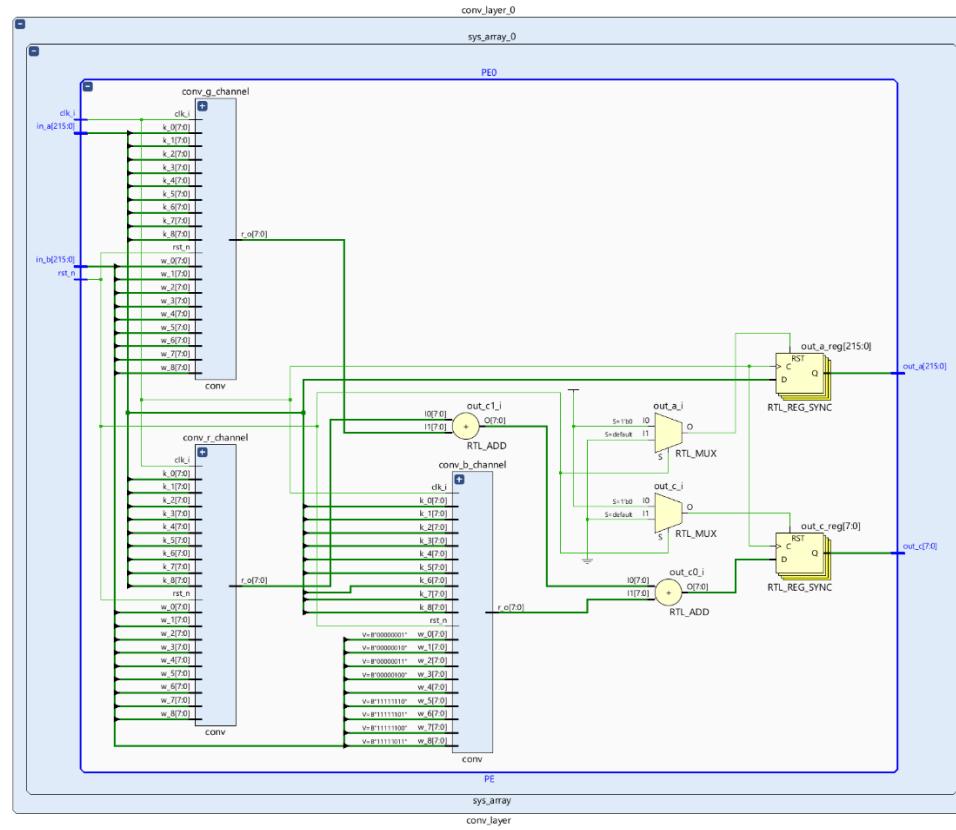


Figure 19: RTL schematic of processing element (PE)

The figure 19 displays the RTL schematic of PE0, a single Processing Element within the systolic array (sys_array_0) of the conv_layer_0 module. PE0 is composed of three parallel convolutional sub-modules: `conv_r_channel`, `conv_g_channel`, and `conv_b_channel`, each responsible for processing the red, green, and blue input channels respectively. Each convolution unit receives pixel values (`k_`) and weights (`w_`) as 8-bit inputs, processes them, and generates intermediate results (`r_o`).

These results are summed using two RTL_ADD blocks to combine the outputs of all three channels. The final result is temporarily stored in registers such as out_a_reg and out_c_reg, using RTL_REG_SYNC blocks that synchronize the output with the system clock (clk_i) and reset signal (rst_n). Multiplexers (RTL_MUX) are also used for selecting between incoming and computed data paths. The combined output of PE0 is directed to the output ports out_a [215:0] and out_c [7:0], contributing to the overall feature map generation in the CNN accelerator. This modular design enables parallel channel-wise processing, improving computation efficiency and making it suitable for high-speed image processing in hardware-based convolutional neural networks.

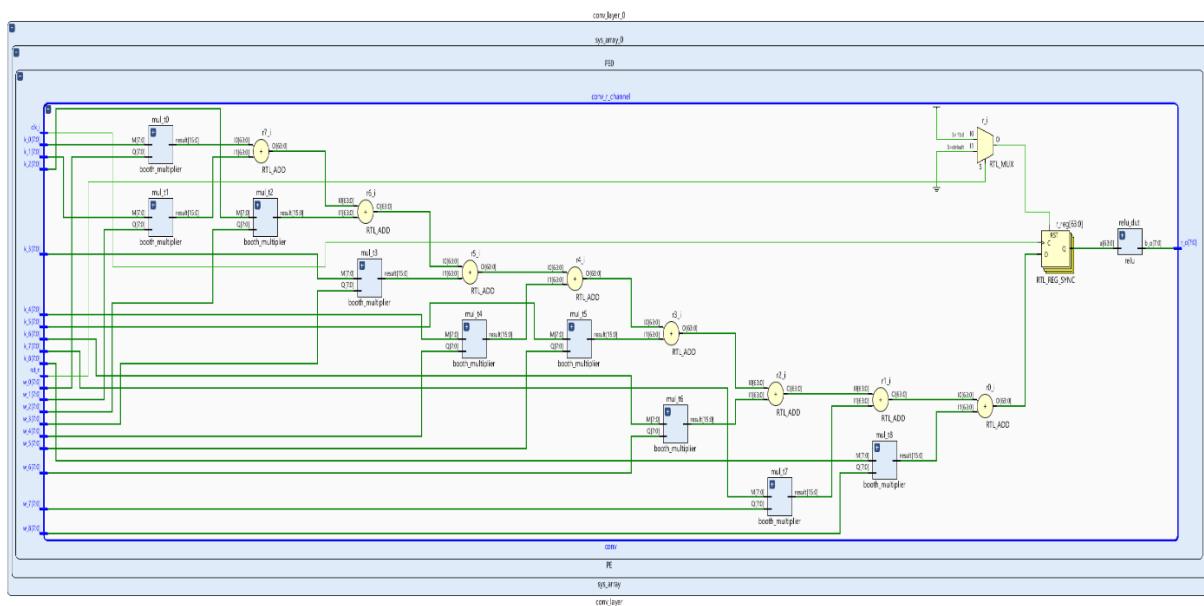


Figure 20: RTL schematic of the conv_g_channel block with PE

The image 20 above shows the internal RTL schematic of the conv_g_channel block within PE0 of the systolic array. This module performs the convolution operation for the green channel of the input image. It takes 8 kernel values (k [7:0]) and 9 pixel weights (w [8:0]) as inputs and performs nine parallel multiplications using Booth multipliers, which are optimized for high-speed, signed arithmetic. Each multiplier's output is 16 bits wide and is passed through a chain of RTL_ADD blocks, forming a multi-stage adder tree that accumulates all partial products to generate the final convolution result. The output of this accumulation is stored in a 64-bit register (r_reg) via a RTL_REG_SYNC block, which synchronizes the result with the system clock and reset signals. Before producing the final output (r_o [7:0]), the data is processed through a ReLU activation function that has been

implemented in the `relu_dut` block, ensuring that all negative values are clipped to zero. This schematic highlights the modular and pipelined structure of the convolution operation, enabling fast and efficient processing of image data at the hardware level.

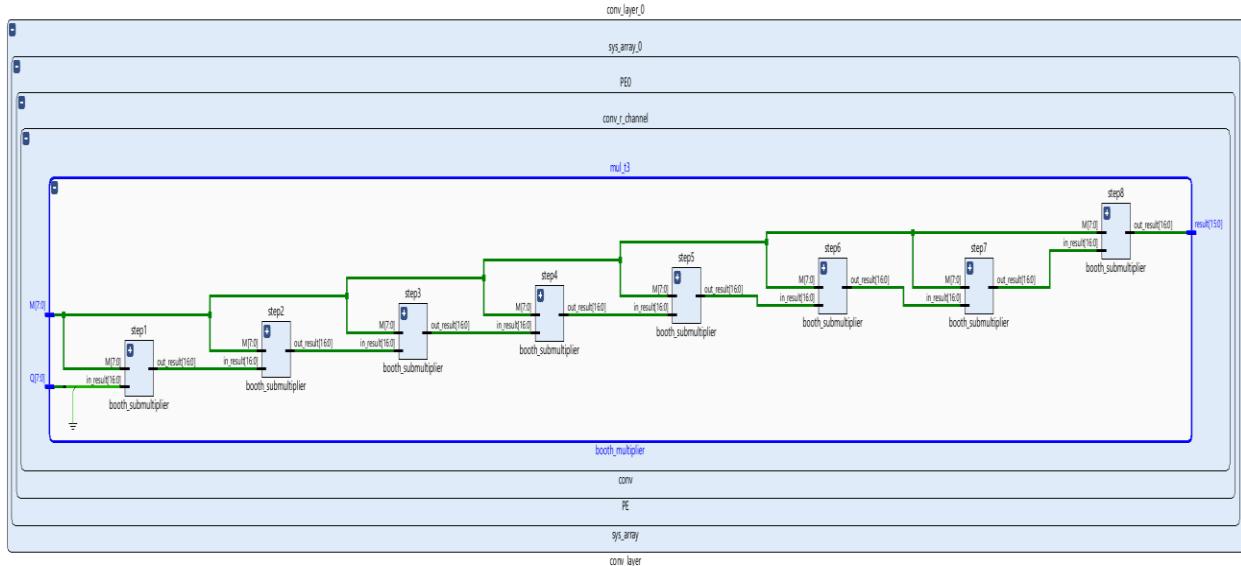


Figure 21: RTL schematic of Booth multiplier module

The figure 21 displays the RTL schematic of the Booth multiplier module used within the convolution blocks of the CNN accelerator. This module is designed to perform high-speed signed multiplication using the Modified Booth algorithm, which reduces the number of partial products and improves overall computational efficiency. The architecture consists of a sequence of eight `booth_submultiplier` blocks, labeled from `step1` to `step8`. Each submodule takes an 8-bit multiplicand (`M [7:0]`) and multiplier (`Q [7:0]`) and computes partial products step by step. These partial results (`out_result [15:0]`) are passed serially through each submodule, accumulating the final multiplication output at the end. The result of the final step is stored in the `result [15:0]` output signal. The pipeline-like structure of this Booth multiplier enables parallel processing of individual multiplication stages, significantly enhancing throughput while maintaining low latency. This optimized structure is crucial for accelerating convolution operations in hardware implementations of deep learning models like CNNs, particularly in power-efficient FPGA and ASIC designs.

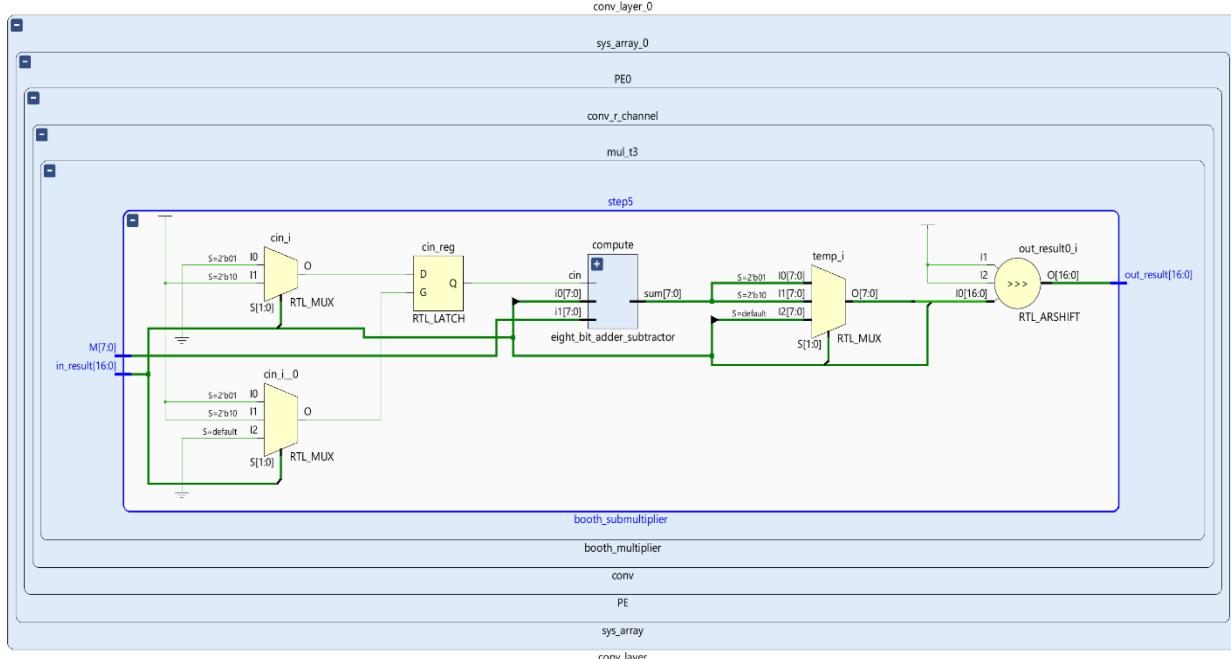


Figure 22: RTL schematic of internal structure of a single Booth submultiplier

The above figure 22 RTL schematic illustrates the internal structure of a single Booth submultiplier stage (step5), which is part of the multi-step pipeline used in the Booth multiplier module for efficient signed multiplication. In this design, the submodule accepts an 8-bit multiplicand ($M [7:0]$) and a 16-bit accumulated partial product ($in_result [16:0]$). It uses a combination of multiplexers (RTL_MUX) and a latch (RTL_LATCH) to decode the control logic based on the Booth encoding. The core computation is carried out by the `eight_bit_adder_subtractor` block, which adds or subtracts the multiplicand depending on the control inputs. The result is then selected through another multiplexer and passed to a right arithmetic shifter (RTL_ARSHIFT), which aligns the bits appropriately to form the final 16-bit output ($out_result [16:0]$). This pipelined stage helps in reducing the total number of partial products by encoding patterns in the multiplier, leading to faster and more area-efficient multiplication. Each such submultiplier stage builds on the previous stage's result, supporting a modular and scalable design suited for hardware-accelerated convolution in CNNs.

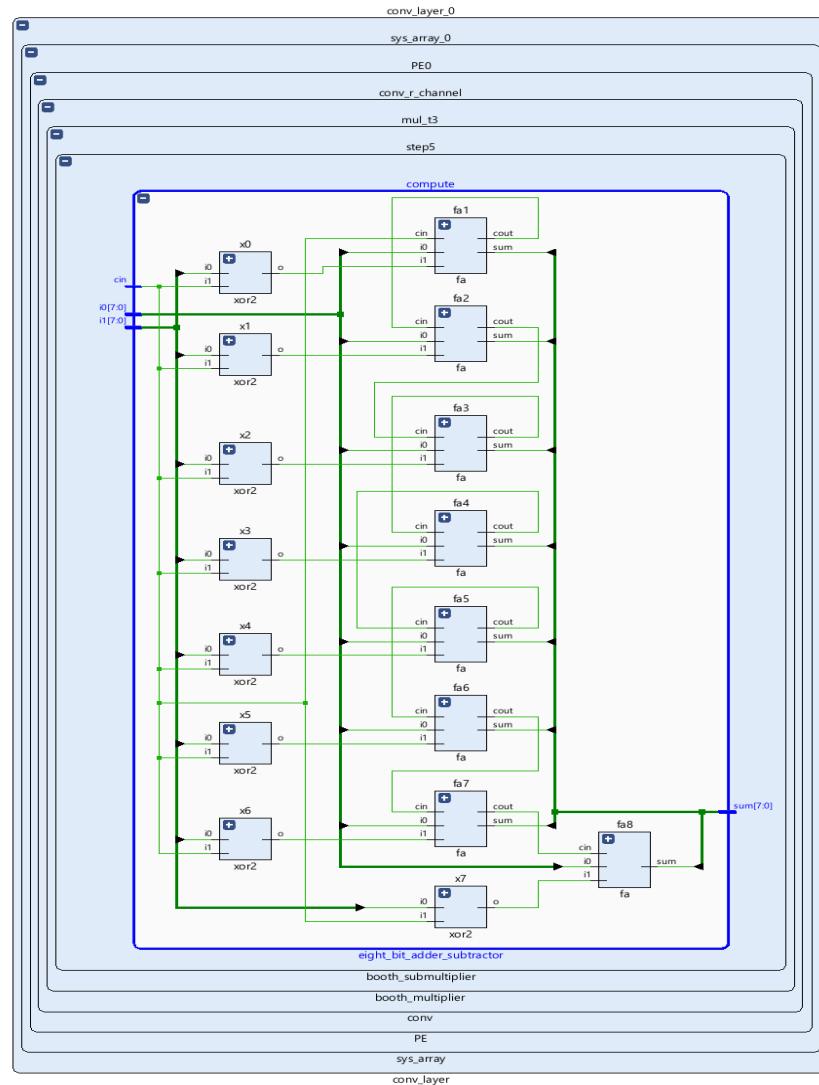


Figure 23: RTL schematic of 8-bit adder subtractor module

The figure 23 illustrates the internal RTL structure of the `eight_bit_adder_subtractor` module, which is a key arithmetic component within each `booth_submultiplier` stage of the Booth multiplier. This module performs the core signed addition or subtraction between two 8-bit binary numbers, $i0[7:0]$ and $i1[7:0]$, depending on the control signal cin . The design uses a sequence of eight full adders (fa0 to fa7) and one final full adder (fa8) to compute the result bit-by-bit along with carry propagation. In parallel, XOR gates ($x0$ to $x7$) are used to conditionally invert the input bits of $i1$ based on the control signal, effectively enabling two's complement subtraction when needed. The carry-in (cin) signal initiates the operation, supporting either addition or subtraction by controlling the initial bit logic. The

result of the operation is output as sum [7:0], which forms a part of the partial product accumulation pipeline in the Booth multiplication process. This modular and systematic approach allows efficient implementation of signed arithmetic operations, which are critical for performing multiply-and-accumulate (MAC) tasks in convolutional neural networks (CNNs) with minimal hardware resources and high-speed performance.

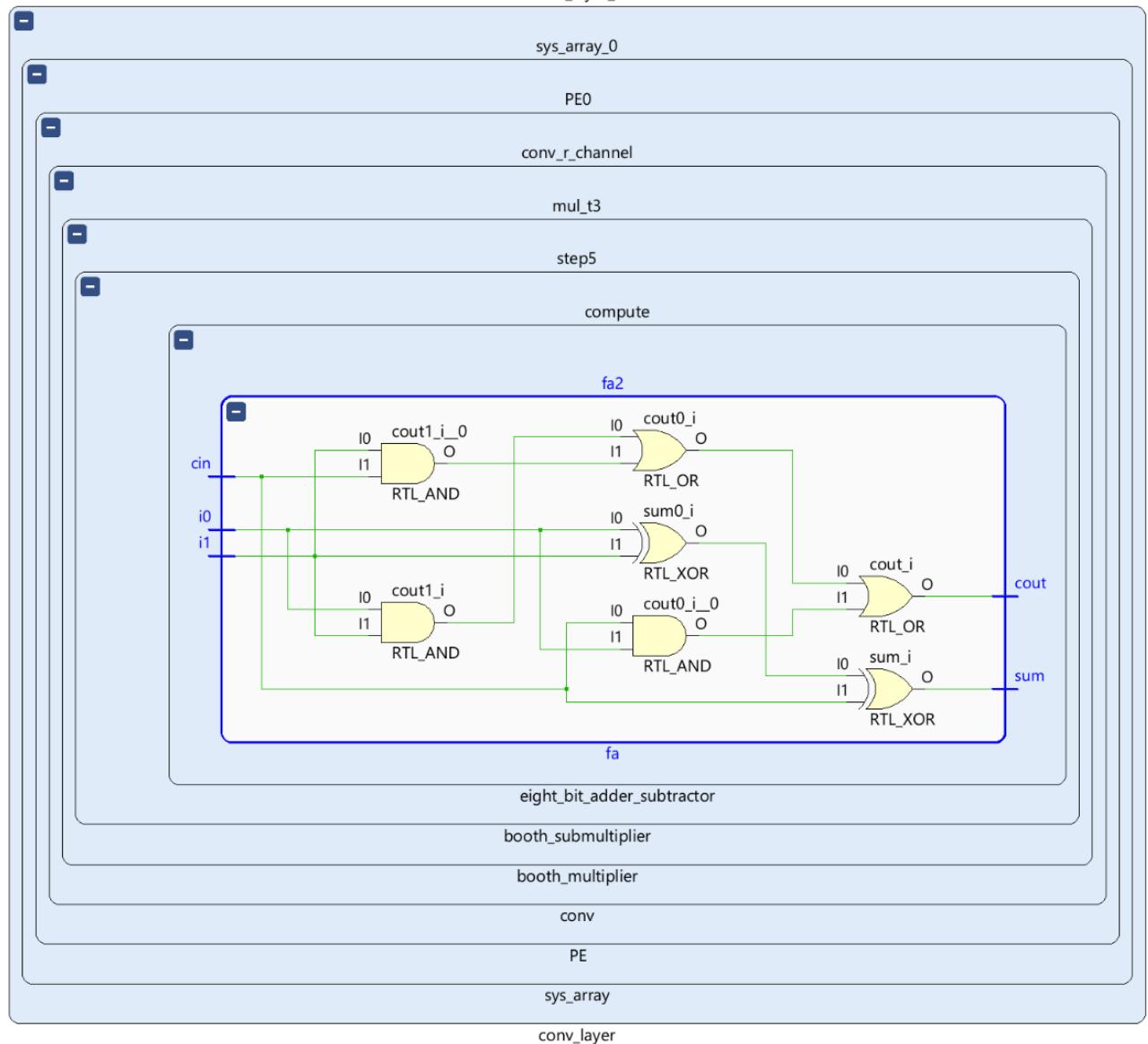


Figure 24: RTL schematic of 1-bit full adder(fa) module

The figure 24 shown above represents the 1-bit full adder (fa) module, which serves as the basic building block in the eight_bit_adder_subtractor used throughout the Booth multiplier. This full adder

takes three inputs: i0 and i1, representing the current bits to be added, and cin, which is the carry-in from the previous bit position. Internally, the design uses basic logic gates such as AND, OR, and XOR to compute both the sum output (sum) and the carry-out (cout). XOR gates are used to determine the sum of the input bits, while a combination of AND and OR gates is used to generate the carry logic. The modularity and simplicity of this 1-bit full adder design allow it to be replicated eight times in the 8-bit adder, forming the backbone of signed arithmetic operations within the convolution blocks of the CNN accelerator. This low-level structure is crucial for supporting fast and accurate multiply-and-accumulate functions within the system.

4.3.1 DIAGNOSIS

From the overall functional simulation and RTL schematic analysis, it is shown that proposed CNN accelerator architecture operates correctly and efficiently at register-transfer level. The functional simulation confirms accurate data flow, synchronization, and control signal handling across all modules, including the convolution layers, Booth multipliers, and activation functions. The RTL schematics clearly illustrate a modular and hierarchical design, enabling parallel processing and reusability of components. Together, these results validate the functional correctness and structural integrity of the hardware design, making it better-suited for high-performance and low-power image processing applications.

4.4 LOGIC SYNTHESIS

In the next step of the design process, logic synthesis is carried out using Synopsys DC, which converts the RTL design into a corresponding gate-level representation known as a netlist. This process maps the design to standard logic gates from a specific technology library. In this case, a 45nm NAND gate library allowing for real-world implementation. The primary goal of synthesis is to optimize the design with respect to area utilization, power efficiency, and timing performance.

4.4.1 AREA OPTIMIZATION

The figure 25 shows the area report generated from Synopsys Design Compiler, synthesized using the Nand gate 45nm Open Cell Library. The design includes a total of 14,745 cells, with 14,093 combinational and 632 sequential elements. A significant number of buffer/inverter cells (3,043) are used to support signal integrity and timing. The total cell area amounts to approximately 20,375 μm^2 ,

with the combinational area being around $17,919 \mu\text{m}^2$ and the non-combinational (sequential) area about $2,456 \mu\text{m}^2$. The net interconnect and total physical area are marked as undefined, possibly due to zero wire load modeling at this stage. This report helps evaluate the silicon footprint of the design before moving into placement and routing.

```
*****
Report : area
Design : PE
Version: S-2021.06-SP5-1
Date   : Thu Apr 10 20:23:02 2025
*****
Library(s) Used:
    NangateOpenCellLibrary (File: /synopsys/Nangate_FreePDK45/
NangateOpenCellLibrary_PDKv1_3_v2010_12/Front_End/Liberty/CCS/NangateOpenCellLibrary.db)

Number of ports:                      658
Number of nets:                        15600
Number of cells:                       14745
Number of combinational cells:         14093
Number of sequential cells:            632
Number of macros/black boxes:          0
Number of buf/inv:                     3043
Number of references:                  24

Combinational area:                   17919.355902
Buf/Inv area:                         1621.536016
Noncombinational area:                2455.711950
Macro/Black Box area:                 0.000000
Net Interconnect area:                undefined (Wire load has zero net area)

Total cell area:                      20375.067852
Total area:                           undefined
1
```

Figure 25: Area optimization report

4.4.2 POWER OPTIMIZATION

Figure 26 presents the power analysis report generated by Synopsys Design Compiler, detailing the distribution of power consumption across various components. The total dynamic power is reported as 8.6975 mW, consisting of 4.7036 mW of internal power and 3.9939 mW of net switching power, indicating a near-even distribution. Additionally, the cell leakage power is 1.5332 mW, contributing to the overall total power of approximately 10.231 mW. Among power groups, the combinational logic dominates, consuming about 9.1 mW, which accounts for over 89% of the total power. The register and sequential elements consume less, reflecting efficient pipelining and storage logic. This report helps evaluate the energy efficiency of the design and is critical for low-power CNN accelerator applications.

Cell Internal Power	=	4.7036 mW	(54%)
Net Switching Power	=	3.9939 mW	(46%)
Total Dynamic Power	=	8.6975 mW	(100%)
Cell Leakage Power	=	1.5332 mW	

Power Group (%) Attrs	Internal Power	Switching Power	Leakage Power	Total Power
io_pad (0.00%)	0.0000	0.0000	0.0000	0.0000
memory (0.00%)	0.0000	0.0000	0.0000	0.0000
black_box (0.00%)	0.0000	0.0000	0.0000	0.0000
clock_network (0.00%)	0.0000	0.0000	0.0000	0.0000
register (9.55%)	834.6434	27.9693	1.1429e+05	976.9024
sequential (1.34%)	42.8361	63.5540	3.0433e+04	136.8233
combinational (89.11%)	3.8261e+03	3.9024e+03	1.3885e+06	9.1170e+03
Total	4.7036e+03 uW	3.9939e+03 uW	1.5332e+06 nW	1.0231e+04 uW
1				

Figure 26: Power Optimization report

4.4.3 TIMING ANALYSIS

The image illustrates the timing analysis report produced by Synopsys Design Compiler, verifying that the design successfully adheres to the specified timing constraints. The report indicates a data arrival time of 3.77 ns, and a data required time of 3.78 ns, with a setup time margin (slack) of 0.00 ns, meaning the design has met timing (MET). This analysis also accounts for clock uncertainty of -0.20 ns and a library setup time of -0.02 ns. Since the slack is zero, the circuit operates at the target clock frequency without violations, ensuring timing correctness and reliable performance for the CNN accelerator when synthesized at a clock period of 4.00 ns.

add_5_root_add_116_82/U1_14/CO (FA_X1)	0.09	2.91	f
add_5_root_add_116_82/U1_15/CO (FA_X1)	0.06	2.97	f
add_5_root_add_116_82/U1_16/CO (FA_X1)	0.06	3.03	f
add_5_root_add_116_82/U1_17/CO (FA_X1)	0.06	3.09	f
add_5_root_add_116_82/U1_18/S (FA_X1)	0.11	3.19	r
U1_182/CO (FA_X1)	0.07	3.27	r
U1_191/CO (FA_X1)	0.05	3.32	r
U1_201/CO (FA_X1)	0.05	3.38	r
U1_211/S (FA_X1)	0.09	3.47	f
U1_213/CO (FA_X1)	0.08	3.55	f
U1_222/CO (FA_X1)	0.06	3.61	f
U1_231/S (FA_X1)	0.10	3.71	r
U16994/ZN (AND2_X1)	0.05	3.76	r
r_reg[24]2/D (DFF_X1)	0.02	3.77	r
data arrival time		3.77	
clock clk (rise edge)	4.00	4.00	
clock network delay (ideal)	0.00	4.00	
clock uncertainty	-0.20	3.80	
r_reg[24]2/CK (DFF_X1)	0.00	3.80	
library setup time	-0.02	3.78	
data required time		3.78	
data arrival time		-3.77	
slack (MET)	0.00		

1

Figure 27: Timing Analysis report

4.5 PHYSICAL DESIGN

The physical design flow is a critical stage in the ASIC implementation process that transforms the synthesized gate-level netlist into an actual chip layout ready for fabrication. It involves placing the logic cells on a silicon die and routing the metal connections between them to meet performance, area, and power requirements. The physical design process involves multiple key stages, as illustrated in Figure 28, including floorplanning, placement, clock tree synthesis (CTS), routing, and physical verification. Each of these steps is essential for achieving optimal timing performance, power efficiency, and effective use of chip area throughout the layout design. Tools like Cadence Innovus is used in this stage to automate and fine-tune the layout process. A successful physical design ensures that the final chip meets the necessary specifications and operates without errors after fabrication.

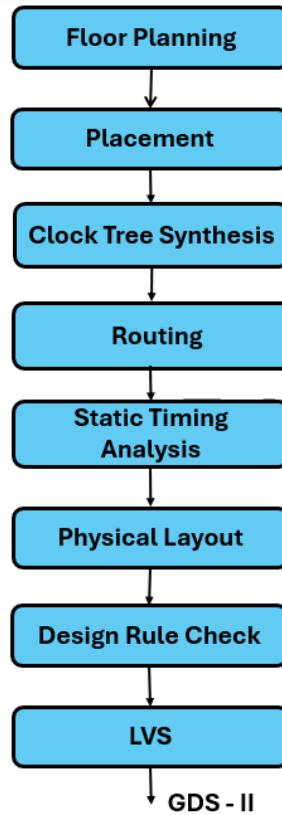


Figure 28: Physical design flow [25]

The above figure represents Physical design flow which is implemented in Cadence Innovus. Here I have used Cadence Innovus with 45nm NAND gate library. This tool is used to implement floorplanning, placement, routing and LVS and final chip tape out.

4.5.1 DESIGN IMPORTING

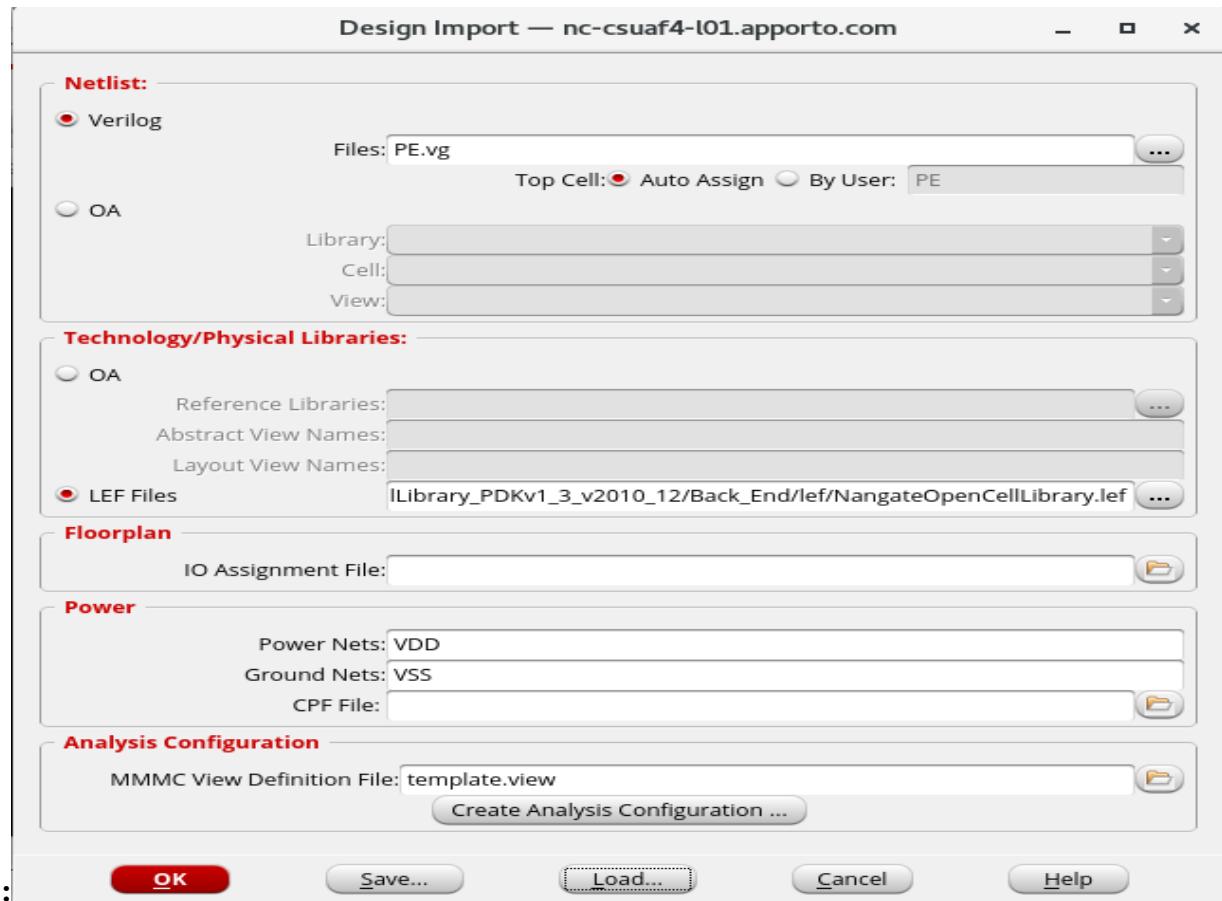


Figure 29: Importing NETLIST and LEF file

The above figure 29 shows the Design Import window in Cadence Innovus, where the synthesized netlist and technology libraries are loaded to begin the physical design flow. In this setup, the Verilog netlist file (PE.vg) is imported as the top-level design, with auto assignment of the top cell enabled. For physical library information, the LEF file (NangateOpenCellLibrary.lef) is selected from the 45nm Nangate PDK, providing the necessary cell dimensions and pin layouts for floorplanning and placement. Power connections are defined with VDD as the power net and VSS as the ground net. The MMMC view file (template. view) is also provided to define the timing corners and analysis conditions. This step marks the beginning of place and route in Innovus, setting up the environment to translate the gate-level netlist into a finalized physical chip layout.

4.5.2 FLOOR PLANNING

Floor planning is a vital phase in the physical design process where the overall layout of the chip is organized before placement and routing. It involves defining the core area, aspect ratio, location of input/output (I/O) pins, and the placement of major functional blocks such as memory, logic, and clock sources. Good floorplanning ensures efficient area utilization, minimizes routing congestion, and helps meet timing, power, and performance goals. It also defines the power planning strategy, including the placement of power and ground rails. In this project, floorplanning was guided by the LEF file from the Nangate 45nm library, and the layout was structured to support the parallel architecture of the CNN accelerator for better performance and lower power usage.



Figure 30: Core Utilization Setup and Detailed Floorplan

The above figure 30 show the floorplanning step in Cadence Innovus, where the physical layout of the chip's core and die are defined. In the first image, the design dimensions are configured using an aspect ratio of approximately 1:1, ensuring a square-shaped core layout for better symmetry and routing balance. The core utilization is set to 1, meaning 100% of the core area is intended for standard cell placement, with 10-micron margins defined between the core and I/O boundaries on all four sides.

4.5.3 ADDING GLOBAL NETS

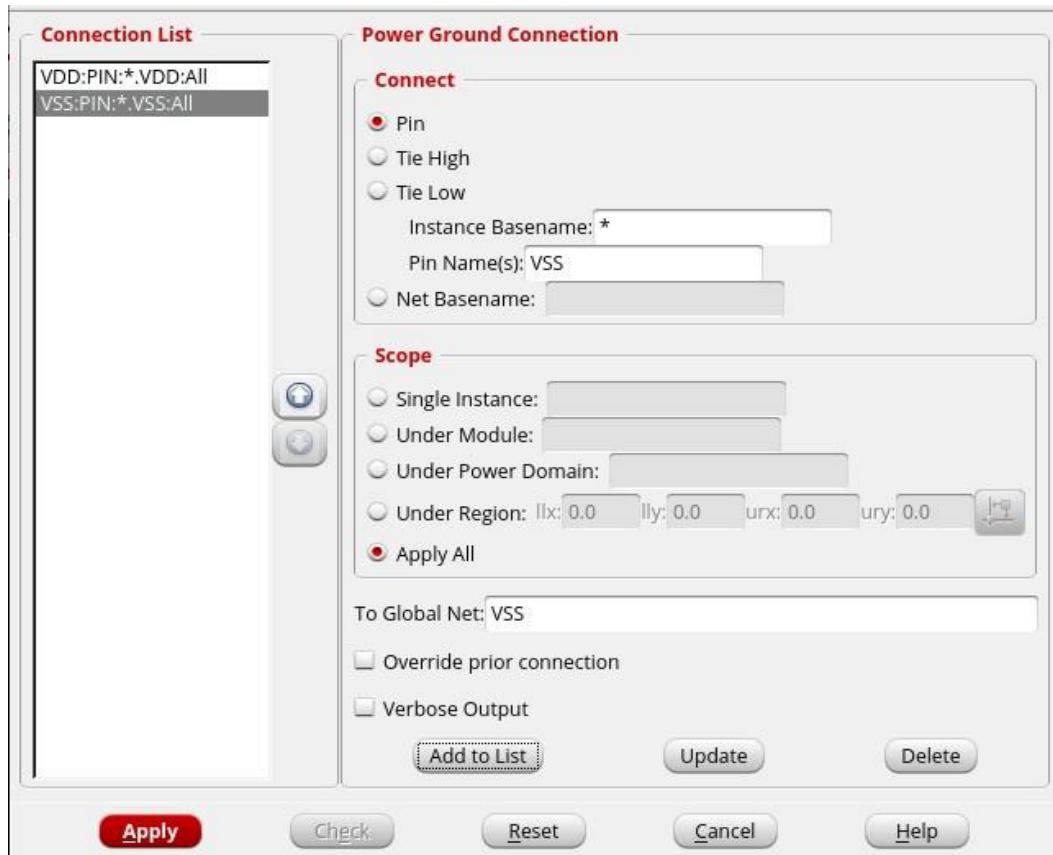


Figure 31: Connecting Global nets

The above figure 31 shows the power-ground net connection setup in Cadence Innovus, where global nets such as VDD and VSS are linked to their respective power and ground terminals of all standard cells. The Pin Names are specified as VDD and VSS. The scope is set to "Apply All", meaning the connections will be applied throughout the entire design. This step ensures that every logic element in the layout has proper power and ground connectivity, which is essential for correct functionality during fabrication. Making these connections early in the physical design phase is crucial for enabling reliable power distribution and avoiding floating or unconnected pins.

4.5.4 ADDING POWER RINGS

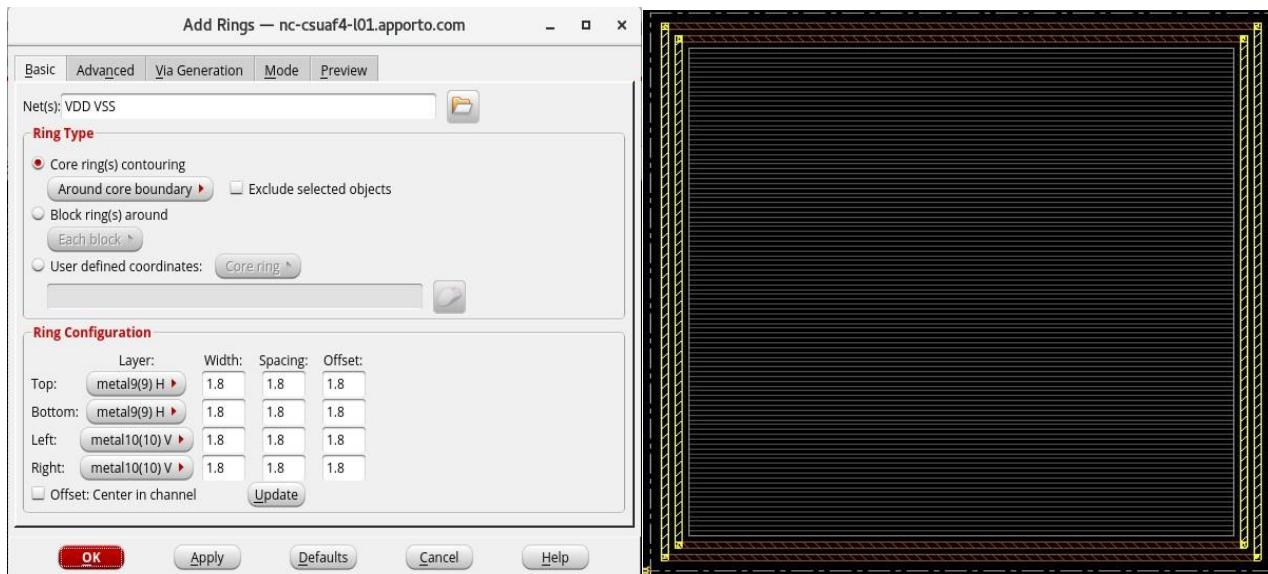


Figure 32: Power rings Configuration and Power rings

The figure 32 displays the Add Rings configuration, which is used to create power and ground rings around the core area of the chip. In this setup, VDD and VSS nets are selected to form the rings, ensuring a robust power distribution network. The ring type is set to "Core ring(s) contouring", which places the power rings tightly around the core boundary. The ring configuration uses metal layers 9 and 10, where metal9 handles the horizontal top and bottom rings, and metal10 handles the vertical left and right rings. Each ring has a width, spacing, and offset of 1.8 microns, providing uniform and symmetrical coverage. These rings play a critical role in distributing power evenly across the chip and reducing voltage drop, especially in high-performance designs like CNN accelerators.

4.5.5 SPECIAL ROUTING

The figure 33 shows the Special Routing (SRoute) configuration, which is used to route power (VDD) and ground (VSS) nets across the chip. In this setup, routing includes connections for block pins, pad rings, pad pins, and floating stripes, ensuring complete power coverage. The routing is allowed to span from metal1 (bottom layer) up to metal10 (top layer), enabling vertical distribution of power with support for layer changes. This ensures that every standard cell and macro receives reliable power and ground connections. Special routing is a critical step after adding power rings, as it physically

connects all power structures and reinforces the robustness and reliability of the power distribution network throughout the chip.

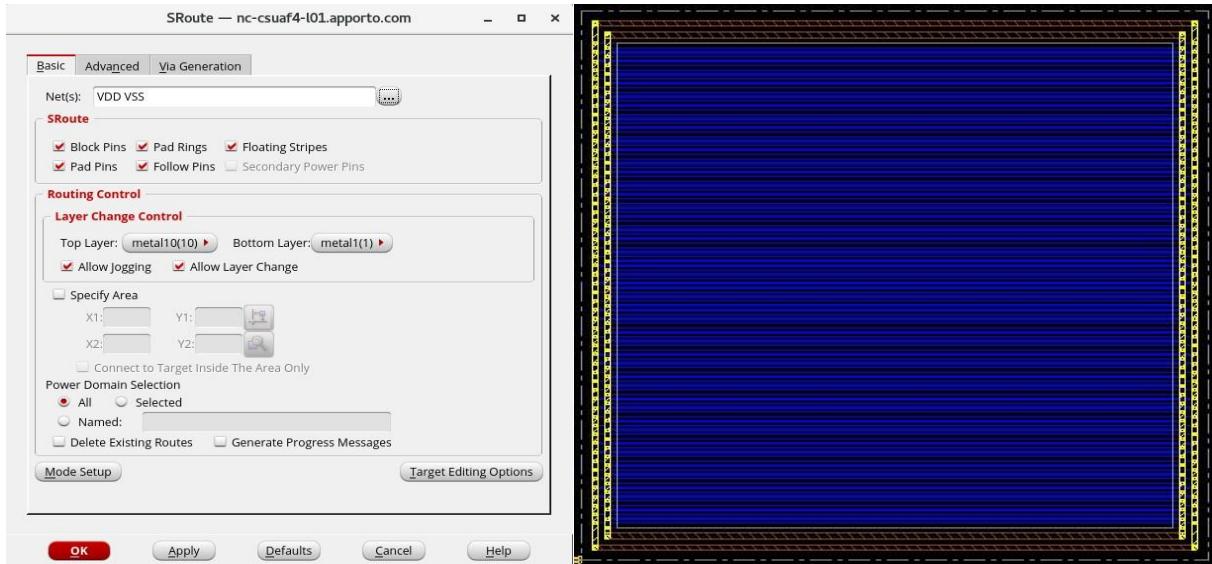


Figure 33: Special routing setup and Nets after routing

4.5.6 ADDING POWER STRIPES

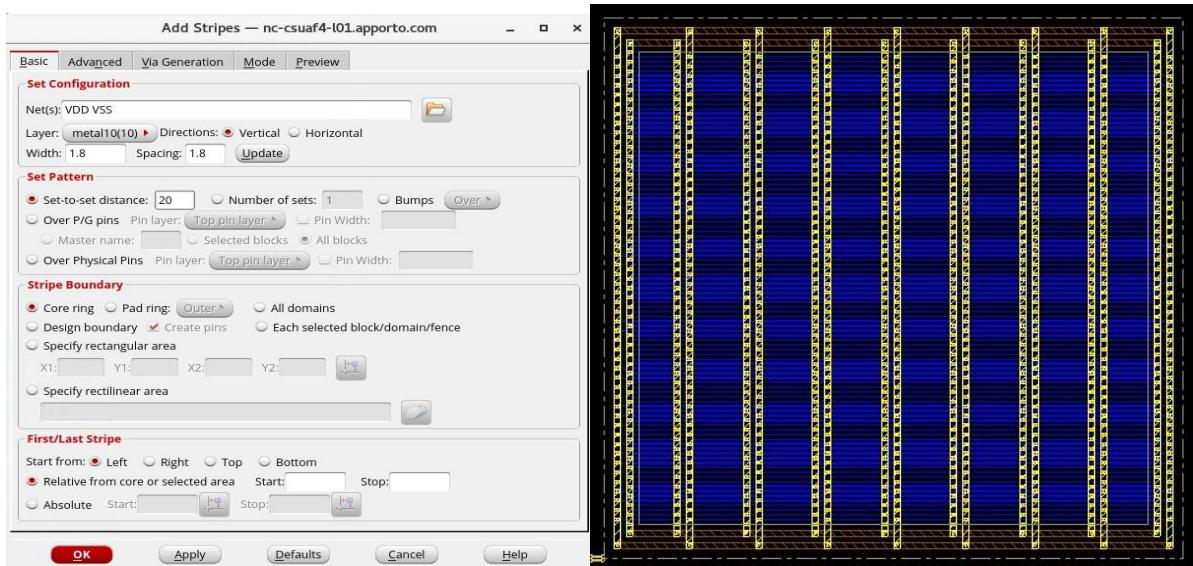


Figure 34: Power stripes

The figure 34 shows the Add Stripes setup window, where vertical power stripes for the VDD and VSS nets are defined to strengthen the power distribution network within the chip core. In this

configuration, metal10 is used for the stripes with a width and spacing of 1.8 microns, and the stripes are placed at a regular set-to-set distance of 20 microns across the core area. These stripes connect to the existing power rings and ensure that power and ground are evenly delivered to all standard cells and functional blocks. Placing power stripes improves IR drop performance, reduces power congestion, and enhances electromigration reliability, which is essential for stable operation of the CNN accelerator under high computational loads.

4.5.7 PLACEMENT

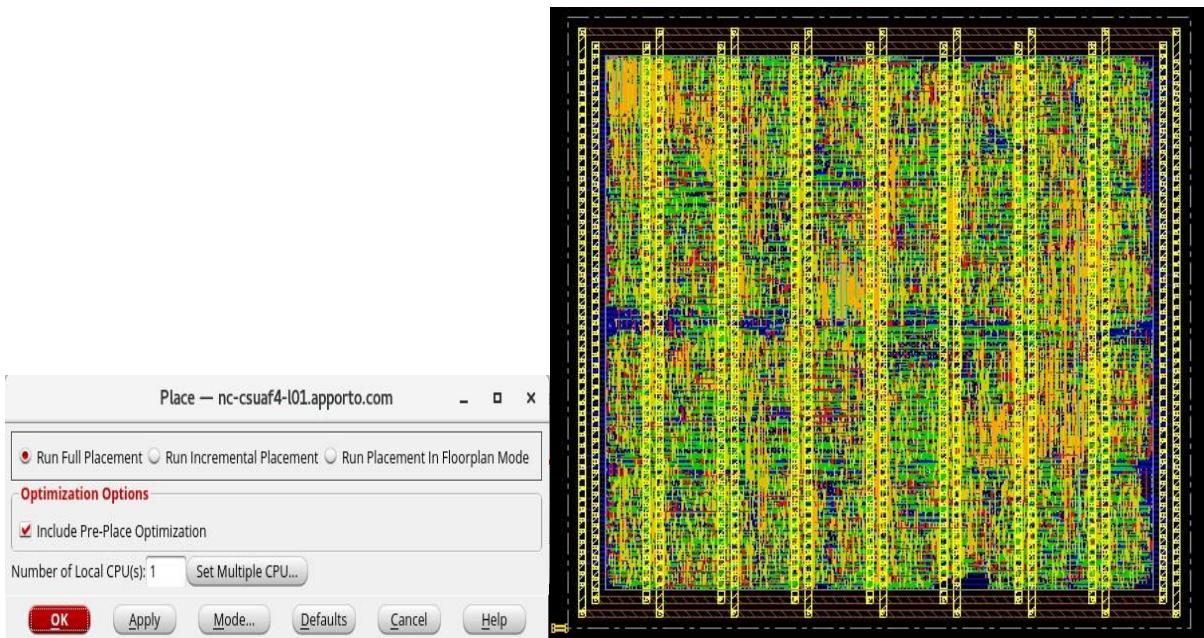


Figure 35: Standard cell placement

Figure 35 illustrates the standard cells placement in the physical design stage using. In this step, I make sure all synthesized logic gates and standard cells are arranged within the defined core area of the chip layout. The placement tool automatically positions the cells to minimize wirelength, reduce routing congestion, and satisfy timing constraints. The vertical yellow lines in the layout represent power stripes, which distribute VDD and VSS across the chip. The colorful regions indicate densely packed logic cells, with spacing adjusted for thermal and routing efficiency. Proper placement ensures that the design is physically realizable, optimizes performance, and prepares the design for subsequent stages.

4.5.8 CLOCK TREE SYNTHESIS

Clock Tree Synthesis (CTS) represents a fundamental stage in the physical design flow where a balanced tree structure is created to distribute the clock signal uniformly to all sequential elements, such as flip-flops and registers. The primary objective of CTS is to reduce clock skew and insertion delay across the design, ensuring that all parts of the circuit receive the clock signal at nearly the same time. During this step, buffers and inverters are automatically inserted to drive the clock signal efficiently and meet timing requirements. A well-designed clock tree improves the timing stability and performance of the overall chip, which is especially important for high-speed designs like CNN accelerators.

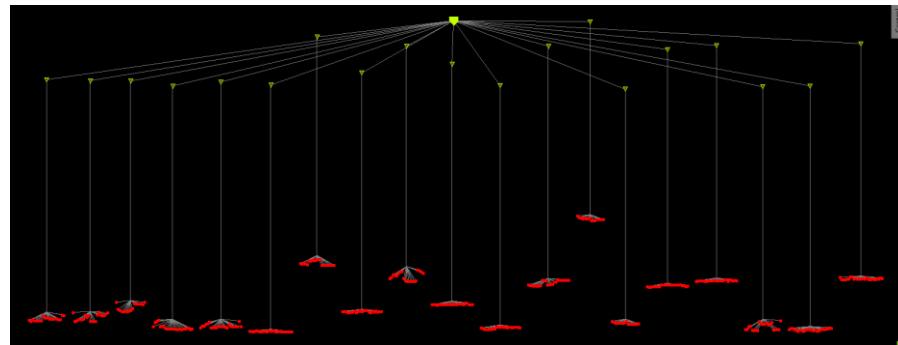


Figure 36: Clock Tree Synthesis

I have used 3 main commands for clock tree optimization. The following command is used during Clock Tree Synthesis (CTS) to specify which buffer cells can be used for building the clock tree. It helps the tool choose appropriate buffering options to optimize clock delay, skew, and power consumption.

```
set_ccopt_property buffer_cells {CLKBUF_X1 CLKBUF_X2 CLKBUF_X3}
```

The command `create_ccopt_clock_tree_spec` is used in Cadence Innovus to generate the Clock Tree Optimization (CCOpt) specification file, which contains the settings, constraints, and cell usage rules for clock tree synthesis. This spec file guides the tool in building an optimized, low-skew clock tree, helping to meet timing and power goals efficiently.

```
create_ccopt_clock_tree_spec
```

The command `ccopt_design` in Cadence Innovus is used to run the Clock Tree Optimization (CCOpt) process based on the previously defined clock tree specification. It automatically inserts buffers, balances the tree, and it fine-tunes delays to reduce clock skew and insertion delay, ensuring to ensure that the clock signal is delivered uniformly and consistently across the entire design.

`ccopt_design`

4.5.9 INSERTING I/O FILLERS

Inserting filler cells is an essential stage in the physical design flow that guarantees electrical continuity and layout completeness after placement and routing are finished. Filler cells are small, non-functional standard cells inserted into the gaps between logic cells to maintain proper well and n-well connections, and to fill empty space on the chip for manufacturing reliability. These cells also help align the power rails and prevent issues like metal disconnection or lithography errors during fabrication. Although they don't contribute to the logical behavior of the design, filler cells are essential for ensuring design rule compliance and creating a manufacturable and stable ASIC layout.

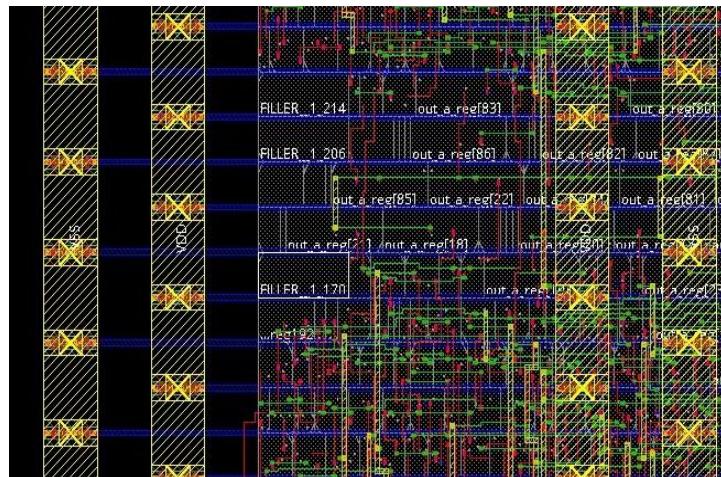


Figure 37: Filler Cells added on layout

4.5.10 VERIFYING DESIGN RULE CHECK

Design Rule Check (DRC) verification serves as the final and one of the most critical part in the physical design process where the layout is checked against the manufacturing rules to ensure it can be fabricated without errors. These rules include constraints on spacing, width, enclosure, and

alignment of metal layers, vias, and other features. Running DRC helps identify and fix violations such as shorts, spacing issues, or overlaps, which could lead to chip failure or poor yield. I have done DRC for 1000 violations.

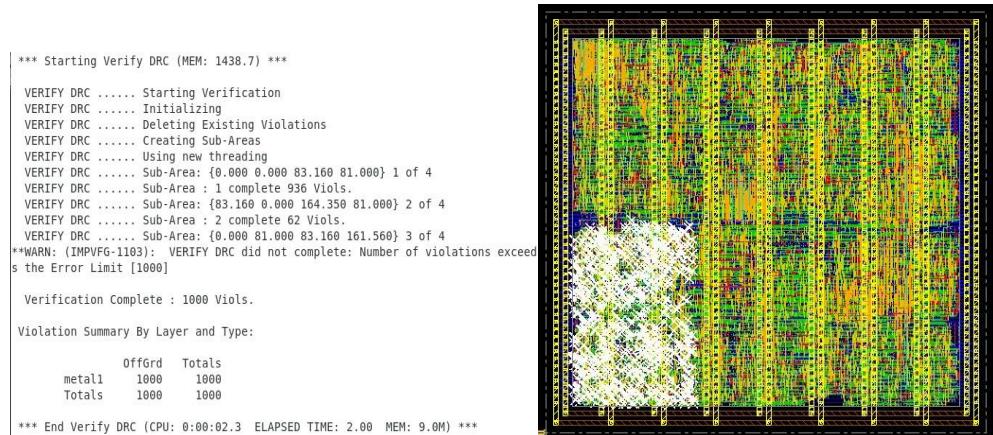


Figure 38: DRC Violations after placement

4.5.11 EARLY GLOBAL ROUTING AND NANO ROUTING

Early Global Routing and NanoRoute represent two critical stages in the routing phase of the physical design flow that help connect all cells and blocks with proper signal paths while optimizing for timing, congestion, and design rules.

Early Global Routing is performed after placement and clock tree synthesis to provide a rough estimation of wire paths and routing resources. It helps identify potential congestion hotspots, estimates net delays, and guides placement and buffering strategies before detailed routing begins. Although it doesn't produce final routes, it gives valuable feedback for improving timing and physical quality early in the flow.

NanoRoute, on the other hand, is the detailed router in Cadence Innovus that performs the actual final routing of all nets in the design. It ensures that all signal connections are made while strictly following design rules, timing constraints, and metal layer usage. NanoRoute creates the real metal paths that will be used in manufacturing and works to minimize wire length, via count, and delay, resulting in a fully connected, DRC-clean layout ready for signoff.

```
#CELL_VIEW PE,init has no DRC violation.  
#Total number of DRC violations = 0  
#   number of violations = 0  
#cpu time = 00:00:04, elapsed time = 00:00:04, memory = 1255.59 (MB), peak = 14.  
1.72 (MB)  
#CELL_VIEW PE,init has no DRC violation.  
#Total number of DRC violations = 0  
#Post Route wire spread is done.
```

Figure 39: post-routing violations check

4.5.12 FINAL CHIP LAYOUT

The final chip layout, shown in Figure 40, represents the complete physical implementation of the design, following all backend stages including logic synthesis, placement, clock tree synthesis, routing, and design rule checks. The layout encapsulates all critical design elements, including standard cells, interconnects, power rails, and clock distribution, tightly organized within the defined core area. Different metal layers and routing tracks are visually distinguished using color-coded patterns, showcasing the multi-layer routing used to handle signal, power, and ground connections efficiently. This layout was finalized using Cadence Innovus, ensuring full compliance with the 45nm design rules and optimizing the chip for low area, high performance, and minimal power consumption. Additionally, power-saving techniques like clock gating were applied to reduce dynamic power, highlighting the design's suitability for energy-efficient VLSI applications.

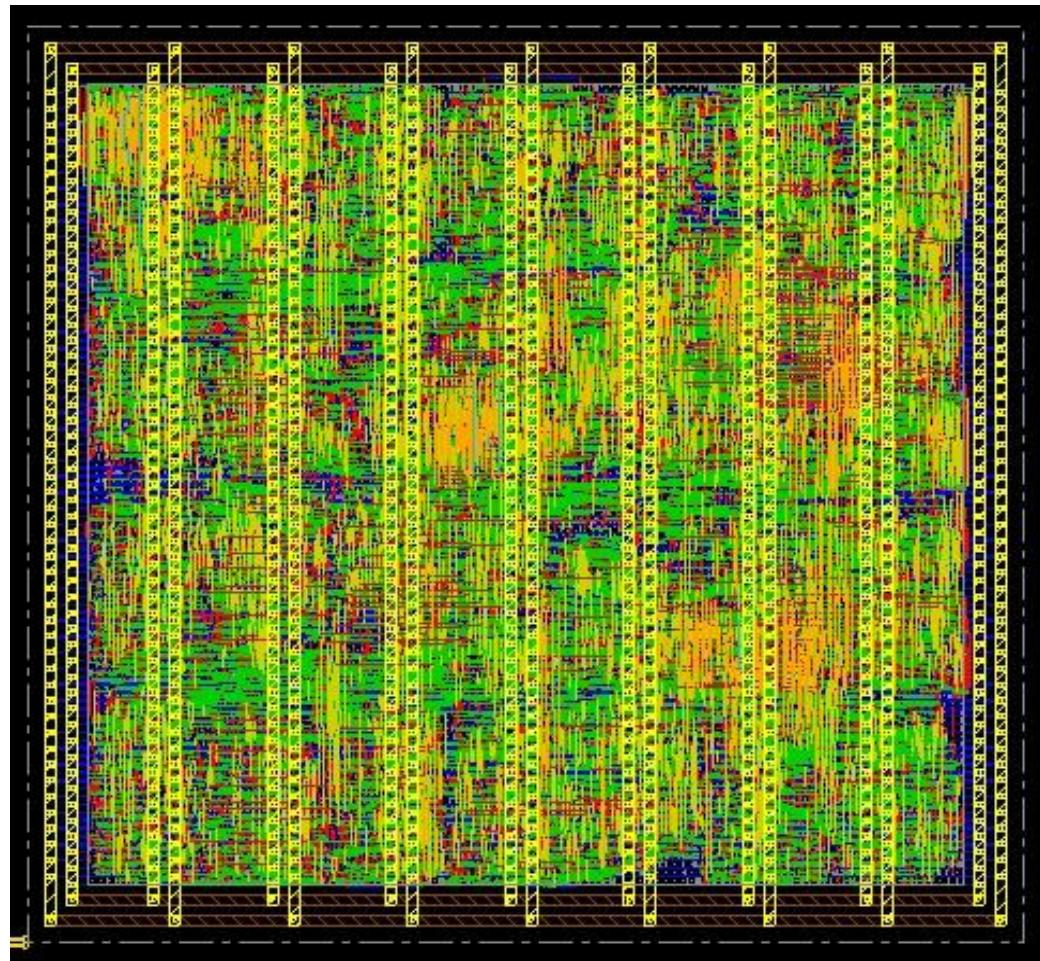


Figure 40: Final Chip Layout

4.5.13 GDS II STREAMOUT

The figure 41 shows the completion status of the GDSII stream out process, which confirms that the final layout has been successfully exported into the standard GDSII format, a critical step for fabrication. The message "Stream out is finished!" at the bottom indicates that all instances—such as standard cells, routing data, and metal layers—have been correctly processed. The count of 15,041 instances confirms that the design's physical content is included, and there are no additional elements like ports, nets, or metal fills reported separately. This GDSII file is now ready to be handed off to the foundry for mask generation and chip fabrication, marking the final stage of the backend flow.

Object	Count
Instances	15041
Ports/Pins	0
Nets	0
Via Instances	0
Special Nets	0
Via Instances	0
Metal Fills	0
Via Instances	0
Metal FillOPCs	0
Via Instances	0
Metal FillDRCs	0
Via Instances	0
Text	0
Blockages	0
Custom Text	0
Custom Box	0
Trim Metal	0
#####Streamout is finished!	

Figure 41: GDSII stream out

5. RESULTS AND ANALYSIS

5.1 VERIFICATION OF FUNCTIONALITY OF CNN ACCELERATOR

The following image demonstrates the verification of the CNN accelerator's functionality for object detection, performed using the MATLAB tool. In this step, the YOLOv3 algorithm—integrated with the Darknet-19 architecture—was used to detect objects such as cars and persons from real-world images. The accelerator's output was validated by comparing the bounding boxes and class predictions generated in hardware with the expected results obtained from MATLAB. As seen in the image, the model successfully identifies and labels both the car and the person with accurate bounding boxes, proving that the hardware CNN accelerator produces correct and reliable detection outputs. This validation ensures that the accelerator functions as intended and can be confidently used for real-time object detection applications in embedded systems.

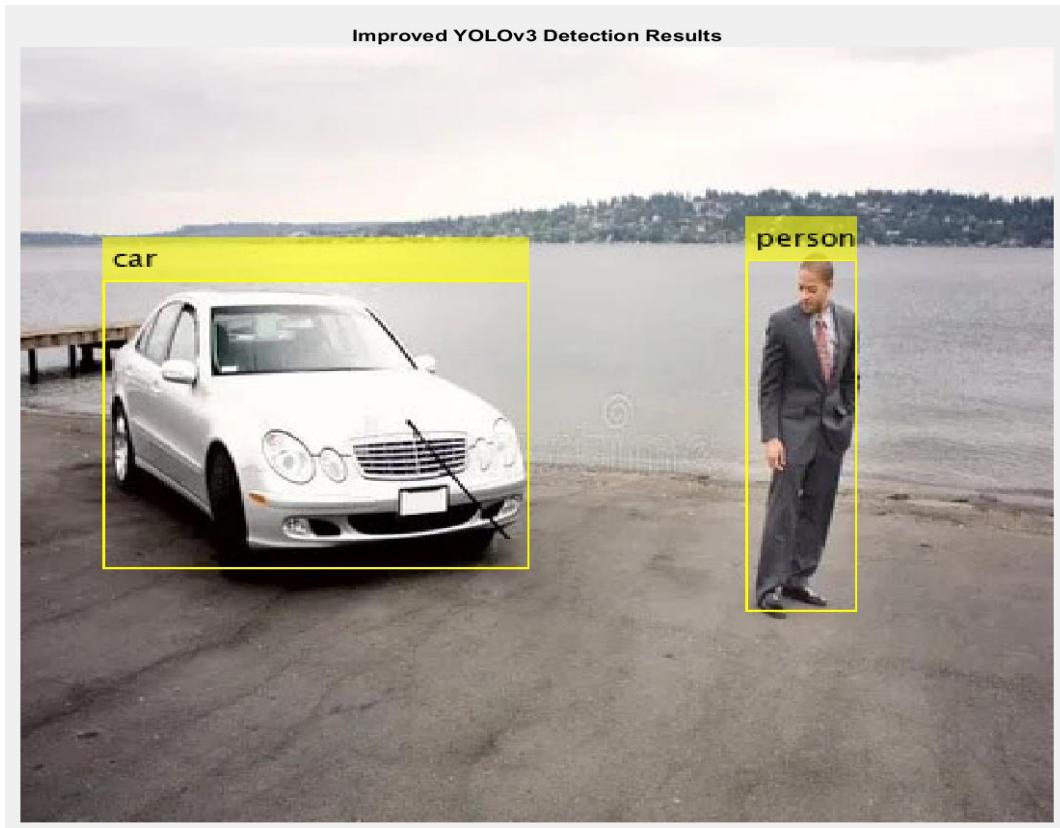


Figure 42: YOLOv3 detection results

5.2 POWER AND AREA ANALYSIS OF PROCESSING ELEMENT

Design	Area(μm^2)	Power(mW)
CNN accelerator based on Cyclic array [27]	29176	18.52
Vectorwise accelerator for CNN [28]	46172	15.98
Energy efficient CNN accelerator [5]	-	9.43
Proposed CNN accelerator using DARKNET-19	20375	8.69

Table 1: Comparison of area and power between existing approaches and proposed model

Based on the Table 1, the comparison offers a comprehensive evaluation of area and power consumption among various CNN hardware accelerators, including the proposed design using the DARKNET-19 architecture. The table includes three previously established accelerators: a cyclic array-based design, a vector wise accelerator, and a power-efficient CNN model. The cyclic array-based CNN accelerator reports an area of 29,176 μm^2 and power consumption of 18.52 mW, while the vector wise accelerator shows a higher area of 46,172 μm^2 but slightly lower power usage at 15.98 mW. The energy-efficient CNN design provides only power consumption data, reporting 9.43 mW, with no area specified.

In contrast, the proposed CNN accelerator based on DARKNET-19 exhibits the lowest area and significantly improved power efficiency. It occupies only 20,375 μm^2 of silicon area and consumes 8.69 mW of power, making it the most optimized among the designs listed. This demonstrates the proposed accelerator's advantage in minimizing hardware resources while maintaining low power consumption—key features for real-time object detection tasks in embedded and power-sensitive applications. The results validate the suitability of the DARKNET-19-based design for next-generation CNN hardware accelerators targeting efficient edge computing.

5.2.1 POWER ANALYSIS

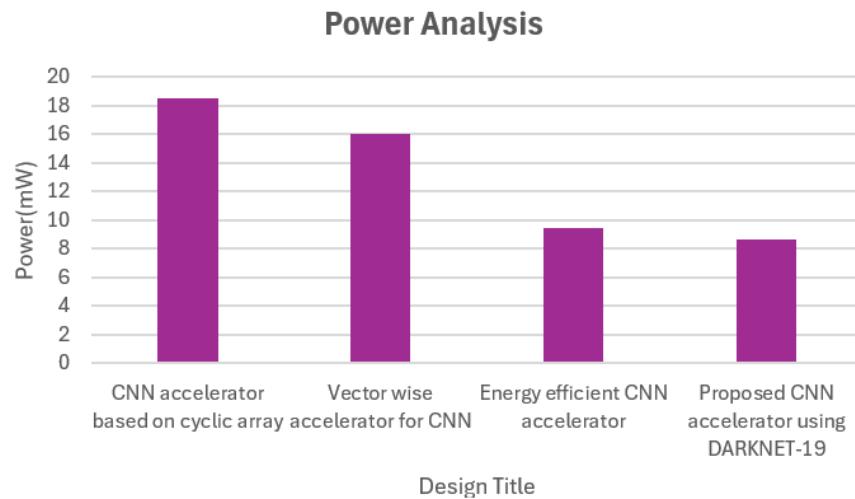


Figure 43 Power comparison of different models

The bar graph titled "Power Analysis" as shown in above figure 43 visually compares the power consumption of different CNN accelerator designs. Among the four designs shown, the proposed CNN accelerator using DARKNET-19 demonstrates the lowest power consumption, highlighting its energy-efficient nature. In contrast, the CNN accelerator based on a cyclic array consumes the highest power, indicating a less optimized architecture in terms of energy use. This comparison underscores the effectiveness of the proposed design in reducing power requirements, making it more suitable for real-time and low-power applications.

5.2.2 AREA ANALYSIS

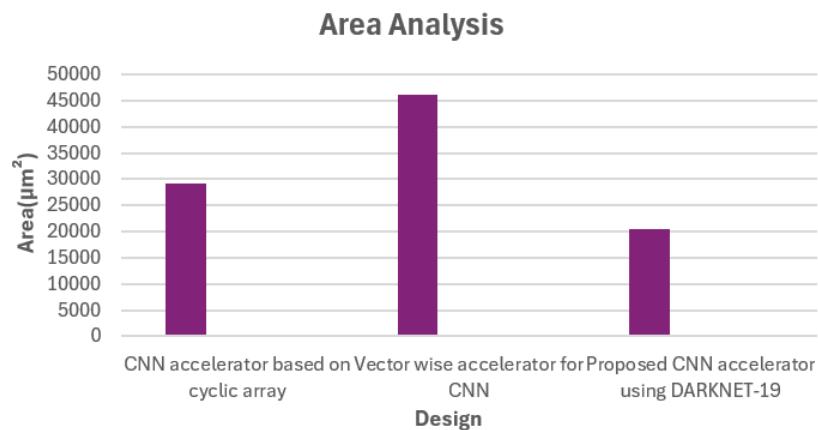


Figure 44 Area Comparison of different models

The bar graph titled "Area Analysis" visually shown in figure 44 compares the silicon area utilization of different CNN accelerator designs. It shows that the proposed CNN accelerator using DARKNET-19 consumes less area compared to the other architectures. Among the listed designs, the vector-wise accelerator occupies the highest area, while the cyclic array-based design falls in between. This comparison highlights the area efficiency of the proposed architecture, making it suitable for compact and resource-constrained hardware applications.

6.CONCLUSION

This project presents the design, development, and verification of a hardware-optimized CNN accelerator for enabling real-time object detection capabilities, specifically utilizing the DARKNET-19 architecture within the YOLOv3 framework. The objective was to implement an energy-efficient and area-optimized accelerator capable of supporting real-time image processing tasks while being suitable for deployment on platforms with limited resources, including embedded systems and IoT devices.

To achieve this, a simplified yet powerful CNN backbone, DARKNET-19, was selected for its lightweight nature and real-time processing capabilities. This architecture is composed of 19 convolutional layers along with 5 max-pooling layers, forming an efficient backbone to capture both low-level and high-level feature representations from input images. The overall CNN pipeline—including convolution, batch normalization, ReLU activation, and pooling layers—was designed and verified using Verilog HDL for synthesis and MATLAB for functional verification and object detection.

The hardware implementation was thoroughly analyzed using industry-standard EDA tools. Functional simulation was carried out in Xilinx Vivado, verifying the behavior of individual modules like the convolutional layer, processing elements (PEs), and the booth multipliers. The simulation waveforms confirmed the correctness of data movement, padding, and feature map generation.

Synopsys Design Compiler was utilized for synthesis, employing the 45nm Nangate Open Cell Library for technology mapping. The synthesized gate-level netlist was optimized for area, power, and timing, and various metrics such as cell usage, power breakdown, and timing slack were extracted. The design achieved an area of approximately $20,375 \mu\text{m}^2$ and total dynamic power of 8.69 mW, which is remarkably lower than other accelerators discussed in related work.

The physical design phase was executed using Cadence Innovus, starting from floorplanning and power planning to placement, clock tree synthesis (CTS), routing, and DRC verification. Clock tree synthesis was carefully performed to ensure minimal clock skew and optimal propagation delay across all sequential elements. Special attention was given to power distribution through the creation of core rings, power stripes, and global routing using metal layers for VDD and VSS networks. The

design was successfully routed with no design rule violations, and final layout verification showed clean stream out with no missing instances or blockages.

To further validate the functionality of the CNN accelerator, MATLAB was used to implement and visualize the YOLOv3 detection pipeline using DARKNET-19. Detection results clearly demonstrated the model's capability to accurately identify and localize objects in real time using bounding boxes. This confirms the functionality and reliability of the proposed hardware design when integrated into a complete object detection system.

Comparison with existing CNN accelerators clearly highlights the effectiveness of the proposed system. It consumes significantly less power and occupies smaller silicon area, while maintaining accurate and real-time object detection capabilities. The combination of Booth multipliers, parallel processing elements, and lightweight CNN layers contributed to a highly efficient design in terms of both performance and resource utilization.

In conclusion, this project effectively illustrates the development of a high-performance, low-power CNN accelerator optimized for real-time object detection capabilities. The architecture not only meets the requirements for speed and accuracy but also achieves a compact and efficient hardware footprint. Future work may involve extending the design to support more complex models like YOLOv4 or YOLOv5 and implementing adaptive power-saving techniques for further efficiency.

REFERENCE

- [1] R. C. Gonzalez and R. E. Woods, *Digital Image Processing*, 4th ed., Global Edition, Pearson Education, 2018. ISBN: 978-1-292-22304-9.
- [2] J. Redmon and A. Farhadi, “YOLOv3: An Incremental Improvement,” *arXiv preprint*, arXiv:1804.02767, Apr. 2018. [Online]. Available: <https://arxiv.org/abs/1804.02767>
- [3] M. Zhao, X. Li, S. Zhu, and L. Zhou, “A Method for Accelerating Convolutional Neural Networks Based on FPGA,” in *Proc. 2019 4th Int. Conf. on Communication and Information Systems (ICCIS)*, Beijing, China, 2019, pp. 241–246, doi: 10.1109/ICCIS49662.2019.00049.
- [4] V. M. B., S. V., R. Deodurg, and S. Soundarya, “Design of Optimized CNN for Image Processing using Verilog,” in *Proc. 2023 4th IEEE Global Conference for Advancement in Technology (GCAT)*, Bangalore, India, Oct. 2023, pp. 1–6, doi: 10.1109/GCAT59970.2023.10353422.
- [5] S. S. Karapurkar, L. K. Bramhane, A. D. Rahulkar, and T. Veerakumar, “Energy-Efficient Implementation of Processing Elements for CNN Hardware Accelerator,” in *Proc. 2023 11th Int. Conf. on Emerging Trends in Engineering & Technology - Signal and Information Processing (ICETET-SIP)*, Goa, India, 2023, pp. 1–6, doi: 10.1109/ICETET-SIP58143.2023.10151639.
- [6] D. Crumley, M. Hossain, K. Martin, F. Ivey, R. Yarnell, R. F. DeMara, and Y. Bai, “Rehosting YOLOv2 Framework for Reconfigurable Fabric-Based Acceleration,” in *Proc. 2022 IEEE SoutheastCon*, Mobile, AL, USA, Apr. 2022, pp. 445–446.
- [7] S.-Y. Lee, M.-Y. Ku, S.-Y. Pan, and C.-C. Lin, “Reconfigurable and Scalable Artificial Intelligence Acceleration Hardware Architecture With RISC-V CNN Coprocessor for Real-Time Seizure Detection,” *IEEE Access*, vol. 13, pp. 31057–31068, 2025, doi: 10.1109/ACCESS.2025.3538781.
- [8] Y.-S. Song and K.-Y. Lee, “A Design of Lightweight Convolutional Neural Network Accelerator for IoT Devices,” in *Proc. 2023 14th International Conference on Ubiquitous and Future Networks (ICUFN)*, Paris, France, 2023, pp. 474–477, doi: 10.1109/ICUFN57995.2023.10200660.
- [9] Y.-S. Song and K.-Y. Lee, “A Design of Lightweight Convolutional Neural Network Accelerator for IoT Devices,” in *Proc. 2023 14th International Conference on Ubiquitous and Future Networks (ICUFN)*, Paris, France, 2023, pp. 474–477, doi: 10.1109/ICUFN57995.2023.10200660.

- [10] Y. Shen, *Accelerating CNN on FPGA: An Implementation of MobileNet on FPGA*, Master's Thesis, School of Electrical Engineering and Computer Science, KTH Royal Institute of Technology, Stockholm, Sweden, 2019. Available: https://kth.divaportal.org/smash/get/diva2:1356292/FULLTEXT_01.pdf
- [11] H. Kwon, *Designing CNN Accelerators – Day 1*, Lecture Notes, Synergy Lab, Georgia Institute of Technology, Dec. 26, 2017. Available: <http://synergy.ece.gatech.edu>
- [12] L. Alzubaidi, J. Zhang, A. J. Humaidi, A. Al-Dujaili, Y. Duan, O. Al-Shamma, J. Santamaría, M. A. Fadhel, M. Al-Amidie, and L. Farhan, "Review of deep learning: concepts, CNN architectures, challenges, applications, future directions," *Journal of Big Data*, vol. 8, no. 53, pp. 1–74, 2021, doi: 10.1186/s40537-021-00444-8.
- [13] A. Wiranata, S. A. Wibowo, R. Patmasari, R. Rahmania, and R. Mayasari, "Investigation of Padding Schemes for Faster R-CNN on Vehicle Detection," in *Proc. 2018 Int. Conf. on Control, Electronics, Renewable Energy and Communications (ICCEREC)*, Bandung, Indonesia, 2018, pp. 208–212, doi: 10.1109/ICCEREC.2018.8712046.
- [14] J. Xie, Q. Song, Z. Long, Z. Liu, Y. Du, and T. Wang, "Visible-Light Insulator Defect Detection Based on Improved YOLOv3," in *Proc. 2023 3rd Int. Conf. on Electrical Engineering and Mechatronics Technology (ICEEMT)*, Kunming, China, 2023, pp. 287–292, doi: 10.1109/ICEEMT59522.2023.10263231.
- [15] <https://analyticsindiamag.com/deep-tech/hands-on-guide-to-implement-batch-normalization-in-deep-learning-models/>
- [16] Redmon, J., & Farhadi, A. (2018). YOLOv3: An Incremental Improvement. *arXiv preprint arXiv:1804.02767*.
- [17] Redmon, J., & Farhadi, A. (2017). YOLO9000: Better, Faster, Stronger. *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*.
- [18] Lin, T.-Y., Dollár, P., Girshick, R., He, K., Hariharan, B., & Belongie, S. (2017). Feature Pyramid Networks for Object Detection. *CVPR*.

- [19] Chakure, A. (2020). All You Need to Know About YOLOv3 – You Only Look Once. *dev.to*. <https://dev.to/afrozchakure/all-you-need-to-know-about-yolo-v3-you-only-look-once-e4m>.
- [20] Zhang, C., Li, P., He, X., & Sun, Y. (2021). FPGA-Based Lightweight CNN Accelerator Using YOLOv3 with Darknet-19 Backbone. *Electronics*, 10(5), 1237. <https://www.mdpi.com/2504-4990/5/4/83>
- [21] Qiu, J., Wang, J., Yao, S., et al. (2016). Going Deeper with Embedded FPGA Platform for Convolutional Neural Network. *Proceedings of the ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, pp. 26–35.
- [22] Zhang, C., Li, P., Sun, Y., Guan, Y., Xiao, B., & Cong, J. (2015). Optimizing FPGA-based Accelerator Design for Deep Convolutional Neural Networks. *Proceedings of the ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, pp. 161–170.
- [23] Sulaiman, N., Saad, N., & Yusof, R. (2010). High Speed Booth Encoder Multiplier Design for FPGA Implementation. *Proceedings of the 2010 International Conference on Computer and Communication Engineering (ICCCE)*, pp. 1–6.
- [24] D.-H. Guem and S. Kim, “Variable Precision Multiplier for CNN Accelerators Based on Booth Algorithm,” *International Journal on Advanced Science, Engineering and Information Technology (IJASEIT)*, vol. 13, no. 3, pp. 1025–1030, 2023, doi: 10.18517/ijaseit.13.3.18456.
- [25] <https://www.maven-silicon.com/blog/physical-design-flow/>
- [26] D.-H. Guem and S. Kim, “Variable Precision Multiplier for CNN Accelerators Based on Booth Algorithm,” *International Journal on Advanced Science, Engineering and Information Technology (IJASEIT)*, vol. 13, no. 3, pp. 1025–1030, 2023, doi: 10.18517/ijaseit.13.3.18456.
- [27] A. Kyriakos, E.-A. Papatheofanous, C. Bezaitis, and D. Reisis, “Resources and Power Efficient FPGA Accelerators for Real-Time Image Classification,” *J. Imaging*, vol. 8, no. 4, pp. 1–18, Apr. 2022, doi: 10.3390/jimaging8040114.
- [28] P. L. Lahari, R. G. Poola, and S. S. Yellampalli, “High Speed and Area Efficient FPGA Implementation of CNN Accelerator for Biomedical Applications,” *Research Square*, preprint, Aug. 2023. [Online]. Available: <https://doi.org/10.21203/rs.3.rs-3312210/v1>.