IMPLEMENTATION OF A NEURAL NETWORK APPLICATION USING ACCELERATOR ON RISC-V ARCHITECTURE IN FPGA

FPGA'DE RISC-V MİMARİSİ ÜZERİNDE HIZLANDIRICI KULLANARAK YAPAY SİNİR AĞI UYGULAMASI GERÇEKLENMESİ

AHMET ANIL DÜNDAR

PROF. DR ALİ ZİYA ALKAR

Thesis Supervisor

Submitted to
Graduate School of Science and Engineering of Hacettepe University
as a Partial Fulfillment to the Requirements
for he Award of the Degree of Master Of Science
in Electrical and Electronics Engineering

2023

ABSTRACT

IMPLEMENTATION OF A NEURAL NETWORK APPLICATION USING ACCELERATOR ON RISC-V ARCHITECTURE IN FPGA

Ahmet Anıl DÜNDAR

Master of Science, Department of Electrical and Electronics Engineering

Supervisor: Prof. Dr. Ali Ziya ALKAR

February 2023, 122 pages

Nowadays, smart devices and Internet of Things (IoT) devices have become an indispensable part of our lives. Especially, IoT devices have taken an important place in every part of our lives with the changing life conditions and the development of technology. While it is estimated that there are approximately 800 million IoT devices and 8 billion other electronic devices in the world in 2010 [1]. This number is around 16 billion for IoT devices and around 10 billion for other devices in 2022[1]. With this increase, the number of IoT devices re predicted to overrun 30 billion in 2030 [2]. On the other hand, due to the Covid-19 pandemic in 2020 a production crisis occurs in the chip industry. Chip prices have increased of Intel, Advanced Micro Devices, Inc. (AMD) and Advanced RISC Machine Inc. (ARM) chips, which are currently being sold at high prices compared to 2019, have increased even more [3]. In this context, work on the RISC-V architecture, which is intended to be developed as a competitor to Intel, AMD and ARM architectures, has accelerated. With the increase in studies on RISC-V, the RISCV-V architecture is seen as a competitor to the ARM architecture [4]. In this thesis, the Instruction Set Architecture (ISA) called RISC-V, which was developed as open source, was examined. For the RISC-V architecture to compete with current architectures, it has been thought that it should be able to run complex operations with similar performance to its competitors. In addition, a neural network application was developed using RISC-V architecture and it was discussed that RISC-V architecture could be a good alternative in terms of performance for neural network and image processing applications. For this reason, the performance and operability of the system has been compared to other processor architectures by trying different RISC-V cores and processors running on simulation environment. In the last part of this study, a neural network supported object recognition based image processing application is developed and has been tested on the Xilinx Zynq UltraScale+ MPSoC ZCU102 development board by using the RISC-V architecture as a hardware accelerator driver. In this structure, peripheral units in Xilinx Zynq architecture were used and a new accelerator method was proposed using RISC-V architecture. The performance results with other known neural network supported object recognition-based image processing algorithms are examined. As a conclusion, within the scope of this thesis, the RISC-V architecture is shown to be capable of executing neural network applications, image processing applications and providing usability in performance critical IoT devices.

Keywords: RISC-V, Object Detection, FPGA Implementation, Neural Network Application, Computer Architecture, Hardware Accelerator Applications

ÖZET

FPGA'DE RISC-V MİMARİSİ ÜZERİNDE HIZLANDIRICI KULLANARAK YAPAY SİNİR AĞI UYGULAMASI GERÇEKLENMESİ

Ahmet Anıl DÜNDAR

Yüksek Lisans, Elektrik ve Elektronik Mühendisliği

Tez Danışmanı: Prof. Dr. Ali Ziya ALKAR

Şubat 2023, 122 sayfa

Günümüzde akıllı cihazlar ve Nesnelerin İnterneti (IoT) cihazları hayatımızın ayrılmaz bir parçası haline gelmiştir. Özellikle Nesnelerin İnterneti cihazları, değişen hayat şartları ve artan teknolojilerle hayatımızın her parçasında önemli yer edinmiş durumdadır. 2010 yılında dünyada yaklaşık 800 milyon IoT cihazı ve 8 milyar diğer elektronik cihazların [1] bulunduğu tahmin edilirken 2022 yılında bu sayı IoT cihazları için 16 milyar, diğer cihazlar için ise 10 milyar civarındadır [1]. Nesnelerin İnterneti cihazlarının bu artış ile 2030 yılında 30 milyarı geçmesi beklenmektedir [2]. Öte yandan 2020 yılında yaşanmış olan Covid-19 salgını ile, cip endüstrisindeki üretim krizi nedeniyle yonga fiyatları yükselmiş ve hâlihazırda yüksek ücretlere satılmakta olan Intel, AMD ve ARM çiplerinin fiyatları daha da artmıştır [3]. Bu bağlamda, Intel, AMD ve ARM mimarilerine rakip olarak geliştirilmesi amaçlanan RISC-V mimarisi üzerindeki çalışmalar hız kazanmış ve RISC-V mimarisi ARM mimarisine bir rakip olarak görülmeye başlamıştır [4]. Bu tezde, açık kaynak kodlu olarak geliştirilmiş olan RISC-V adı verilen Komut Satırı Mimarisi

incelenmiştir. RISC-V mimarisinin güncel mimarilere rakip olabilmesi için karmaşık işlemleri rakipleriyle benzer performansta çalıştırabilmesi gerektiği düşünülmüştür. Bununla birlikte RISC-V mimarisi kullanılarak bir yapay sinir ağı uygulaması geliştirilmiş ve RISC-V mimarisinin yapay sinir ağı uygulamaları ve görüntü işleme uygulamalarının çalışması konusunda performans yönünden günümüz işlemcilerine iyi bir alternatif olup olmadığı tartışılmıştır. Bu nedenle RISC-V mimarisi simülasyon ortamında çalıştırılarak farklı RISC-V çekirdekleri, farklı RISC-V mimarisine ait yazılım üzerinde çalışan işlemcileri kullanılarak görüntü işleme ve yapay sinir ağı uygulamaları denenerek sistemin performansı ve çalışabilirliği diğer işlemci mimarilerine göre kıyaslanmıştır. Çalışmanın son kısmında ise geliştirilmiş olan yapay sinir ağı destekli nesne tanımaya dayalı görüntü işleme uygulaması, Xilinx Zynq UltraScale+ MPSoC ZCU102 geliştirme kartında, RISC-V mimarisinin donanım hızlandırıcı sürücüsü olarak kullanılmasıyla test edilmiştir. Bu yapıda Xilinx Zynq mimarisinde bulunan çevresel birimlerden faydalanılmış ve RISC-V mimarisi kullanılarak yeni bir hızlandırıcı method önerilmiştir. Bilinen diğer yapay sinir ağı destekli nesne tanımaya dayalı görüntü işleme algoritmalarıyla olan performans sonuçları incelenmiştir. Sonuç olarak, bu tez kapsamında RISC-V mimarisinin; gömülü sistemler üzerine özelleştirilerek geliştirildiği takdirde; yapay sinir ağı uygulamalarını, görüntü işleme uygulamalarını çalıştırabildiği ve diğer Nesnelerin İnterneti cihazlarının sağladığı performans açısından kritik alanlarda kullanılabilirliğini kanıtladığı gösterilmiştir.

Anahtar Kelimeler: RISC-V, Nesne Tanıma, FPGA Gerçeklemesi, Yapay Sinir Ağı Uygulaması, Bilgisayar Mimarisi, Donanım Hızlandırıcı Uygulamalar

CONTENTS

LIST OF FIGURES

LIST OF TABLES

SYMBOLS AND ABBREVIATIONS

Symbols

$	Command Line of Linux

Abbreviations

ACP	Accelerator Coherency Ports
ALU	Arithmetic Logic Unit
AMBA	Advanced Microcontroller Bus Architecture
AMD	Advanced Micro Devices, Inc.
ANN	Artificial Neural Network
ARM	Advanced RISC Machine
ASIC	Application Specific Integrated Circuits
AXI	Advanced eXtensible Interface
BAR	Berkeley Architecture Research
BOOM	Berkeley Out Of Order Machine
CAD	Computer Aided Design
CIFAR	The Canadian Institute for Advanced Research

CISC	Complex Instruction Set Computer
CNN	Convolutional Neural Network
CPU	Central Processing Unit
CSR	Control Status Registers
CUDA	Compute Unified Device Architecture
DMA	Direct Memory Access
DNN	Deep Neural Network
DPU	Deep Learning Processor Unit
DRAM	Dynamic Random Access Memory
DSP	Digital Signal Processor
ECG	Electrocardiography
EEG	Electroencephalogram
ELF	Executable and Linkable Format
EMG	Electromyography
FAST	Features from Accelerated Segment Test
FPD	Full Power Domain
FPGA	Field Programmable Gate Array
GB	GigaByte
GCC	GNU C Compiler
GDB	GNU Debugger
GHz	Gigahertz
GNU	GNU’s not UNIX
GPR	General Purpose Registers
GPU	Graphics Processor Unit
H	Arithmetic Logic Unit

HOG	Histogram of Oriented Gradients
HS	Hyperextended Supervisor
Hz	Hertz
ILSVRC	ImageNet Large Scale Visual Recognition Challenge
IoT	Internet of Things
IP	Intellectual Property
ISA	Instruction Set Architecture
KB	KiloByte
kNN	k Nearest Neighborhood
km	kilometer
KVM	Kernel-based Virtual Machine
LLVM	Low Level Virtual Machine
LPD	Low Power Domain
LUT	Look-Up Table
M	Machine Level Privilege
MB	MegaByte
MHz	Megahertz
min	minute
MIPS	Million Instructions per Second
MMU	Memory Management Unit
OCM	On-Chip Memory
OS	Operating Systems
PC	Program Counter
PE	Processing Elements
PL	Programmable Logic

PS	Processing System
QEMU	Quick Emulator
QoS	Quality of Service
RAM	Random Access Memory
ReLU	Rectified Linear Unit
RGB	Red-Green-Blue
RISC	Reduced Instruction Set Computer
RNN	Recurrent Neural Network
ROM	Read Only Memory
RTL	Register Transfer Level
S	Supervisor Level Privileges
sec	second
SIFT	Scale-Invariant Feature Transform
SOC	System on Chip
SP	Stack Pointer
SURF	Speeded Up Robust Features
SVM	Support Vector Machine
U	User Level Privileges
UART	Universal Asynchronous Receiver Transmitter
UCB	University of California, Berkeley
VS	Virtual Supervisor
VU	Virtual User
YOLO	You Only Look Once

1. INTRODUCTION

1.1. Computer Architecture and RISC-V

In recent years, there have been significant developments in the field of processor architecture such as Apple M2 Chipset with 5-nanometer technology and a 40% faster Neural Engine and unified Random Access Memory (RAM) architecture [5]. As technology develops continuously, the performance requirement of processor architectures also increases. The Instruction Set Architecture (ISA) has a very important role in the management of the processor by the software. ISA basically, is a set of specifications that define the characteristics of a microprocessor and how a programmer can use those characteristics to write machine language programs that can be executed on the microprocessor. The ISA specifies the instructions that a microprocessor can execute the format of the data with which the instructions operate, and the input/output mechanisms used by the instructions. In other words, the ISA defines the language that a programmer can use to communicate with a microprocessor and the rules that the microprocessor uses to interpret and execute those instructions and acts as a bridge between software and hardware [6]. The main hierarchy between hardware and software is given in Table-1.1.

Table 1. 1 – Hardware and Software Layers [7]

<table>
<tr><th>Layer Number</th><th>Name of Layers</th><th>Medium</th></tr>
<tr><td>10</td><td>Application</td><td>Software</td></tr>
<tr><td>9</td><td>Algorithm</td><td>Software</td></tr>
<tr><td>8</td><td>Programming Language</td><td>Software</td></tr>
<tr><td>7</td><td>Assembly Language</td><td>Software</td></tr>
<tr><td>6</td><td>Machine Code</td><td>Software</td></tr>
<tr><td rowspan="2">5</td><td rowspan="2">Instruction Set Architecture</td><td>Software</td></tr>
<tr><td>Hardware</td></tr>
<tr><td>4</td><td>Micro Architecture</td><td>Hardware</td></tr>
</table>

3	Gate Level Architecture	Hardware
2	Devices	Hardware
1	Physical Layer	Hardware

Basically, processors are designed by choosing one of the instruction architectures. These architectures have two approaches, Reduced Instruction Set Computer (RISC) principle and Complex Instruction Set Computer (CISC) principle [8]. CISC principle is hardware-based, with fewer lines of assembly code, information stored in the memory, and high clock cycle times are considered [9]. On the contrary, the RISC architecture is based on a software-based structure with longer lines of assembly codes but fewer number of instructions, aiming to perform operations on register units with fewer clocks per unit of time [9]. Considering that each instruction takes one clock cycle in the RISC approach, it seems possible to perform the same functions in fewer transistors and in less physical space.

The CISC principle tries to reduce the number of instructions per program and compromises the number of cycles per instruction. In contrast, RISC does the opposite by reducing cycles per instruction at the expense of the number of instructions per program. The following equation shows the performance capability of the computer. As can be seen in Eq.1.1 [10] there is a trade-off between the number of instructions targeted to be executed and the execution time of the instructions.

$$\frac{time}{program} = \frac{time}{cycle} \times \frac{cycle}{instruction} \times \frac{instruction}{program} \qquad (1.1)$$

The following information is basically required to create an instruction set:

- Define the goals and requirements of the system, including the types of applications and workloads it will support.
- A basic instruction set architecture, such as RISC or CISC, is chosen and how the system can be modified and expanded to meet needs is determined.

- Considering factors such as complexity, the number of operands needed, and the time they will take to execute, individual instructions are designed to build the ISA.
- A detailed specification must be developed for the ISA, including all instructions, their encoding, and the steps the processor must follow to execute them.
- ISA is implemented in hardware or software and tested to ensure that it works as expected and meets defined goals and requirements.

Nevertheless, designing an ISA is not complicated, the developed ISA licenses are sold at very high prices. For example, the ARM sells licenses for $1 million to $10 million [11]. Although the ISA contains only the instructions that the processor is supposed to execute, conventional ISAs offer easier implementation and can reach more users, and companies pay royalties to use these licenses. For example, ARM pays royalties to Intel because it uses the x86 architecture, while Intel pays royalties to ARM for using the x86-64 architecture it developed.

On the other hand, in addition to the ARM architecture, which has been actively used since the 1990s [12], the RISC-V architecture, which has been studied since 2010s [13], aims to be used for computers and all IoT devices, rather than just an academic study. RISC-V architecture is developed as open source and has an open architecture. In this regard, RISC-V does not restrict developers to predefined instructions, but only has base instructions, allowing developers and users to write additional instructions. RISC-V, which has attracted the attention of many companies and developers in this regard, also offers other innovations in the software. RISC-V supports the integer operations, single and double precision floating point operations [13] and fixes the most important bits to speed up operations. In addition, RISC-V can accommodate variable bit width structures, and uses multiplexer structures within the Central Processing Unit (CPU) for bit patterns. Similar to today's processors, RISC-V cores can work with word widths of 32, 64, and 128 bits [14]. For these reasons, the World's largest companies support RISC-V and projects related to RISC-V are constantly evolving [15].

1.2. Introduction to Convolutional Neural Networks

The history of machine learning dates back to the 1950s, although without the term of machine learning [16]. In that decade, researchers began exploring the concept that computers could learn from data without being explicitly programmed. In 1950's a computer program is written that allows a computer to play the checkers game [17]. The program was able to improve its performance by learning from its mistakes, and eventually became one of the best checkers players in the world. In the 1960s, some researchers began to explore the idea of using neural networks, networks of interconnected nodes that mimic the way the brain processes information, for machine learning. This led to the development of a number of algorithms and techniques, such as the backpropagation algorithm [18, 20], which is still used in many machine learning applications today. In the 1970s, the ability to learn from past experience was acquired while working on systems that were capable of making decisions and solving problems [19, 20] in the fields of economics, medicine and finance. The focus of work done in the 1980s shifted to areas that could be used in more practical domains, such as speech recognition systems and predictive models. In this way, the systems used today, such as vector machines and decision trees [20], emerged. In the 1990s, studies were conducted on new techniques and algorithms such as deep learning [21] and reinforcement learning. Systems in which image processing and speech recognition systems are extensively studied have laid the foundation for a field such as computer vision, natural language processing, robotics, and autonomous devices [22].

Despite machine learning progresses, neural network developments proceed differently. In the field of neural networks, in the 1950s and 1960s, researchers studying the information processing processes of the human brain continued to explore the potential of neural networks by developing models to simulate human brains [23]. However, this initial network expectation to use what could not achieve the desired results due to limited computing power and complexity. The field of neural networks, whose development slowed down in the 1970s, is on the rise again with the revival of operations and large amounts of data obtained in the 1980s and 1990s. Backpropagation and convolutional structures [20] of neural network structures were also developed during this period and became an important situation that is still in use.

In addition, deep learning methods, which are one of the most important branches of machine learning and form the basis of the Convolutional Neural Network (CNN) structure, aim to learn from data, make predictions and make decisions based on these data. It is called deep learning since it uses neural networks to model and understand complex data and have many layers of interconnected nodes.

On the other hand, the convolutional neural network has been used frequently since the 1980s, especially in the field of video and image processing. One of the most important advantages of CNN is that it has the ability to learn directly from the data instead of being designed manually by humans. Thanks to this ability, CNN structures automatically learn edges, corners, different textures and many different complex features in images. The CNN structure is inspired by the working mechanism of the visual cortex [23, 24]. Visual cortex is the part of the human brain that processes visual information; there are interconnected layers, each of which performs a specific task. The input layer receives the raw data from the user and undergoes a series of filters to understand certain patterns. The output of these layers, called the convolutional layer, contains the information necessary to make a comment about the content of the image. One or more fully connected layers are used to make predictions about the content from previous layer's information [25]. CNN structures that are used in many areas such as object recognition face recognition, natural language processing, speech recognition, video processing, and medical imaging and even training other neural networks, are making progress in performance and accuracy with the development of deep learning.

1.3. Motivation of the Thesis

Even though computers which contain x86 architecture (Intel and AMD processors) work with CISC architectures, RISC architecture is gaining power since IoT devices involved in our lives. RISC architecture promises less power consumption thus it is ideal for smart devices. RISC-V on the other hand, is designed in a customizable structure, unlike the RISC architecture. RISC-V is a new, open-sourced and open-standard ISA. This free and open ISA offers a unique opportunity to design and build custom processors for a variety of applications. In other words, RISC-V starts working with base instructions only and allows users to add external instructions. By this means, the processor, which uses RISC-V architecture, can be customizable and may perform designated operations and it is thought that over time, it can replace today's processors. There are many different opinions on this matter. For example, Google stated that they want RISC-V to be developed on Android as a “tier-1 platform” with the same standards as ARM, and it may take a long time to develop an Android operating system that can run on RISC-V [26]. On the other hand, Nvidia stated that the RISC-V architecture is not suitable for advanced systems and third-party software, and announced that they have stopped the software support of RISC-V on Nvidia Graphics Processor Unit (GPU) [27].

On the other hand, artificial neural network applications started to take place in every aspect of human life. Especially with the increase in technological needs and the increase in the number of IoT devices, many different artificial neural network applications are frequently taking place in people's lives. Neural network supported image processing technologies, one of the most important of these, are now frequently used in the security of smart phones, home security systems, city and traffic cameras.

Object recognition algorithms, on the other hand, is a field that has emerged with the combination of neural network and computer vision and image processing technologies. This method, which aims at recognizing the object or objects in a given image by a trained algorithm, has been actively studied since 2001 [28]. Although the convolutional neural network structure was initially studied with LeNet in the 1980s [29], it was actively applied to object recognition algorithms in the following years. Object recognition algorithms, which worked with classical methods until 2012, started to include deep learning methods with the introduction of AlexNet in 2012 [30]. Today, neural network assisted image processing algorithms are still a field that is being studied with the discovery of new methods, the expansion of datasets, and the increase in computing power.

Since the RISC-V and neural network approaches are popular in computer architecture and in the industry, the aim of the study is to find a RISC-V based and high performance solution for neural network based image processing applications. In addition, in this thesis, we would like to increase the operability of RISC-V architecture and to use RISC-V architecture as a hardware accelerator in artificial neural network algorithms running on Field Programmable Gate Array (FPGA)-based systems. By comparing the applied method with other efficient application methods, we would like to prove that it is successful in terms of performance and accuracy parameters, which are the building blocks of neural network and image processing applications.

1.4. Thesis Work

RISC-V, which is an architecture that is frequently used in codes running on FPGAs and in the field of IoT in recent years, with its user-customizable capabilities and capable of performing complex operations using less and basically designed instructions, is an instruction set architecture of choice for applications that require high performance.

The RISC-V architecture has the most common examples of RISC type architectures today. In this study, although the structure of the base instructions that are determined by the RISC-V developers were not changed on RISC-V, the application area was differentiated and code execution processes from different types of use cases and languages were carried out. The C++ compiler system was built using the RISC-V basic instruction set on the Linux operating system as a softcore processor with Chipyard, Berkeley Out Of Order Machine (BOOM) and Gemmini architectures [31, 32]. Thus, simple mathematical operations, matrix multiplications, image filtering, image processing operations and neural network applications executed on the RISC-V system. After this process, an Arch Linux-based operating system was installed on the RISC-V system, and C++ and Python codes were executed on this virtual operating system. Python libraries were installed and simple mathematical operations, matrix multiplications, image filtering, image processing operations and neural network codes were executed. In the third part of the study, the image processing library adapted to the RISC-V architecture by using C and C++ libraries are executed on the Linux operating system and image processing operations with the RISC-V architecture are examined.

In the last part of this study, Xilinx Zynq UltraScale+ MPSoC ZCU102 model FPGA was used. Basically, here we aim to design a neural network supported object recognition based image processing and computer vision algorithm with the help of algorithms containing convolutional neural network structure such as ResNet. In addition, the designed algorithm is compared in terms of performance and accuracy with algorithms that have proven their success, such as ResNet and SqueezeNet. The results were compared with the computers used today and the usability of the RISC-V architecture in neural network applications is discussed.

This thesis work consists of the background work in Section 2, which provides a background on the RISC-V instruction set, the RISC-V architecture, the structure of the CNN, the structure of the object recognition algorithm, and related work. In Section-3, RISC-V architectures working on software are examined first. In this context, image processing was implemented on the Chipyard System on Chip (SoC) Generator and Gemmini accelerator. Then, image processing and neural network application are simulated by running Linux on a RISC-V kernel created with the Quick Emulator (QEMU). At the end of the RISC-V architecture review working on the software, the OpenCV library and dependencies are installed for the RISC-V ISA, and image processing and neural network applications are run with the cross-compiling method. The results of image processing and neural network applications run with RISC-V cores on the software are examined. When RISC-V ISA is run on software, it has been seen with the results that it does not have better time results than its competitors ARM and Intel architecture. Therefore, in the last part of the study, a new method is proposed in which the RISC-V core is run on FPGA. According to this method, the RISC-V core manages the Deep Learning Processing Unit (DPU) and ARM cores in the Xilinx Zynq architecture over the Advanced eXtensible Interface (AXI) data transmission interface, creating a structure with similar time performance to its competitors. With this method, it has been shown that the RISC-V architecture can have the capabilities to perform complex operations if customized. The results and comparison of this thesis work with other implementations are presented as results and discussion in Section 4.

2. BACKGROUND WORK

In this section, the working mechanisms and details of RISC-V, artificial neural networks and image processing algorithms used in the thesis are explained. The information explained in this section is used as input in the Thesis Work in Section-3.

2.1. RISC-V ISA

RISC-V is an open source ISA designed using the RISC approach. RISC-V architecture has been developed by the University of California Berkeley (UCB), which has been doing research on RISC architecture for many years [33]. Although it was initially developed for educational purposes, it is now shown as a competitor to x86, ARM and Million Instructions per Second (MIPS) architectures. Since does not require commercial transactions and facilitates the design of the processor and availability of the customization RISC-V is widely used. Since 2015, instead of UCB, the RISC-V Foundation has been contributing to the use of RISC-V in the industry by managing research and studies. [34]

2.1.1. RISC-V Design Privileges

Based on the load-store RISC architecture, RISC-V operates on memory or registers and main working principle of ISA is divided into 3 privilege levels [35]. These are user level, supervisor level and machine level. Machine level privilege ("M") has the highest level [36] of privileges and specifies the basic hardware specifications required for the RISC-V architecture to work. If only machine level mode is used, an environment that is completely open to external development. The idea of privileges is crucial because they allow the operating system to carry out tasks that would be illegal for programs running in user space. This enables isolation and protection because no process can access another's memory, and when a process crashes, the entire system is not corrupted. Hardware must offer many levels of privileges and the flexibility to switch between them in order to support this approach. On the other hand, User level privilege ("U") [36] has the least amount of privileges. The RISC-V documentation has released the ("H") Hypervisor extension and appends three new privileged modes to the system [35]. These modes are ("HS") Hyperextended Supervisor, ("VS") Virtual Supervisor, and ("VU") Virtual User. It is recommended to be used for integrating requests that are difficult or expensive to integrate into the hardware. User-level privileges allow users to make certain enhancements, while pre-defining key hardware specifications, drawing a secure boundary between the hardware and the user. Most development boards with RISC-V architecture have user level privileges. On the other hand, supervisor level privileges, which are denoted in Table-2.1 creates complete isolation and allow the user to work on RISC-V using an operating system.

Table 2. 1 – RISC-V ISA Privileges [35]

Level	Encoding	Name	Abbreviation
0	00	User/Application Level	U
1	01	Supervisor Level	S
2	10	Reserved (Not Used)	-
3	11	Machine Level	M

2.1.2. RISC-V Extensions

Since the output purpose is aimed to allow a stable development environment, 32-bit, 64-bit stable and 128-bit trial versions have been released [14]. The instructions are differentiated according to the released versions and it is aimed not to use unnecessary instructions. For this, "I" base integer version, "M" arithmetic code version, "F" and/or "D" floating-point version, instructions for these versions have been released separately, and it is shown in Table-2.2. The RV64I which is indicates a RISC-V architecture with an instruction set that supports the 64-bit base integer instructions. In addition to these, RISC-V also has sub extensions that define the standard enhancement capabilities, which are provided by ISA as seen in Table-2.3 [37]. Thus, it offers a structure that can be expanded and developed by users when requested.

Table 2. 2 – RISC-V ISA Main Extensions [37]

Extension Name	**Description**
RV32I	32 bit Base Integer ISA
RV32E	32 bit Embedded Base Integer ISA
RV64I	64-bit Base Integer ISA
RV128I	128-bit Base Integer ISA

Table 2. 3 – RISC-V ISA Sub Extensions [38]

Extension Type	**Usage Process**
M	Integer Multiplication and Division
A	Atomic Instructions
F	Single-Precision Floating-Point
D	Double-Precision Floating-Point
G	Base and above extensions
Q	Quad-Precision Floating-Point
L	Decimal Floating-Point
C	Compressed Instructions
B	Bit Manipulation
J	Dynamically Translated Languages
T	Transactional Memory
P	Packed-SIMD Instructions
V	Vector Operations
N	User-Level Interrupts
H	Hypervisor
S	Supervisor-level Instructions
Zbkb, Zbkc, Zbkx etc.	Cryptography Extensions
Zve32x,Zve32f, Zve64x etc.	Vector Extensions

Zicntr, Zihpm	Basic Timers and Counters

A proper RISC-V definition must follow the order in Table-2.3 respectively. In this manner defining the RISC-V ISA subset, for example, RV64IMADJ can be useful. In this research, the ("J") version is frequently used due to the relationship between RISC-V and C++ and Python programming languages. In this research, a 64-bit integer version of the RISC-V ISA (RV64I) is used. The instructions and registers which will be mentioned in Section-2.1.3 contain 64-bit register information.

2.1.3. RISC-V Instructions and Registers

The RAM of a computer stores programs in the form of 1's and 0's, which the CPU interprets as instructions [39]. One machine language instruction is contained in one word that is a term in which a computer operates with the RAM as shown in Eq-2.1. These instructions are loaded one at a time into the CPU, where they are decoded and executed. The memory reference instruction, register reference instruction, and input-output instruction format are the 3-instruction code types found in a basic computer. Besides the R type, I type and J type instructions in the MIPS architecture [40], the RV64I architecture has a total of 6 instruction [34] types with 3 different instructions such as S type, U type and B type. RISC-V instruction types are shown in Table-2.4 and their detailed structure is demonstrated in Table-2.5 R-Type, I-Type, S-Type, SB-Type, U-Type and UJ-Type (J-Type) respectively. One word corresponds to four bytes and 32-bits in little endian convention and 32-bit address formats form structures called fields such as Eq-2.1.

$$1\, word = 4\, bytes = 32 bits \tag{2.1}$$

Table 2. 4 – RISC-V ISA Instruction Types

Instruction Type	**Usage Process**	**Example**
R Type	Arithmetic and Logical Operations	mul, add, or
I Type	Immediate and Load Instructions	slli, sltiu, addi
J Type (UJ Type)	Jump Instructions	jal
S Type	Store Instructions	sw, sd, sh, sb

U Type	Upper Immediate Instructions	lui, auipc
B Type (SB Type)	Branch Instructions	beq, bgeu, bge

The two key terms in computer science and computer architecture, are instructions and registers. The fundamental units that make up a computer program are instructions, which specify what the machine should do. Programming languages, which are sets of rules and syntax that determine how a computer should do specific tasks, are implemented by instructions.

Table 2. 5 – RISC-V ISA Core Instruction Format for all Types [34]

<table>
<tr><td>31 27 26 25</td><td>24 20</td><td>19 15</td><td>14 12</td><td>11 7</td><td>6 0</td></tr>
<tr><td>function</td><td>rs2 (input2)</td><td>rs1 (input1)</td><td>function</td><td>rd (output)</td><td>opcode</td></tr>
<tr><td colspan="2">imm[11:0]</td><td>rs1 (input1)</td><td>function</td><td>rd (output)</td><td>opcode</td></tr>
<tr><td>imm[11:5]</td><td>rs2 (input2)</td><td>rs1 (input1)</td><td>function</td><td>imm[4:0]</td><td>opcode</td></tr>
<tr><td>imm[12|10:5]</td><td>rs2 (input2)</td><td>rs1 (input1)</td><td>function</td><td>imm[4:1|11]</td><td>opcode</td></tr>
<tr><td colspan="4">imm[31:12]</td><td>rd (output)</td><td>opcode</td></tr>
<tr><td colspan="4">imm[20|10:1|11|19:12]</td><td>rd (output)</td><td>opcode</td></tr>
</table>

The instructions are then translated into a form that the computer can understand and execute. They are a necessary component of computer programming because they enable programmers to develop programs and apps that can execute a variety of functions. Instructions are frequently arranged into groups known as functions or methods that can be called upon to accomplish specific tasks as needed. RV64I has additional instruction variations for manipulating 32-bit values, denoted by ”W” ending in the opcode. These “W” instructions always output 32-bit signed values, sign extend them to 64 bits, and disregard the upper 32 bits of their inputs.

ADDIW instruction, for example, is an RV64I instruction that performs add operation the sign-extended 12-bit instant to register and generates the necessary sign-extension of a 32-bit result. The general structure for I-Type instructions are shown in Table-2.6. Overflows are disregarded, and the final result is the result's low 32 bits sign-extended to 64 bits.

Table 2. 6 – RISC-V Core Instruction Format for I-Type [34]

31 27 26 25 24 20	19 15	14 12	11 7	6 0
imm[11:0]	rs1	function	rd	opcode
I-immediate[11:0]	src	ADDIW	dest	OP-IMM-32

Shift operations are represented employing the same instruction opcode as RV32I as a specialization of the I-type instruction format. The operand to be shifted and the amount of shift is stored in the lowest 6 bits of RV64I's I-immediate field. The 30th bit represents the right shift type. SRLI is a logical left shift, SRAI is an arithmetic right shift and the sign bit is copied into the nullified upper bits. The format for shift immediate operation is shown in Table-2.7.

Table 2. 7 – RISC-V Core Instruction Format for Shift Immediate Operations [34]

31 27 26	25	24 20	19 15	14 12	11 7	6 0
imm[11:6]	[5]	imm[4:0]	rs1	function	rd	opcode
000000	[5]	shamt[4:0]	src	SLLI	dest	OP-IMM
000000	[5]	shamt[4:0]	src	SRLI	dest	OP-IMM
010000	0	shamt[4:0]	src	SRAI	dest	OP-IMM
000000	0	shamt[4:0]	src	SRLIW	dest	OP-IMM-32
000000	0	shamt[4:0]	src	SLLIW	dest	OP-IMM-32
010000	0	shamt[4:0]	src	SRAIW	dest	OP-IMM-32

The Program Counter (PC) is a register in the CPU of a computer that contains the address of the next instruction that will be executed. When the CPU reads an instruction from memory, it does so from the memory location designated by the PC. The PC is then advanced to point to the next instruction in memory, ensuring that the instruction is read from the right memory address when the CPU fetches the following instruction and AUIPC instruction in RISC-V ISA which is used to build the PC [34]. The opcode LUI is the same as RV32I. LUI inserts the 32-bit U-immediate, leaving the bottom 12 bits empty. The result is sign-extended to 64-bits from 32-bits. The opcode AUIPC is the same as RV32I [34]. AUIPC, which employs the U-type format, is used to create PC relative addresses. AUIPC creates a 32-bit offset from the U-immediate that demonstrates in Table-2.8, replacing the bottom 12-bits are filled with zeros and sign-extending the result to 64-bits. After adding it to the AUIPC instruction's address, the result is stored in its register.

Table 2. 8 – RISC-V Core Instruction Format for U-Type [34]

31 27 26 25 24 20 19 15 14 12	11 7	6 0
imm[31:12]	rd	opcode
U-immediate[31:12]	dest	LUI
U-immediate[31:12]	dest	AUIPC

The LD instruction loads a 64-bit value from memory into RV64I registers. The LW instruction reads a 32-bit value from memory and signs it up to 64 bits before placing it in the RV64I registers. In contrast, the LWU instruction zero-extends the 32-bit value from memory for RV64I [34]. For 16-bit values, LH and LHU are defined similarly, as are LB and LBU for 8-bit values. The SB instruction stores 8 bits, the SH instruction stores 16 bits, SW instruction stores 32 bits and SD instruction stores 64 bits values from the low bits of the register to memory. The main structure for all types of load and store instructions are detailed in Table-2.9.

Table 2. 9 – RISC-V Core Instruction Format for Load and Store Instructions [34]

31 27 26 25 24 20	19 15	14 12	11 7	6 0
imm[11:0]	rs1	function	rd	opcode
offset[11:0]	base	width	dest	LOAD

31 27 26 25	24 20	19 15	14 12	11 7	6 0
imm[11:5]	rs2	rs1	function	imm[4:0]	opcode
offset[11:5]	src	base	width	offset[4:0]	STRORE

ADDW and SUBW instructions are RV64I-only instructions that work on 32-bit data and output signed 32-bit results, similar to ADD and SUB. The instruction structures that do not contain immediate are explained in Figure-2.10. Overflows are ignored, and the result's low 32 bits are sign-extended to 64 bits before being put to the target register. SLL, SRL, and SRA conduct logical left and right respectively and arithmetic right shifts by the shift amount maintained on the value in the register. Only the low 6 bits of its register are evaluated for the shift amount in RV64I.

Table 2. 10 – RISC-V Core Instruction Format for Without Immediate [34]

31 27 26 25	24 20	19 15	14 12	11 7	6 0
function	rs2	rs1	function	rd	opcode
0000000	src2	src1	SLL	dest	OP
0100000	src2	src1	SRL	dest	OP
0100000	src2	src1	SRA	dest	OP
0000000	src2	src1	ADDW	dest	OP-32
0000000	src2	src1	SLLW	dest	OP-32
0000000	src2	src1	SRLW	dest	OP-32
0100000	src2	src1	SUBW	dest	OP-32
0100000	src2	src1	SRAW	dest	OP-32

Contrarily, registers are tiny and fast memory components that are a part of the CPU of computers. The smallest unit of information in a computer, a bit, makes up each instruction in a computer program. The CPU's registers, which are compact, high-speed memory components incorporated into the CPU, are where the instructions are pulled from memory and stored [41]. Instructions and registers work together in that the CPU executes them after reading them from the registers after being fetched from memory. The registers are essential to the speed and effectiveness of a computer because they store the data and instructions that the CPU is currently processing.

Registers are utilized in the RISC-V architecture to store data and instructions as they are processed. RISC-V contains 32 integer registers and 32 floating-point registers, each of which is 64 bits large. These registers are used to store data values and addresses, as well as intermediate results generated during instruction execution. RISC-V additionally features a variety of special-purpose registers for specialized tasks such as storing the PC or the stack pointer (SP). These special-purpose registers are used to regulate program execution and manage data flow between the CPU and the main memory.

There are 32 General Purpose Registers (GPR) in the RISC-V ISA. These registers, are described in the unprivileged specification. GPRs are also may referred to as integer registers. On the other hand, Control and Status Registers (CSR) are privileged registers described in the RISC-V standard [42]. GPRs are available at any privilege level, unlike CSRs, which are defined at a certain privilege level and may only be accessed by that level and any levels of greater power [43]. The privileged specification specifies a common set of CSRs, as well as, address ranges that each CSR must be situated inside. 12-bit address space is defined for CSR in RISC-V ISA. Therefore, there can be a maximum of 4096 CSRs from $2^{12} = 4096$. A CSR address's bits describe its usefulness, accessibility, and CSR number. All details about Control and Status Registers are shown in Table-2.11.

Table 2. 11 – RISC-V Core Instruction Format for Control and Status Register [34]

11	10	9	8	7	6	5	4	3	2	1	0
Read/Write Mode Select		Privilege		Usage				Number			
00 (R/W) 01 (R/W) 10 (R/W) 11 (Read Only)		00 (User) 01 (Supervised) 10 (Hypervisor) 11 (Machine)		Standard Usage or Custom Usage				4- Bit Number field From 0000 to 1111			

In the RISC-V architecture, integer registers are kind of registers that are utilized to hold integer data values. The integer registers are shown in Table-2.12. An integer is an entire number, either positive or negative, without a decimal point. There are 32 integer registers in RISC-V (RV64 version), which are each 64-bits long. In RISC-V, integer registers are used to store data values and addresses, as well as intermediate outcomes during execution. For example, when executing an arithmetic operation, the operands and results are stored in integer registers. In RISC-V, integer registers are also used to store control information and status information. For instance, the PC register stores the address of the presently executed instruction, whereas the SP register stores the address of the top of the stack. Thus, integer registers [44] in RISC-V are critical to CPU performance and efficiency because they enable the CPU to rapidly access and alter integer data as they are processed. The floating point register denoted by an “x” prefix.

Table 2. 12 – RISC-V Integer Registers [44]

Register	**ABI Name**	**Description**
x0	zero	Hard-wired zero
x1	ra	Return address
x2	sp	Stack pointer
x3	gp	Global pointer
x4	tp	Thread pointer
x5	t0	Alternate link register

x6, x7	t1, t2	Temporaries
x8	s0/fp	Saved register and frame pointer
x9	s1	Saved register
x10, x11	a0, a1	Function arguments and return values
x12, x13, x14, x15, x16, x7	a2, a3, a4, a5, a6, a7	Function arguments
x18 to x27	s2 to s11	Saved registers
x28, x29, x30, x31	t3, t4, t5, t6	Temporaries

Similar to integer registers; floating-point registers [44] store intermediate values of operations. There are 32 64-bit floating point registers. Likewise, floating point registers that store values and addresses are also able to store control and status information. The main difference from integer registers is that they can operate on decimal numbers and include a rounding mode. Rounding determines where the decimal part of the number rolls over. The floating-point registers are denoted by an "f" prefix as shown in Table-2.13.

Table 2. 13 – RISC-V Floating Point Registers [44]

Register	**ABI Name**	**Description**
f0, f1,f2, f3, f4, f5, f6, f7	ft0, ft1, ft2, ft3, ft4, ft5, ft6, ft7	Floating Point Temporaries
f8, f9	fs0, fs1	Floating Point Saved Registers
f10, f11	fa0, fa1	Floating Point Arguments and Return Values
f12, f13, f14, f15, f16, f17	fa2, fa3, fa4, fa5, fa6, fa7	Floating Point Arguments
f18 to f27	fs2 to fs11	Floating Point Saved Registers
f28, f29, f30, f31	ft8, ft9, ft10, ft11	Floating Point Temporaries

2.1.4. RISC-V Counters and Timers

Counters and timers are hardware or software devices that measure the passage of time or keep track of the number of events that have happened. Counters are components that record how many times an event occurs [45]. They are frequently used to record the number of times a machine or process has been utilized, or to determine the frequency of an electrical signal. Timers are devices or programs that measure the passage of time. They can be used to time an event or to initiate an action at a precise moment in time. Timers can be configured to count up or down, to repeat at regular intervals, or to stop after a specified period of time has passed. RISC-V counter and timer structure [34] is in Table-2.14.

Table 2. 14 – RISC-V Timers and Counters [34]

Number	Privilege	Name	Description
Timers and Counters			
0xB00	MRW	mcycle	Cycle counter
0xB02	MRW	minstret	Instruction retired counter
0xB03 and 0xB04	MRW	Mhpmcounter3-4	Performance monitoring counters

RISC-V ISAs provide a series of up to 32x64-bit performance counters and timers. These counters and timers are accessible by unprivileged XLEN read-only CSR registers [34]. RV32I has many 64-bit read-only user-level counters that are mapped into the 12-bit Control and Status Registers' address space and accessible in 32-bit chunks through CSRRS instructions [45]. On the other hand, Control and Status Registers' instructions in RV64I can modify 64-bit CSRs. The pseudo instructions like DCYCLE and RDTIME can read the whole 64 bits of the cycle and time counters. As a result, the RDCYCLEH and RDTIMEH instructions are only available in RV32I which is shown in Table-2.15 [34]. Since the RISC-V ISA configuration and "Zicntr" sub-extension are used timers and counters can be found in the CSR.

Table 2. 15 – Control and Status Register for 32 bit RISC-V Architecture [34]

31 27 26 25 24 20	19 15	14 12	11 7	6 0
CSR (control and status registers)	rs1	function	rd	opcode
RDCYCLE[H]	00000	CSRRS	dest	SYSTEM
RDTIME[H]	00000	CSRRS	dest	SYSTEM
RDINSTRET[H]	00000	CSRRS	dest	SYSTEM

2.1.5. RISC-V Relationship with Rocket Chip SoC Generator, Rocket Core, BOOM Core and Gemmini

Rocket Chip is UCB's open-source SoC design generator that produces Register Transfer Level (RTL). The Rocket Core is an in-order core configured and created using the Rocket Chip generator. They are a component of the same framework that enables SoC designers to quickly produce design architectures. The Berkeley Architecture Research (BAR) team at the UCB created the open source SoC generator known as Rocket Chip [46], which is currently supported by SiFive. The Rocket Chip is a RISC-V CPU generator that also incorporates several other SoC components. The major processor outputs of the Rocket Chip SoC generator are the Rocket Core, Berkeley Out-of-order Machine (BOOM) Core, CVA6 Core, and Ibex Core [46]. The primary accelerators of Chipyard are the Hwacha vector architecture, RoCC and Gemmini Deep Neural Network (DNN) hardware exploration. To put it simply, Rocket Chip is a tool that produces synthesizable RTL and was created to let designers create and modify their own SoCs based on the RISC-V ISA. This project is written in the Chisel dialect of Scala, which is intended for creating and building hardware. Utilizing the Chisel language's parametrization features, the SoC generator enables the creation of modular architectures. It provides the ability to adjust and customize a variety of design configuration factors, including support for various ISA standard extensions, in order to produce distinct cores that may be tailored to the designer's needs. The project itself may alternatively be seen as a repository and library of programmable parts for building SoCs. The fundamental design process permits the integration of newly developed hardware components as described in Chisel. The development of novel custom configurations, the composition of Chisel/Scala language sources to produce C++ models for cycle-accurate simulations, and the production of synthesizable RTL sources (Verilog in this work) to push the design into the widely used industry Computer-Aided Design (CAD) program tools [46].

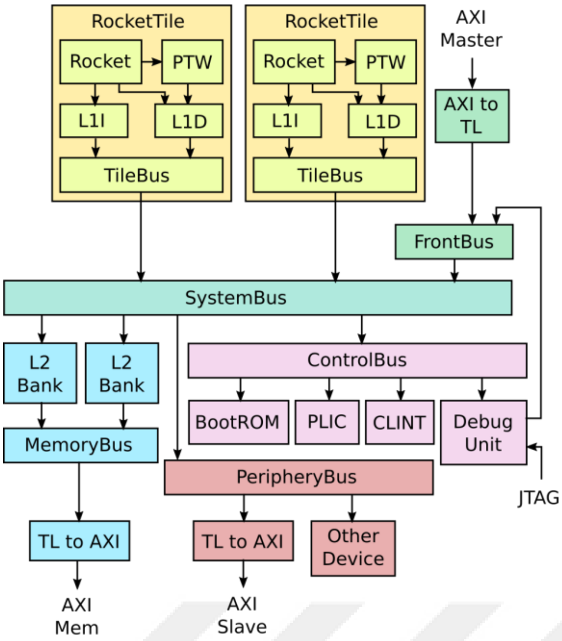


Figure 2. 1 - Detailed Diagram for Rocket Chip SoC Generator Bus Design [46]

Rocket Core is a collection of processing parts and a processor generator. It has the ability to generate a variety of processor core designs with various configuration options. The resulting processors can implement both the 32-bit and 64-bit base integer RISC-V ISAs and have the traditional five-step in-order scalar pipeline [47]. The cores feature a programmable non-blocking private data cache, a front-end with a configurable branch predictor, and a Memory Management Unit (MMU) with optional paged virtual memory support. Detailed bus and peripheral orientation of RocketChip SoC Generator can be shown in Figure-2.1. The generator exposes a wide range of options for setting the core, including the ability to add support for a few optional standard ISA extensions (such as “M”, “A”, “F” and “D” as well as choosing the cache size. Various RV32 and RV64 instruction set configurations, together with the Gemmini, which is shown in Figure-2.2 were used for this research. Gemmini is an accelerator developed by Chipyard that accelerates matrix operations. It basically consists of the systolic array and the Static Random Access Memory (SRAM) modules connected to the systolic array. Data transfer between the scratchpad and main memory is accelerated by the DMA engine [48]. A systolic array is a type of parallel processing architecture that is used to accelerate the performance of digital signal processing, linear algebra, and other computationally intensive tasks. In a systolic array, a large number of simple processing elements (PE) are arranged in a two-dimensional grid and data is passed between processing PE in a wave-like pattern. This allows for high levels of parallelism and efficient data flow, making systolic arrays well-suited for applications such as image processing and neural networks. The Gemmini accelerator is composed of a large number of small PEs, arranged in a two-dimensional array. Each PE is capable of performing a simple matrix multiplication operation on a small block of data. Data is passed between PEs in a wave-like pattern, similar to a systolic array, allowing for high levels of parallelism and efficient data flow. The accelerator also includes a specialized memory hierarchy, optimized for the data access patterns of matrix multiplications, which helps to reduce the memory bandwidth requirements.

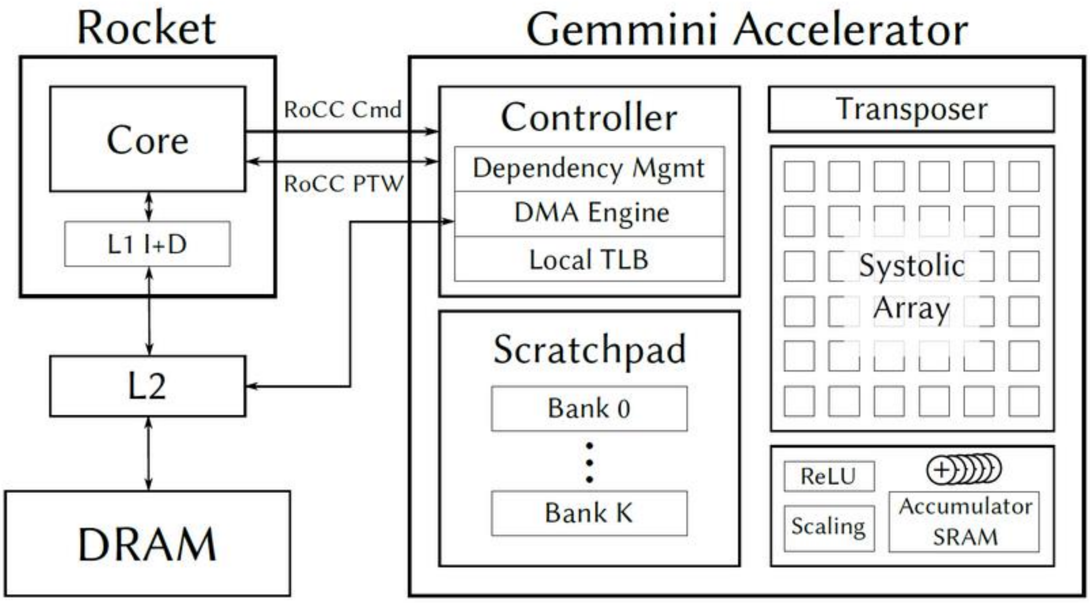


Figure 2. 2 – Gemmini Accelerator Diagram [48]

Chipyard SoC Generator also supports the AXI interface for data transmission in configurations. The Periphery_Bus It can also provide an external AXI4 port that can be connected to supplier AXI4 Intellectual Property (IP) with AXI-TileLink converters. Rocket Chip supplies libraries a library of TileLink and AXI interface. The AXI interface of RocketChip is shown in Figure-2.1.

2.2. QEMU

QEMU is a free and open-source virtualization and emulator program that may be used to run operating systems and applications on a computer in a virtual environment. It lets users to run a guest operating system (OS) on a host computer that is running a different OS or has different hardware. QEMU can emulate a variety of RISC and CISC hardware architectures, including x86, ARM, MIPS, PowerPC, and others [49]. QEMU may be used for a number of reasons, including software testing and debugging, concurrently running several operating systems, and running legacy applications on contemporary hardware. It is frequently used in conjunction with other software, such as (Kernel-based Virtual Machine) KVM or Xen, to add functionality and improve performance.

2.3. NEURAL NETWORK AND OBJECT DETECTION

2.3.1. General Terms

Machine learning is a branch of computer science that use algorithms and statistical models to allow computers to learn and improve their performance over time. It entails the use of massive volumes of data and computing power to create models capable of making predictions or judgments based on the data. There are different methodologies and approaches. The approaches and relationships are shown in Figure-2.3 [50]. The objective of machine learning is to create models and networks that can automatically improve their performance on a particular task based on experience. This is accomplished by giving a huge quantity of data to the model and enabling it to discover patterns and correlations within the data. For example, the ImageNet dataset contains over one million training images [51]. The model is then evaluated on a different set of data to see how well it performs.

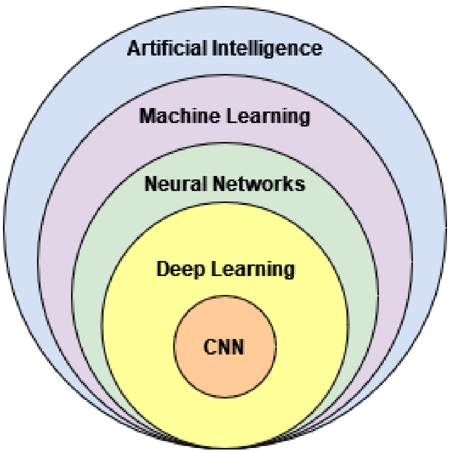


Figure 2. 3 – Relationship in Methodologies [50]

Machine learning may be approached in a variety of ways, including supervised learning, unsupervised learning, and reinforcement learning [52]. A model is trained on labeled data which contains informative tags about the datasets and clusters in supervised learning, and the proper output is produced for each input. Based on this input-output mapping, the model learns to generate predictions. The model in unsupervised learning is not given labeled data and must discover patterns and correlations in the data on its own [52]. Machine learning has various application areas [53]. However, in this research neural networks are the focused. The popular study areas of supervised machine learning are shown in Figure-2.4.

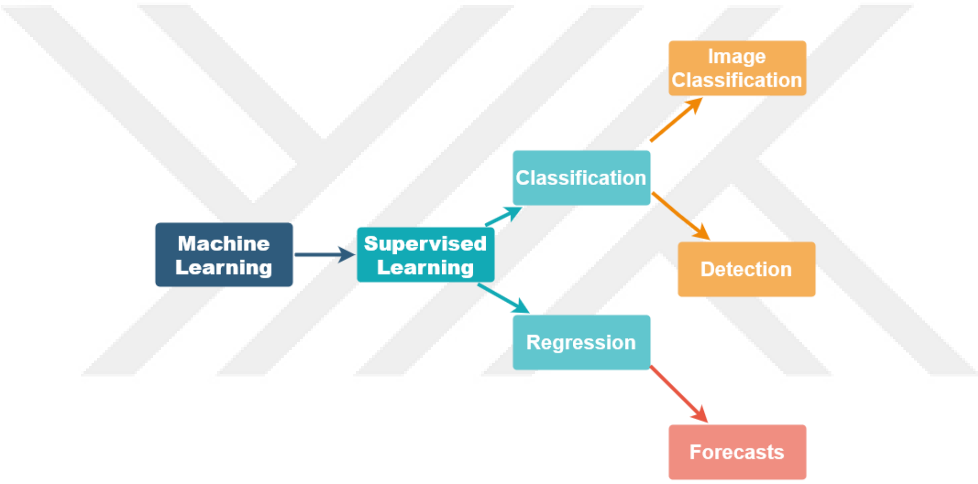


Figure 2. 4 – Supervised Machine Learning Study Areas [53]

Neural networks are supervised machine learning models inspired by the structure and function of the human brain. They are composed of layers of linked nodes, or neurons, that process and send data. Each neuron is input from other neurons and calculates an output, which is subsequently passed on to neurons in the following layer [54]. Classification, regression, and creation are just a few of the tasks that neural networks may be trained to do. One significant feature of neural networks is their capacity to learn and adapt when new data is provided to them. This enables them to gradually improve their performance on a specific assignment. They are also extremely adaptable and may be used to handle a broad variety of issues, including image and audio recognition, natural language processing, and predictive modeling. Nevertheless, there are several disadvantages to adopting neural networks. They need a relatively huge quantity of data to adequately train and might be computationally demanding to execute. They may also be biased if the training data is not reflective of the real-world data to which they will be applied.

Neurons in neural networks, like neurons in the human brain, function as calculating units. Synapses are weighted values that link two neurons. Following that, the bias value is added to the total of the weighted input values. Activation functions serve as a means for translating input to output. It decides whether or not to activate the neuron based on whether the output fulfills particular criteria or not. Activation functions are essential components of neural networks because they enable the network to simulate complicated nonlinear interactions and generate more accurate predictions. A basic forward network can be expressed in Eq. 2.2 x is the given input, y is the output, w is the weights, b is the bias and n is the number of neurons in the network [55].

$$y = \sum_{i=1}^{n} (w_i * x_i) + b \tag{2.2}$$

Multilayer perceptrons are the most commonly utilized Artificial Neural Network (ANN) algorithms [55]. Several neurons are arranged in various levels of multilayer perceptrons. Typically, the input layer just sends data along without altering anything. The majority of computing occurs in the hidden units. The hidden layer activation is converted to an output, such as a classification, by the output layer. Each hidden layer's outputs serve as the inputs for the next hidden unit [55]. The number of neurons in the output layer is the same as the number of classes taught for the neural network. Different layers perform different important operations. Convolutional Layer, Pooling Layer and Fully Connected Layer [56] are explained in detail in Section-2.3.2.

2.3.2. Layers

Convolution is the process of extracting features from input data by applying a series of filters known as kernels or weights. Each filter is a tiny matrix that is applied to a limited section of the input data, and the convolution process produces a feature map that reflects the existence of certain features in the input data. A convolution layer in a convolutional neural network is a layer that performs a convolution operation on the input data. A convolution layer may be used in image classification applications to extract characteristics such as edges, corners, and patterns from an image [57]. Features that are extracted from different layers with different convolution kernels can be shown in Figure-2.5.

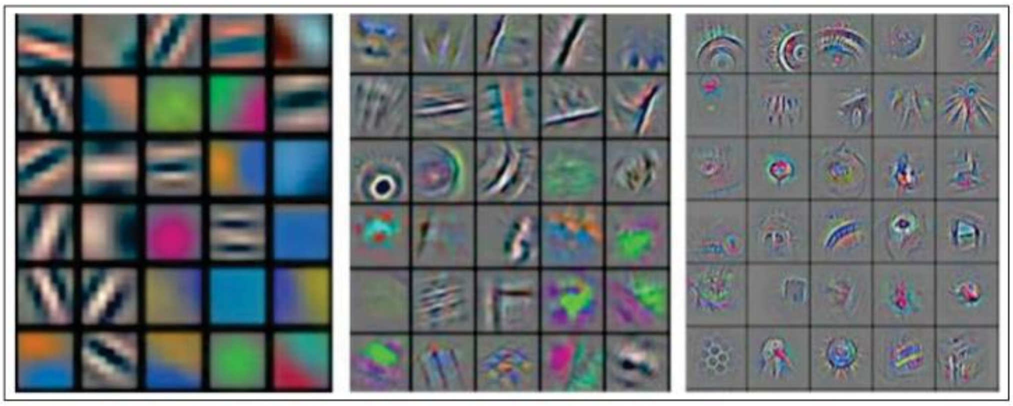


Figure 2. 5 – CNN Layers [58]

Traits and features are subsequently forwarded to the network's next layer for further processing. Convolution layers are critical components of CNNs the fact that they enable the network to learn meaningful characteristics from incoming data and utilize them to create predictions. The complete mechanism of convolution is sliding the selected kernel with stride value all over the matrix which is shown in Figure-2.6. Pooling layers are often used after convolution layers to downsample the feature maps produced by the convolution layers. This helps to minimize data complexity and improve network efficiency.

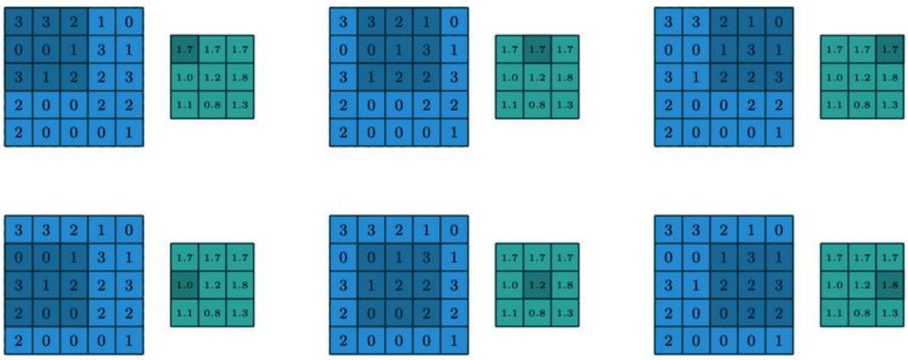


Figure 2. 6 – Convolution Layer Operations [59]

Pooling is a technique for shrinking the amount of input data by summarizing information in a limited region of the input data. Pooling processes of many sorts, such as max pooling and average pooling can be employed. A pooling layer in a convolutional neural network is a layer that conducts the downsampling of the input data. Pooling layers are commonly employed after convolution layers to minimize the size of the feature maps generated by the convolution layers. This helps to minimize data complexity and improve network efficiency. Pooling also contributes to the network's resistance to minor translations and deformations of the input data. The result of the pooling operation in max pooling is the maximum value of the items in the local region. The result of average pooling is the average of the components in the local region.

There are several types of pooling methods commonly used however; the most common are max pooling and average pooling. Max pooling takes the largest value of the matrix elements on the selected window and makes the most visible feature stand out. It is useful when the background is dark or the subject to be highlighted is brighter. On the other hand, average pooling takes the average of the values on the selected window. This gives an average result, but can also include undesirable results if the edges and borders are not sharp. The max pooling and average pooling operations are shown in Figure-2.7.

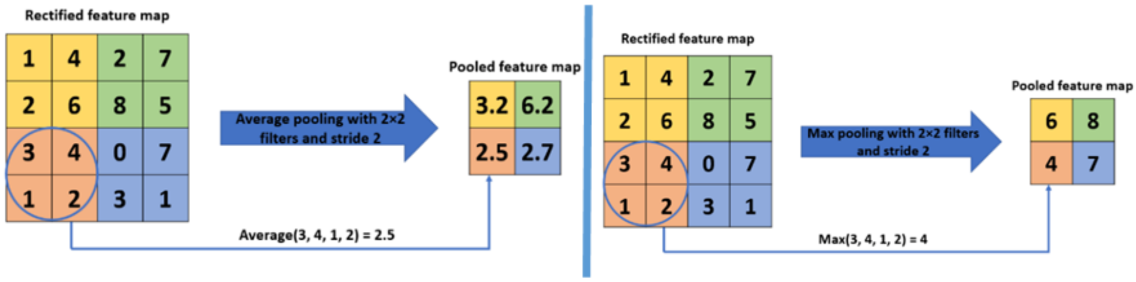


Figure 2. 7 – Pooling Operations [60]

In a neural network every layer does not have to be connected with all previous layers. A fully connected layer, also known as a dense layer, is a layer in a neural network in which all of the neurons in the layer are linked to every neuron in the preceding layer. The fully connected layer structure is in Figure-2.8. This implies that each neuron is input from all of the neurons in the preceding layer and generates an output that is passed on to all of the neurons in the following layer. At the end of a neural network, fully connected layers are often employed to create the final prediction based on the processed input data. They may also be employed in the network as intermediary layers to learn more complicated data representations [61]. Fully connected layers are an essential component of neural networks because they allow the network to simulate complicated interactions between input and output data. However, they could be computationally costly, especially if the input data contains a high number of features since each neuron in the layer must receive input from all of the characteristics.

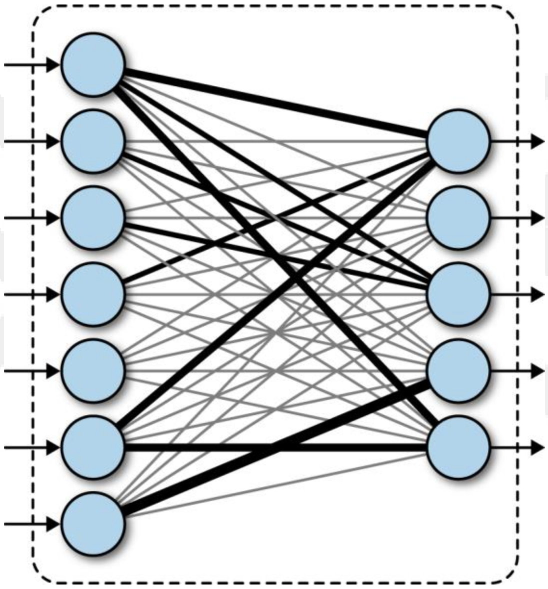


Figure 2. 8 – Fully Connected Layer [61]

2.3.3. Object Detection and Classification

Object detection and classification refers to the process of detecting and categorizing pixels into a finite set of distinct classes based on their data values and pixel attributes. Object detection is an important topic of research in computer vision and deep learning. It is a necessary step toward more advanced computer vision tasks including object tracking, event detection, analysis, and semantic behavior understanding. Object detection aims to locate the target object in an image, accurately classify it, and provide the frame for each target. That frame named bounding box. It has found widespread use in driving automation, video and image recovery, surveillance systems, medical analysis like cancer research and other sectors [62].

Traditional object detection algorithms work to extract features manually, which takes several steps. During the pre-processing stage, the dimensions of the image and its properties such as color and contrast are adjusted. In the window sliding stage, a window kernel is determined and the image is sliding all over the input image depending on the window size and stride parameters. Then, image properties are rearranged by methods such as opening, closing and binarization [63]. After that, algorithms such as Scale-Invariant Feature Transform (SIFT), Features from Accelerated Segment Test (FAST), Speeded Up Robust Features (SURF) and HOG algorithms, etc. extract features [64]. Then, the determined features are compared and classified with k-Nearest Neighborhoods (kNN), Support Vector Machines (SVM) and Maximum Likelihood Estimation methods. The traditional methodology is visualized in Figure-2.9. In this way, the object on the image given in the post-processing part is marked. Although it has difficulties such as the difficulty of working with large and complexity of images, adapting to different types of images, long processing time and the necessity to determine the window size, it has been used as traditional method for a long time.

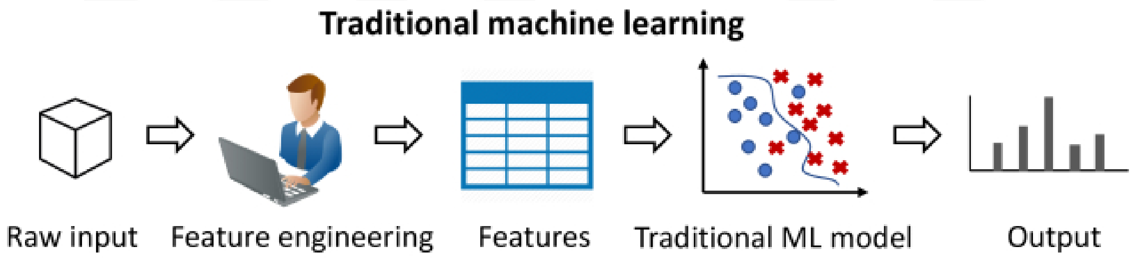


Figure 2. 9 – Traditional Object Detection Methodology [65]

On the other hand, AlexNet, gained popularity by completed the ImageNet Large Scale Visual Recognition Challenge (ILSVRC) in 2012 [66]. AlexNet is a set of deep learning and convolutional neural network-based image processing and object recognition algorithms, which are more successful than traditional methods and can be applied to all kinds of visual inputs [67]. AlexNet structure is shown in Figure-2.10. Rectified Linear Unit (ReLU) activation function [68] is used in AlexNet instead of tanh or sigmoid activation functions. The main reasons for this success and development in the field of deep learning are the increase in computing power, the preparation of high resolution and detailed datasets for training and testing the models, the increase in memory capacity, and the improvements in sensor quality. Deep learning methods, which have higher accuracy rates than traditional methods, have become applicable to more systems with minor adjustments and less human impact.

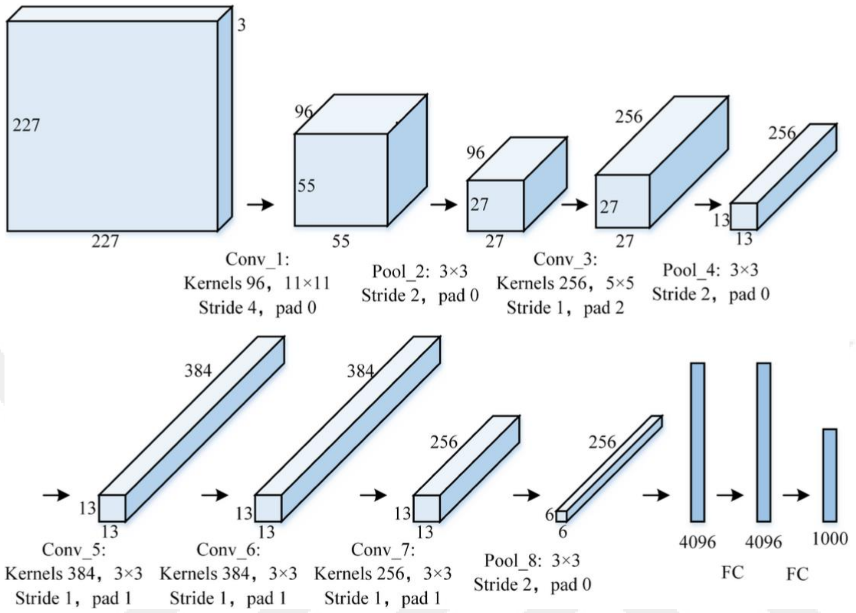


Figure 2. 10 – AlexNet General Structure [69]

These advances, which began with AlexNet, were followed by a new proposal based on the R-CNN algorithm. This technique uses a selective search strategy to find proposal areas. Each suggestion is then scaled to a given size and sent into a CNN module that has already been trained. In the last stage, the features are classified using linear SVM, and the bounding box is performed by linear regression [70]. This approach, which has made significant progress in comparison to older methods, has drawbacks such as excessive computing complexity and extended calculation time.

In 2014, VGG-Net [71] was developed, which uses many smaller kernel matrices instead of large kernel sizes. ReLU activation function is used in VGG-Net and 1x1 linear transformation filters developed with 224x224 input images were used [71]. VGG-Net’s general structure is shown in Figure-2.11. VGG-Net, which provides a great advantage over AlexNet in this aspect, has multiple fully connected layers and kernels up to 512x512 [71].

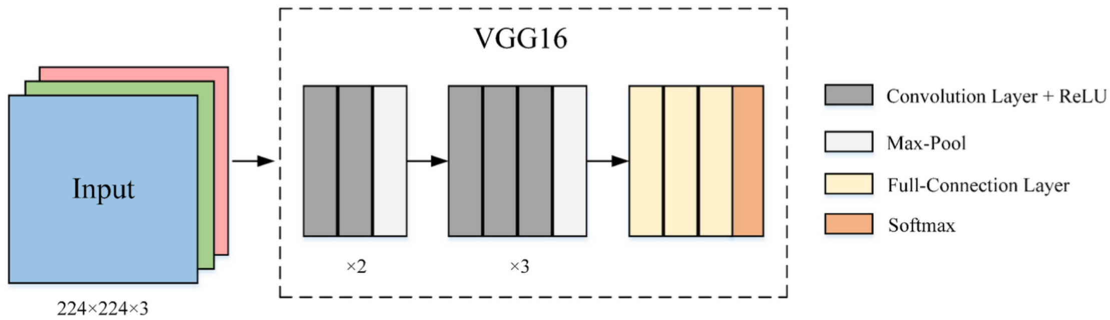


Figure 2. 11 – VGG-16 Net General Structure [69]

GoogleNet, which was designed with the InceptionNet infrastructure and launched in 2014, provides more complex features and pattern learning. Trained using over 1 million images, GoogleNet is designed using multiple low-dimensional kernels [72]. The main structure of InceptionNet block which generates the GoogleNet is shown in Figure-2.12. After the last convolutional layer, GoogleNet replaced the fully-connected layers with a simple global average pooling that averages out the channel values throughout the 2-dimensional feature map. The overall number of parameters is considerably reduced as a result.

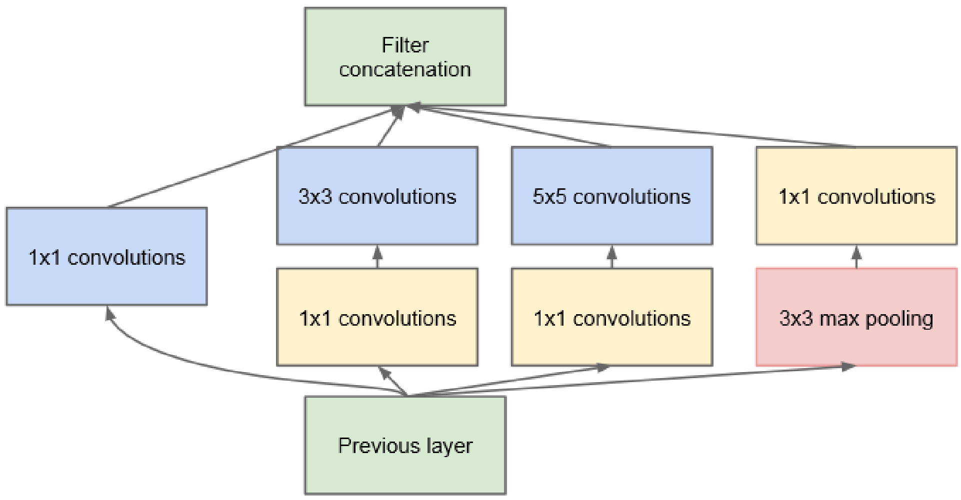


Figure 2. 12 – InceptionNet Inception Block Structure [72]

In 2015, SPP-Net [73] was introduced as a solution to the fixed input size and computational complexity problems which is one of the major handicaps of the R-CNN algorithm. On the other hand, with the spatial pyramid pooling layer added in the last layer, it is aimed to develop the feature extraction method and avoid repetitive calculations. In the Fast R-CNN algorithm, which was also developed in 2015, softmax was used instead of SVM and the number of fully connected layers was increased [74]. In the Faster R-CNN method, the region proposal method was used instead of the selective search method, while a part of the algorithm, which consists of two main parts, includes the Fast R-CNN method, the other part is a fully convolutional neural network for all region proposals. General structure of Fast R-CNN is shown in Figure-2.13.

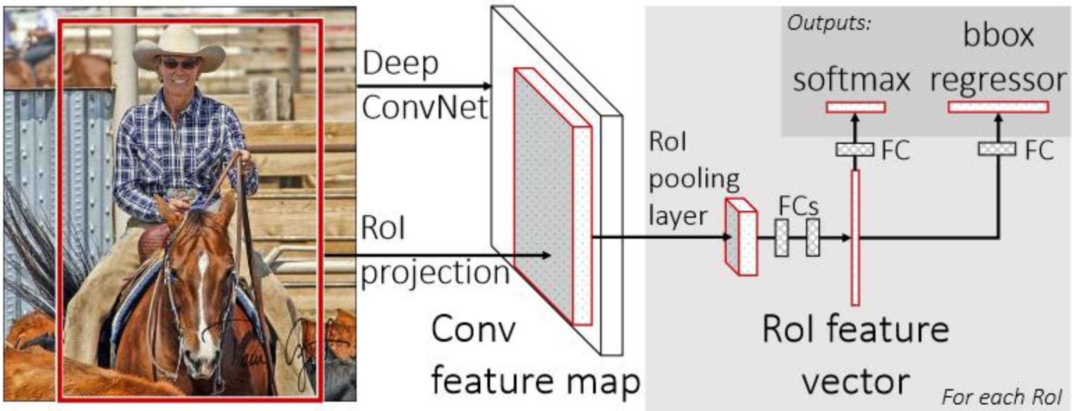


Figure 2. 13 – Fast R-CNN Layer Structure [74]

ResNet, the deep residual network algorithm, was developed in 2015 and won the ILSVRC2015 competition with an error rate of 3.6% approximately [75]. Despite having an error performance of 5% approximately of the human eye, has come to the fore with its performance compared to other deep learning algorithms [76]. The ResNet architecture, modeled after the VGG network, has less complexity and fewer filters than the VGG network. VGG-19 has a total of 137.7 million parameters in a 2-dimensional structure, while the ResNet-34 structure has only 21.5 million parameters [77]. Then, by adding jump connections and residual blocks to this flat network structure, ResNet architecture was created. The general belief in the algorithms developed up to ResNet was that the performance increased up to 20 layers, and after 20 layers, the performance did not improve due to the disappearance of the gradients and sometimes the performance decreased [78]. The reason for this decrease is not overfitting, but the saturation of the accuracy rate and the difficulty of optimization in deepening networks. It has a structure designed to create residual blocks instead of the classical algorithms that pass from the input representation to the output representation and add the result of each block to the previous one. The classical CNN approach and residual learning structure are shown in Figure-2.14 (a) and (b) respectively.

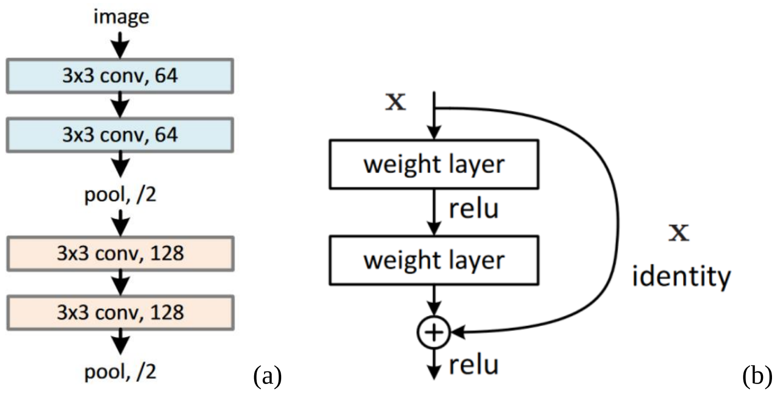


Figure 2. 14 – ResNet-34 Residual Blocks [75]

ResNet-50, on the other hand, is obtained as a result of combining the 2-layer blocks in ResNet-34 with the 3-layer bottleneck block. While there are two 3x3 convolution layers in ResNet-34, the new models designed have 1x1, 3x3 and 1x1 convolution operations again [75]. With these structures, bottleneck layers are created. ResNet-101 and ResNet-152 models were created by using more 3-layer blocks. The bottleneck blocks for ResNet-50/101/152 are shown in Figure-2.15.

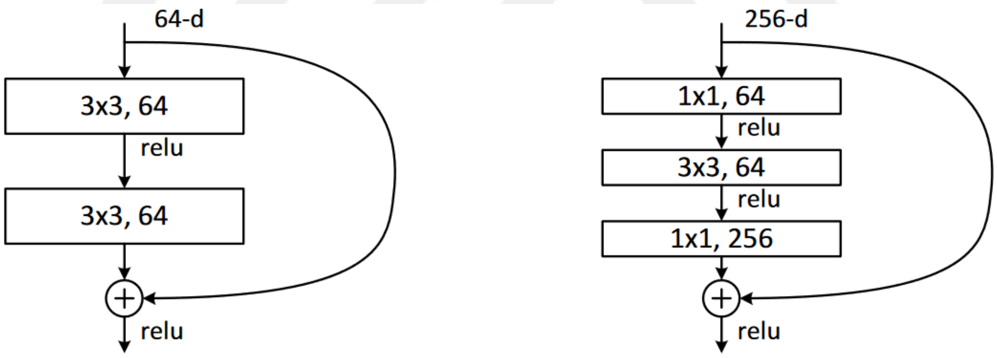


Figure 2. 15 – ResNet-50-101-152 Model Bottleneck Blocks [75]

One of the important problems faced by deep convolutional neural networks is the loss of gradients (vanishing-gradient problem) and degradation, which limits the learning method and the number of layers. While ResNet has overcome this problem with residual blocks, DenseNet is designed to solve this problem [79]. In DenseNet, each layer in DenseNet links to every other layer in a feed-forward method. As a result of this, each layer has directly reaches to the gradients of the loss function and the originated version of the input signal, resulting in insinuating deep supervision. AlexNet and VGG-Net approaches have “N” number of layers and there are “N” connections between these layers but in DenseNet there are $\frac{N(N+1)}{2}$ layers [79]. The dense blocks and layer structure are shown in Figure-2.16 and Figure-2.17 respectively. The feature-maps of all previous layers are utilized as inputs for each layer, and its own feature-maps are used as inputs for all following levels. The benefits of DenseNet include solving the vanishing gradient problem, improving feature propagation, increasing feature reuse, and drastically reducing the number of parameters. [79]. DenseNet outperform the bulk of state-of-the-art, while requiring less computation to achieve a good performance.

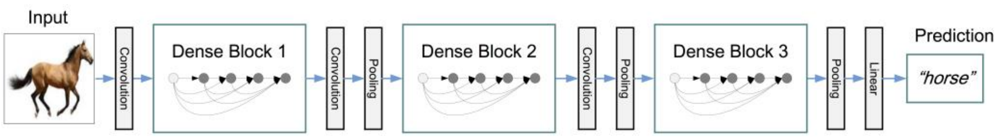


Figure 2. 16 – DenseNet, Dense Block Diagram [79]

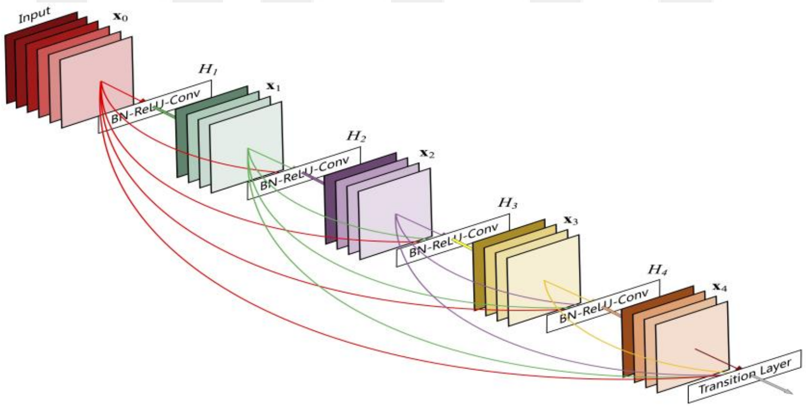


Figure 2. 17 – DenseNet General Algorithm [79]

2.4. Advanced eXtensible Interface and XILINX Zynq Architecture

AXI bus is Advanced Microcontroller Bus Architecture (AMBA) based interface [80]. It was introduced with the third revision of AMBA and now the fourth revision is currently in use. It is an open standard that specifies how to connect and manage the various components or blocks of a SoC system. AXI interface basically consists of two main blocks named master and slave. Read and write operations also consist of channels that are independent of each other as address or data channels. Address transfer and control information is sent via address channels and it is checked that the communication between the main components is working properly. In the AXI interface, the master initiates all operations and the slave only returns when necessary. Data transfer, which seems to be unidirectional, is carried out bidirectional due to control signals, identification signals and ready signals. The AXI structure, and the directions of the signals connected to the processes are shown in Figure-2.18. As it seen from the Figure-2.18, all processes are initiated from master. [80]

Figure 2. 18 – AXI Master Slave Bus Structure

In XILINX Zynq architecture, AXI method is responsible for communication between Programmable Logic (PL) and Processing System (PS) units. In general, a rule set must be defined in the FPGA that will enable PI blocks to send data to and receive data from the outside world. In XILINX Zynq architecture, this rule set is designed based on the AXI method. Again, as in the normal AXI method, all read and write transactions are initiated by the master and the necessary responses for the transactions are provided by the slave. The transmission of data between master and slave is done by a more simplified method called stream or by a more complex method called Memory Interface [80] [81]. The general general AXI structure In XILINX Zynq is shown in Figure-2.19. HP components are high performance parts and work with Memory Interface structure, not Stream. While two of four high performance units can be connected directly to Dynamic Random Access Memory (DRAM), the remaining two of them are connected to AXI Interconnection first. AXI Interconnection, connects the AXI blocks to rest of the architecture [81].

Additional to the high performance blocks, with Accelerator Coherency Ports (ACP) in PL unit, connected to ASYNC block for asynchronous clock before connect to Snoop Detector in PS unit. The snoop detector provides access to On-Chip Memory (OCM) and CPU Cache units. AXI connections reaching caches can access peripheral devices via General Purpose AXI Slave Interconnection, while also accessing DDR Controller unit through Quality of Service (QoS) and ASYNC block. [81]

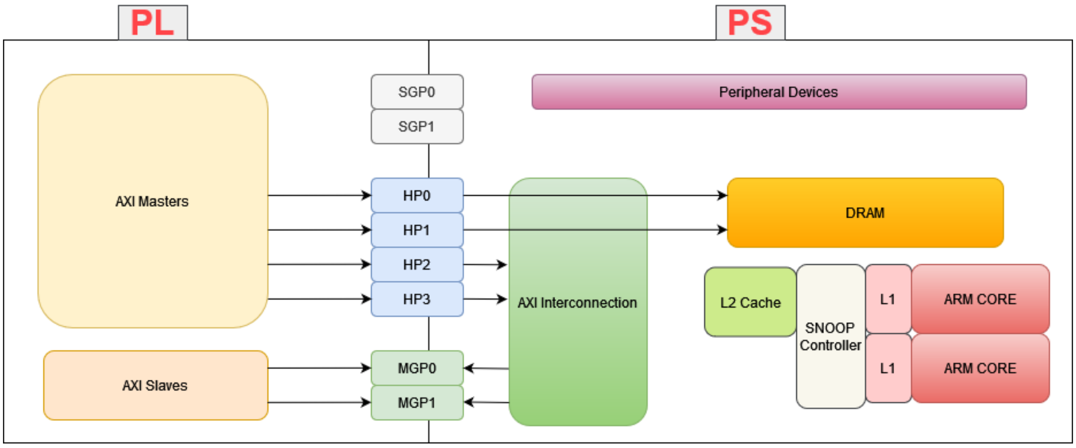


Figure 2. 19 – AXI Structure In XILINX Zynq Architecture

The bus structure of the AXI model denoted as ARX and AWX. “A” denotes the AXI structure and is found in the name of all buses. “R” and “W” specify the Read and Write operations, respectively. Finally, “X” replaces the functions between master and slave. For example, in address writing, the VALID signal goes from the master to the slave, but in the read response, the direction of the VALID signal is opposite. Again, as mentioned above, even if the direction of the signals changes, all transactions are initiated by the master.

The basic structure of Xilinx Zynq architecture is shown in Figure-2.20. Accordingly, the AXI interface is responsible for the data transfer between the PL and PS sections. The DPU is located in the PL section of the Xilinx card, and all externally accessed peripherals and ARM cores are located in the PS section. All these sections are connected to the AXI interface and communicate with each other.

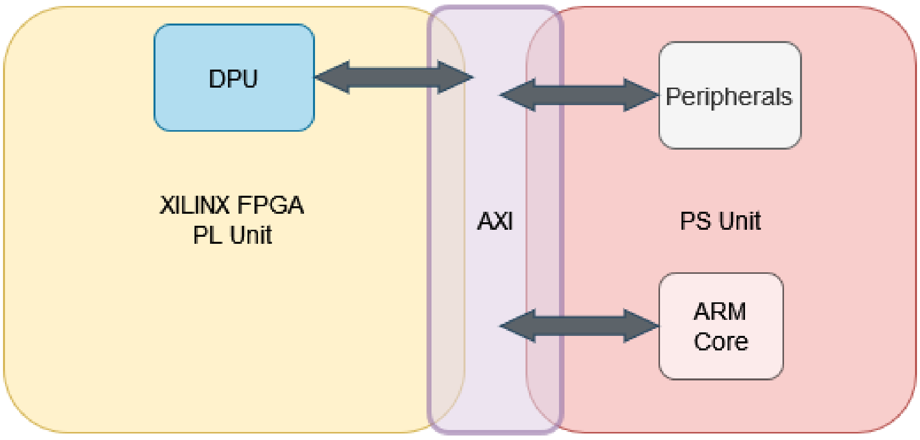


Figure 2. 20 – Xilinx Zynq PL, PS and AXI Relationship Diagram

3. EXPERIMENTAL WORK

In this part of the study, RISC-V architecture was implemented in different configurations and image processing neural network applications were run. In this section, first, image processing applications were applied with the Chipyard SoC Generator, then the Gemmini accelerator was implemented on the same configuration and the experiments were repeated. Then, a Linux operating system was run on the RISC-V kernel with the QEMU emulator, and image processing and neural network applications were executed. Later, OpenCV libraries were installed on the Linux operating system under the RISC-V ISA and a full network test was provided with the cross compiling method. The results obtained were compared in terms of time, the competing architectures were evaluated in Section-4 in terms of usability and effectiveness.

3.1. RISC-V ISA OPERATIONS

In this portion of the research, various studies employing RISC-V ISA are shown. The major goal of this section is to execute image processing and, if feasible, neural network applications on the Linux operating system by running alternative RISC-V kernels built on the same RISC-V ISA architecture as softcore. The performance and interoperability of RISC-V cores produced with various modifications were thus examined. The versions of the operating systems and other libraries used for these processes are shown in Table-3.1.

Table 3. 1 – RISC-V ISA Prerequisites

System	**Version**
Operating System	Ubuntu 20.04
Operating System	Fedora Rawhide 34
OpenCV	v4.6.0
RISC-V GCC Version	riscv-gcc-12.2.0-branch
GNU C Compiler (GCC) Version	12.2

GNU Debugger (GDB) Version	12.1
Python Version	3.9
Vivado	Vivado ML Edition
QEMU	v7.0.0
MATLAB	2021b

Ubuntu 20.04 system was installed as the first stage of the study. The reason for using the Linux operating system instead of Windows is that RISC-V architecture enables better management of library dependencies and development as open source. Ubuntu was preferred for its ease of use. Next, the prerequisites and GNU's not Unix (GNU) Compiler Toolchain required to run the RISC-V instructions are installed by following the steps on the official GitHub [82] page. The important point here is that the extensions we will use should be installed, but not with the default configuration. The detailed installation procedure is shown in Appendix-4. Furthermore, the 64-bit Newlib and Linux versions provided by the RISC-V ISA Compiler Toolchain were installed. Executable and Linkable Format (ELF) GCC installed with Newlib may run executable files. Linux GCC, on the other hand, is intended for libraries and executable files that will operate on the Linux operating system [83]. C and C++ languages can be executed with both ELF GCC and Linux GCC, as seen in Appendix-4. Although their execution processes and usage domains differ.

In this section, where the first layer of the network similar to VGG-19 structure is simulated, the sample application is run as in Figure-3.1 and the time results are compared with each sub-sections results.

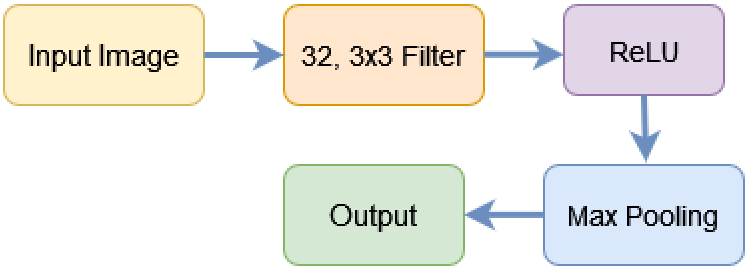


Figure 3. 1 – Sample Flow

3.1.1. Installing Chipyard RISC-V ISA

In this section, RTL Generators named Rocket Core and BOOM Core were created using the Chipyard Framework. While Rocket Core has an in-order RISC-V core structure, BOOM Core has an out-of-order RISC-V core structure. After configuring these created systems, it is aimed to perform matrix operations of filters and other operations used in Section 3.2.1 of the study by using the matrix multiply accelerator named Gemmini [32]. For this purpose, changes have been made to the Chipyard configuration files. On top of the basic “riscv-tools” toolchain collection with assembly toolchains installed and configured, the “esp-tools” system Chipyard made it possible to use the ISA simulator, Bootloader and non-standard RISC-V extensions [84]. Verilator was used for RTL simulation [85, 86]. Chipyard tools and Verilator installation is shown in Appendix-4.

After installing Chipyard, for this research a custom configuration was created in the Chipyard SoC. This special configuration named “AnilConfig” which is represented in Figure-3.2. Hereunder, this created configuration consist of a BOOM Core (out-of-order) and a RocketChip core (in-order). This configuration will be used in full system simulation.

```
package chipyard

import freechips.rocketchip.config.{Config}

// ---------------------
// Heterogenous Configs
// ---------------------
class AnilConfig extends Config(
  new boom.common.WithNSmallBooms(1) ++
  new freechips.rocketchip.subsystem.WithNBigCores(1) ++
  new freechips.rocketchip.subsystem.WithInclusiveCache(capacityKB=256) ++
  new chipyard.config.AbstractConfig)


  // DOC include start: DualBoomAndSingleRocket
class DualLargeBoomAndSingleRocketConfig extends Config(
  new boom.common.WithNLargeBooms(2) ++                    // add 2 boom cores
  new freechips.rocketchip.subsystem.WithNBigCores(1) ++  // add 1 rocket core
  new chipyard.config.AbstractConfig)
// DOC include end: DualBoomAndSingleRocket

class LargeBoomAndRocketWithControlCoreConfig extends Config(
  new freechips.rocketchip.subsystem.WithNSmallCores(1) ++ // Add a small "control" core
  new boom.common.WithNLargeBooms(1) ++                     // Add 1 boom core
  new freechips.rocketchip.subsystem.WithNBigCores(1) ++   // add 1 rocket core
  new chipyard.config.AbstractConfig)
```

Figure 3. 2 – Chipyard Heterogeneous Configuration

In the Verilator folder, the configuration is compiled with following commands. In this research, the new configuration is created with “AnilConfig” name and it contains a Rocket Core and a BOOM Core. The BOOM Core has a faster processor than Rocket Core, due to out-of-order structure; nevertheless, Rocket Core is included in the configuration for both AXI interface connection and Gemmini integration.

```
$ cd sims/verilator
$ make debug CONFIG=AnilConfig
```

With regard to custom configuration operation, the new configuration involves two different cores. After this stage, the sample image selected from the ImageNet dataset reshaped as a square matrix and converted to grayscale, can be used as input for RISC-V. Executing the image convolution C code with RISC-V ELF-GCC is shown in Appendix-4 section.

The C file written with this process was converted into an object file with the extension “.o” created by the compiler. The “.o” file is converted to an executable file format using RISC-V ELF GCC. In order to create executable files, the outputs of the codes can be observed with the bare-metal simulator called Spike, which was installed together with Chipyard.

In order to simulate on Verilator with “AnilConfig” configuration, the “conv.riscv” executable file is executed. To reach the time result of this operation, the prefix “time” has been added to the beginning of the console code. First of all, Spike is used for checking the results of matrix convolution operation without timing analysis and detailed RTL simulation. Spike simulator is used only to show that the architecture and the written codes are working correctly. Spike does not provide a full RTL simulation. Since the softcore processor is running on the instructions of the real processor, the time results can be compared and does not show the time performance of the processor running on the software. In this study, Spike is used only to demonstrate the operability of the architecture.

```
$ time ./simulator-chipyard-AnilConfig-debug conv.riscv
```

Through this configuration, a full RTL simulation is provided on the RISC-V core produced with the Chipyard SoC Generator.

```
$ time ./simulator-chipyard-AnilConfig-debug conv.riscv –V –vconv.vcd
```

After running of the command, a ".VCD" file was created using the "V" prefix and the clock signals and program counter were monitored with the GTKWave waveform simulator. The complete RTL simulation output is observed with a custom Verilator configuration, which is observable in Appendix-4. It is observable that the system that the outputs of the spike and Verilator simulators are the same. After the matrix convolution operation, scalar dividing, ReLU operation and max pooling operation are performed. In that manner, the first layer of the neural network is applied. The complete C language codes are located in Appendix-1.

On the other hand, when the Chipyard SoC Generator was downloaded, the same experiment was repeated with the Gemmini accelerator whose installation files were also downloaded, and the time results were noted. With the possibilities such as Systolic array and SRAM connections, data compression, parallel processing, operations related to matrices can be accelerated with Gemmini. It has been observed that normal matrix multiplications are accelerated between 60% and 70% compared to a softcore processor produced with the Chipyard SoC Generator. With Gemmini, the first layer of a network similar to VGG-19, which includes matrix convolution, ReLU and Pooling operations, was run and time results were obtained. For the full RTL simulation, the given code is executed and the program counter and the all registers of the RISC-V ISA and the sample code and the states of the PC and registers are shown in Appendix-4.

```
$ time ./simulator-chipyard-CustomGemminiConfig –V –vcode.vcd code.riscv
```

The systolic array only supports operations up to the specified limit value, and the input matrix limit is given as 16 in the "params.h" file, Gemmini is designed to operate on up to 16x16 matrices at one time. Accordingly, the used 100x100 input was fed to Gemmini with the tile mechanism. As already mentioned, it is thought that there will be a decrease in performance when the input image is fed with the tile system in this way, in the Gemmini architecture, which has a 60% to 70% acceleration compared to the normal softcore processor created with Chipyard.

Spike simulator is used for Chipyard SoC generator for system and configuration check only. Spike uses the processor of the computer on which the code is running, using the RISC-V ISA produced. Thus, when creating a complete cross-compiling environment, it does not use real RISC-V cores. Since there is an AMD Ryzen 5 Pro 4650U 2.10GHz (Gigahertz) processor on the computer with the Linux operating system on which the RISC-V ISA is installed, the processing time was much shorter than Verilator, which includes a full processor and RTL simulation and was taken as the basis for the actual simulation results. Experiments carried out on Chipyard are shown in Figure-3.1 For this, a sample image from the ImageNet dataset is used. The results of Chipyard SoC Generator and Gemmini Accelerator are shown if Table-3.2 and Table-3.3 respectively.

Table 3. 2 – Chipyard SoC Generator Initial Test Results

Architecture	**Operation**	**Time Result**
1 RocketCore 1 BOOMCore	Convolution with 32, 3x3 Kernel ReLU Operation Max Pooling Operation	134 min 46.237 sec

Table 3. 3 – Chipyard SoC Generator and Gemmini Initial Test Results

Architecture	**Operation**	**Time Result**
1 RocketCore 1 BOOMCore Gemmini Accelerator	Convolution with 32, 3x3 Kernel ReLU Operation Max Pooling Operation	92 min 76.383 sec

3.1.2. Installing RISC-V ISA with QEMU and Linux

After observing that processing on large matrices using Chipyard SoC Generator is not suitable in terms of time performance, it is aimed to bring a solution to the targeted system by using RISC-V ISA in another simulation environment. For this reason, after installing RISC-V Toolchain in a Linux environment, RISC-V Proxy Kernel was installed and QEMU was used to perform bootstrap on the RISC-V ISA simulator and bare-metal. Since Spike can only perform bare-metal simulations and simulations that require complex or visual interfaces cannot be performed on bare-metal, a kernel cross-compiling process has been performed using the RISC-V ISA. For this, the kernel compile program BusyBox was used and it was aimed to perform simulations by installing a Linux Arch-based operating system on a virtual RISC-V kernel.

Fedora, a Linux Arch-based system, has been made workable on the RISC-V architecture. It enabled development and cross-compiling in Python and C++. Developed for an experimental study, it is designed with a virtualized single-core RISC-V processor and unlimited RAM permission for the Spike simulator on RISC-V-based Fedora. In this designed system, display ports are not added.

As can be seen, the Linux operating system can run on RISC-V and the RISC-V architecture is built with the RV64ACDFIMSU structure. This means that 64-bit RISC-V core has “A, C, D, F, M, S, and U” extensions. In other words, the created RISC-V core can perform atomic instructions, compressed instructions, floating point instructions with single and double precision and supervisor level operations. The most important point seen in this section is that Vector extension cannot be included on RISC-V Linux.

After observing that the Linux operating system can run on the RISC-V processor, it is necessary to install libraries that can run image processing and neural network applications with the “pip” Python library installer [87]. Since the numpy and Pillow libraries are essential for matrix operations, which has more than one dimension. These libraries are installed firstly with pip installer. Due to the inability to add vector extensions, cross-compiling libraries compiled during Linux installation could not be built for vector instructions, and neural network libraries could not be installed on Python. As a result, creating a comprehensive neural network structure is unachievable.

For image processing and neural network operations, numpy, mathplotlib, opencv and tensorflow libraries initially attempted to be installed. Due to the problems in the establishment of libraries other than Numpy and Pillow, the operability of the system was tested by running the DNNTestNetwork script on Fedora and it was understood that the system could not work in harmony due to the test with errors. There are some errors in working with Darknet, You Only Look Once (YOLO), VGG16 Network, AlexNet, MobileNet, SqueezeNet, Fast/Faster R-CNN, etc. Due to failure on DNN and RISC-V Fedora version is not developed for full functionality, and the fully connected layers are not applicable. The reason for this seems to be that the vector extension of RISC-V cannot be included in the processor.

It has been tested with the application in Figure-3.1 for the operability and simple comparison of the installed architecture. Thus, the capabilities of the architecture and simple time analysis have become possible. The time result and details of the experiment are given in Table-3.4.

Table 3. 4 – RISC-V and QEMU Linux Initial Test Results

Architecture	Operation	Time Result
QEMU Fedora OS RV64ACDFIMSU	Convolution with 32, 3x3 Kernel ReLU Operation Max Pooling Operation	70.152 sec

3.1.3. Installing RISC-V ISA with OpenCV

OpenCV is an open source library designed for computer vision, image processing and machine learning algorithms. With OpenCV, it is possible to accelerate, run and improve Computer vision and machine learning operations. This acceleration requires instructions that are already supplied by next-generation processors, such as single input multiple data instructions and vector instructions. Although the RISC-V architecture that is planned to be used is different from the normal CPU structure, it is thought that RISC-V and OpenCV can work together with the vectorization cross-compiling platform called Wide Universal Intrinsic and inline functions used by OpenCV.

In order to perform the cross compilation operation with RISC-V ISA, the RISC-V ISA Toolchain, QEMU ISA Simulator and Low Level Virtual Machine (LLVM) are installed. LLVM is a program, which is used for different toolchains to work, and this time RISC-V Frontend Server instead of RISC-V Proxy Kernel, unlike Section-3.2.1. Since RISC-V Frontend Server is given over Spike in the RISC-V ISA version used, the Spike simulator was installed again to see that the cross compiler processes were working. The toolchain part is built with the RV64IMAFDC and RV64GCVZFH architecture, which means the 64-bit RISC-V ISA can handle integer, single and double precision floating point, atomic and compressed operations. After the installation of prerequisites, it can be installed according to OpenCV extensions. For the installation, the installation interface called CMake was used and the OpenCV libraries were directed to RISC-V in order to run C and C++ languages, and the installation was made according to RISC-V's C and C++ code compiler according to the cross compile method. The vector extension of RISC-V must be operable in order to implement a complete neural network application, Hardware Acceleration Layer and Universal Intrinsic have been installed with OpenCV in this part of the study. In addition, the C and C++ compiler of the OpenCV selected as RISC-V ISA Toolchain C and C++ cross compiler that is installed as Linux and glibc library mode which is demonstrated in Appendix-4.

Although a real or virtual RISC-V processor is not created in this section, it has been tested with the application included in Figure-3.1 for the operability of the installed OpenCV libraries and for simple comparison with other systems. Thus, the library's capabilities and simple time analysis have become possible. The time result and details of the experiment are given in Table-3.5.

Table 3. 5 – RISC-V and OpenCV Initial Test Results

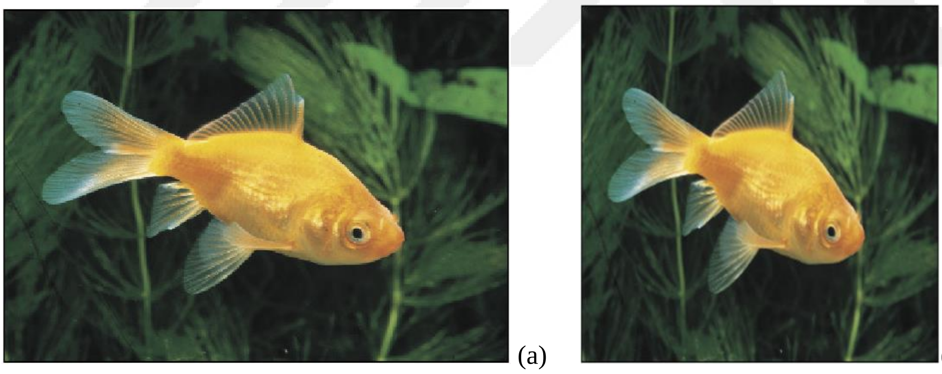
Architecture	**Operation**	**Time Result**
RISC-V ISA Linux, Newlib, ELF, GCC OpenCV	Convolution with 32, 3x3 Kernel ReLU Operation Max Pooling Operation	6.619 sec

3.2. NEURAL NETWORK AND IMAGE PROCESSING OPERATIONS

3.2.1. Single Object Detection with a Traditional Improved Method

The main topic covered in this section is determining the location of the main object in an input image. Since the images are actually matrices and easy operations with matrices, the main method was studied on Matlab. Today, when the developments in the processor and memory units of newly produced, smart phones have slowed down; manufacturers have focused on developing the peripheral units of smartphones. In this respect, cameras have been the most studied parts of smartphones. Due to these advances, cameras can produce images with very high resolutions. In this part of the study, it is aimed to determine the main object in an input image, to take the area where the object is located into a bounding box, and to process only this bounding box in the next stages. Thus, it is aimed to increase the performance, as the size of the matrix to be processed will be reduced. In addition, when the input image is reduced to relatively small input sizes such as 224x224, which is used by artificial neural network applications, it is aimed that the details of the objects are lost the least and the accuracy rate is increased by this way.

In this section, firstly, images of different sizes are resized to a square size for ease of processing. 600x600 was chosen for the resized size since the dimensions of the studied photographs are generally large and the minimum pixel dimensions used today are 800x600. ImageNet dataset and my own photographs were used to develop and test the system. Figure-3.3.a shows the original image and Figure-3.3.b demonstrates the resized image.



(a) (b)

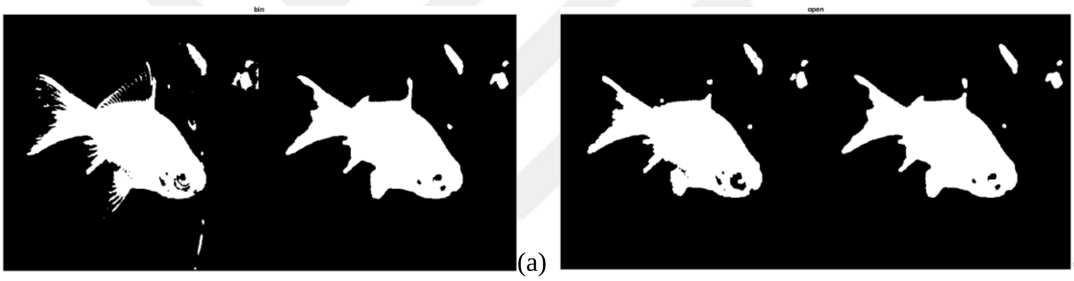
Figure 3. 3 – Input image in (a) and resized image in (b)

Then, since working on three channels such as red-green-blue (RGB) causes complexity problems in the visuals and it is aimed to facilitate the operations, the visual was moved to the grayscale area and passed through filters to soften the sharp edges and average the values. Figure-3.4.a shows the grayscaled image and Figure-3.4.b shows the filtered images. The right side of the filtered image is used for further operations.

(a) (b)

Figure 3. 4 – Grayscaled image in (a) and filtered images in (b)

In the advanced stage, the image, which has softened edges by passing through filters, is converted to binary format and turned completely black and white, and then the gaps in the image are filled by opening/closing. The results of these binary and erosion/dilation operations can be seen in Figure-3.5.a and Figure-3.5.b respectively.



(a) (b)

Figure 3. 5 – Binary images in (a) and eroded/dilated images in (b)

After this stage, the general histogram graphic of the image has been taken into account so that the background is dark or light, or the color of the object to be detected is light or dark compared to the background. According to this method, if the histogram values are concentrated on the right side of the histogram graph, the binary pixel values are inverted. On the other hand, if the histogram values are dense on the left side of the graph, the image remains unchanged. With this method, the position of the image can be determined independently of the background and detected object. As can be seen from Figure-3.6, the input image is not inverted due to the histogram values.

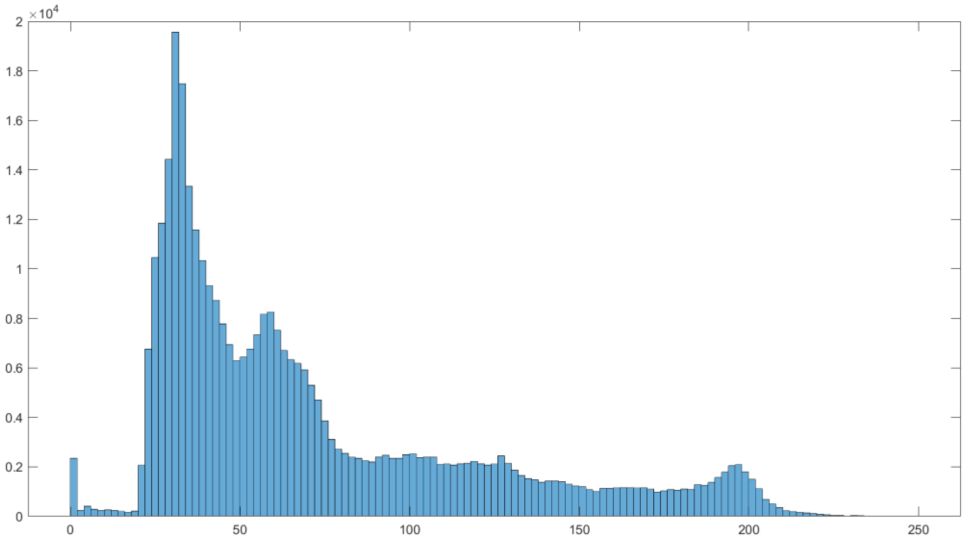


Figure 3. 6 – Histogram Map of the Image

After the histogram process, the excess white or black dots in the image are filled in according to the result of the histogram value. In Figure-3.7, the changed or fixed image and the part where the white or black parts are filled can be seen.

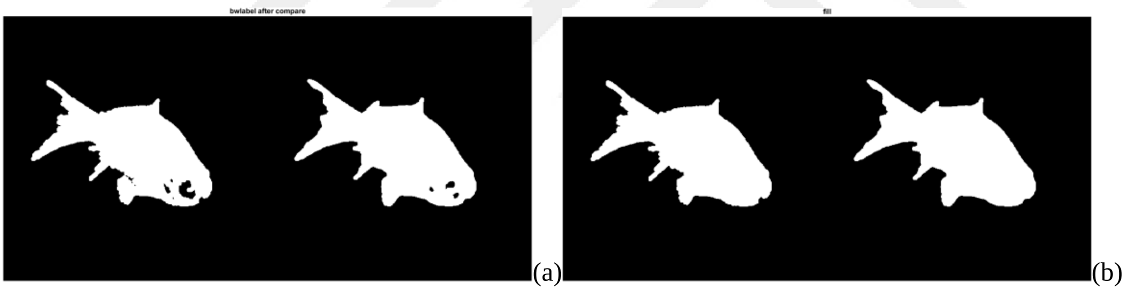
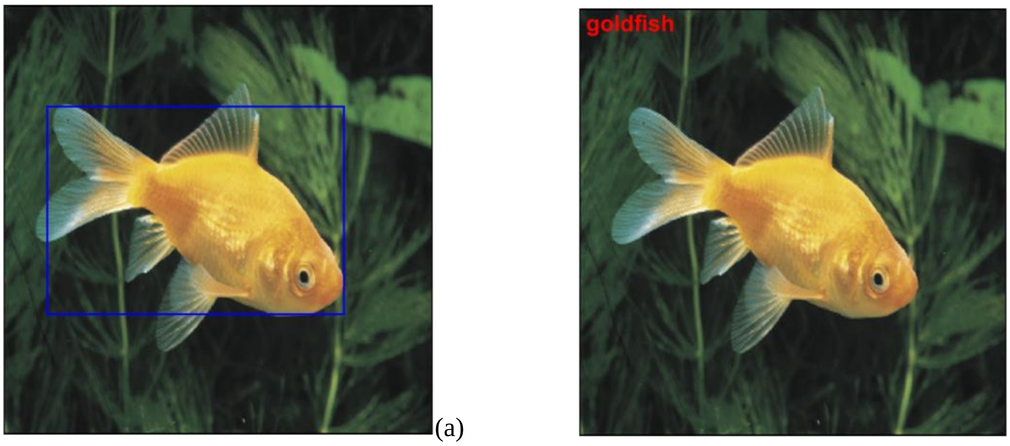


Figure 3. 7 – Reversed or unchanged images in (a) and filled images in (b)

Consequently, all that needs to be done is to insert the image that is different from the background into the bounding box, then cut this part of the image, and process it. Then, this part of the image is given to the neural network algorithm and the result of the classification is printed on the image at the beginning of the process. In the Figure-3.8.a part, the part taken into the bounding box and the result of the classification are seen in Figure-3.8.b.



(a) (b)

Figure 3. 8 –Bounding Box in (a) and Classification Output in (b)

The complete Matlab codes used in all operations may be accessible from the Appendix-1 section. The comparison of the dataset created on Matlab and the neural network applications trained with ImageNet in the Deep Learning Toolbox and the comparison prepared in terms of time and accuracy are given in Table-3.6. According to the data in the table, although there is a 3.8% delay in time, an 8.4% improvement in accuracy has been recorded. Although improvements can be seen in time outputs compared to classical methods, a time improvement could not be achieved because the networks used in Deep Learning-based object detection and classification applications are complex and powerful.

Table 3. 6 – RISC-V and OpenCV Initial Test Results

Classical DNN Based Object Detection	**Histogram Supported DNN Based Object Detection**
30.237 sec	31.412 sec

3.2.1.1. Single Object Detection with QEMU Linux and RISC-V

In this section, traditional object recognition algorithms that run with Linux working on softcore processor created with RISC-V ISA. The main purpose of this section is to evaluate the time performance of processors created using RISC-V ISA, both according to traditional methods and according to neural network methods. On the other hand, using the numpy and Pillow libraries, it is aimed to perform the convolution operation of the input image which is resized to the square matrix form and converted to grayscale form. The printing of the output is not included in the time measurements. In this section, an object detection application is executed and the time results are compared with the other results. For this experiment, a sample image taken from the ImageNet dataset is used. The time results of the application with QEMU Linux and RISC-V is shown in Table-3.7.

Table 3. 7 – RISC-V and OpenCV Initial Test Results

Architecture	Operation	Time Result
QEMU Fedora OS RV64ACDFIMSU	Single Object Detection Histogram Improved Method ImageNet Dataset Input Image	33.212 sec

3.2.1.2. Single Object Detection with OpenCV and RISC-V

In this section, traditional object recognition application is implemented with RISC-V ISA and OpenCV, which was established according to the cross compiling method. The purpose of this process is to measure the time performance of RISC-V in traditional and neural network-based object detection applications, both on softcore processors created and libraries installed with cross compiling. By using these methods, it is aimed to find the best practice method in which RISC-V can be used. OpenCV v4.6.0 is installed and operated with QEMU and installed 64-bit RISC-V ISA. After installing the OpenCV and modules, the sample image filtering C++ code is compiled with RISC-V cross compiler toolchain. The output timing results and matrix results are shown in Appendix-4. In this part of the study, after converting the sample image from ImageNet to square matrix size, image filtering is applied on a sample kernel. In this section, an object detection application is executed and the time results are compared with the other results. For this experiment, a sample image from the ImageNet dataset is used. The time scores of the application with OpenCV and RISC-V is shown in Table-3.8. The main goal in this section is to create a single object detector similar to the Histogram of Gradients (HoG) [88] approach, which combines histogram values with gradient vector calculations.

Table 3. 8 – RISC-V and OpenCV Test Results

Architecture	Operation	Time Result
RISC-V ISA Linux, Newlib, ELF, GCC	Single Object Detection Histogram Improved Method	

OpenCV	ImageNet Dataset Input Image	4.714 sec

3.2.2. Object Classification with Neural Network Methods

In the previous section, traditional object recognition applications were applied using RISC-V ISA. In this section, it is aimed to implement neural network-based object detection and classification applications using RISC-V ISA. The main purpose of the study in this section is to run a neural network supported object detection and classification application with the RISC-V methods that are used in the previous sections. In this section, it is planned to show that neural network applications run on RISC-V architecture are executable. In the first part, neural network supported object detection and classification application has been implemented on the RISC-V kernel architecture that is run on Linux with QEMU. After this experiment, neural network supported object detection and classification application has been implemented on the architecture that is built with OpenCV and cross-compiling method. The outputs obtained with the Chipyard SoC Generator and Gemmini are not included in this part of the study because they are seen as very slow compared to the time outputs of today's processors and running external libraries is a problem.

3.2.2.1 Object Classification with Neural Network, QEMU Linux and RISC-V

In Section 3.1.2, Fedora Linux was installed on the RISC-V kernel with the QEMU emulator and image processing in Python language was possible. In this part of the study, it is aimed to implement a neural network application on Linux using the Python language. In this context, TensorFlow, Keras, Pytorch and Mathplotlib libraries, which are frequently used in neural network operations, have been tried to be installed. However, since QEMU Linux could not be installed to support the vector extension of the RISC-V ISA and these libraries could not be installed with the Python installer “pip”, the classification part of the deep learning based object recognition application could not be realized. Nevertheless, the VGG-19 network, which is a network that does not contain parallel structures and is easy to implement, has been implemented without classification part. The time output of the VGG-19 Network without classification layer is shown in Table-3.9.

Table 3. 9 – QEMU Linux with RISC-V VGG-19 Network Results

QEMU Linux with RISC-V VGG-19 Network	QEMU Linux with RISC-V VGG-19 Network without Classification Layer
Classification layer cannot be implemented	601.88 sec

3.2.2.2 Object Classification with Neural Network, OpenCV and RISC-V

In Section 3.1.3, it has been seen that OpenCV, which was installed using the RISC-V ISA with the cross-compiling method, can run image processing applications if the necessary libraries are included. In this section, the necessary libraries are included and it is shown that the object detection application can work on C++. Object recognition application was carried out on OpenCV by using GoogleNet, which is a network that is high in accuracy and faster than other networks in terms of time performance.

The purpose of the work done in this section is not to train a neural network on a given dataset, but to decide with a trained model's weight file and the input image. Therefore, the neural network-based object detection code is run with the cross compiler "riscv64-unknown-linux-gnu-g++" was first written in C++ with OpenCV and the all of the dependent libraries. The locations of the dependent libraries and header files are given to the cross compiler to enable the C++ code to run. Nvidia Compute Unified Device Architecture (CUDA) driver is installed to perform DNN operation even the main system does not contain Nvidia GPU. The dependent libraries are written in the terminal codes, the details and outputs are given in Appendix-1 and Appedix-4. The time output of the full GoogleNet implementation is shown in Table-3.10. The result of the OpenCV object detection and classification is shown in Figure-3.9.

Table 3. 10 – OpenCV Object Detection Results

OpenCV Object Detection with GoogleNet	OpenCV Object Detection with GoogleNet and Improved Method
32.93 sec	32.57 sec

Figure 3. 9 – RISC-V OpenCV Object Detection Output

3.3. A NEW APPROACH USING RISC-V ISA FOR FPGA

In the previous parts of the thesis, it is thought that the time results of the softcore Chipyard SoC Generator are not competitive with today's processors. On the other hand, it is thought that the results obtained when OpenCV libraries are installed according to RISC-V ISA with the cross compiling method do not reflect the real results because they do not work on a real RISC-V processor. Finally, although Linux, which was built on the RISC-V softcore processor with QEMU, was considered a successful architecture, a complete neural network application could not be realized because it does not support many Python libraries and vector extensions. In this part of the study, it is aimed to connect the FPGA to the main AXI bus interface by loading a softcore processor architecture designed on the FPGA. With this method, the RISC-V processor will act as the main processor running on the FPGA and will manage all functions, interfaces and peripheral units located on the PL and PS side of the FPGA.

In this context, in the last part of the thesis, an artificial neural network supported object recognition application was implemented by using RISC-V architecture on FPGA. Xilinx Zynq UltraScale+ MPSoC ZCU102 [89] development board is used which is shown in Figure-3.10. Within the scope of this study, it is planned that the on-board DPU units on the aforementioned development board will act as the main processor to perform the neural network application. Instead of the AXI interconnection bus structure, which is responsible for the communication between the PL and PS components [89][90] in the Xilinx Zynq SoC, the RISC-V bus structure is used with AXI buses as main processor and it is aimed to observe performance improvement. In order to implement the AXI bus adapter on RISC-V, the RISC-V ISA Toolchain must be configured to include the “A” atomic extension.

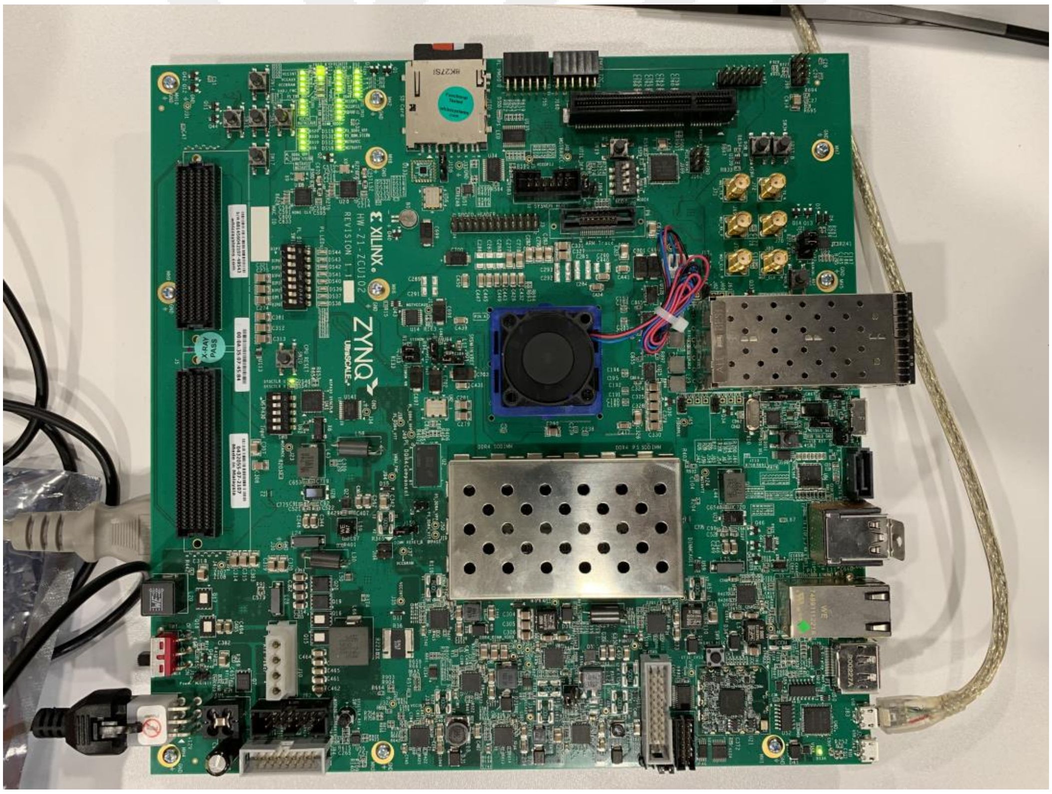


Figure 3. 10 – Xilinx Zynq UltraScale+ MPSoC ZCU102 FPGA

Xilinx Zynq UltraScale+ MPSoC ZCU102 is a general-purpose evaluation and development board, which contains an ARM architecture as a PS unit and FPGA as a PL unit. In addition, this development board, which has 599,550 logic cells and 548,160 programmable flip flops [90], is thought to be suitable for RISC-V integration and processes that require high performance due to extra convolution operation takes approximately 500 to 3000 Look-Up Tables (LUT) [91]. LUTs are basically, array tables that consist of flip-flops and logic cells and are used to hold the processed data. LUTs are one of FPGA's most basic structures and can be thought of as a small RAM structure, holding a one-bit value in return for the address.

On the other hand, it is aimed to improve the performance by increasing the level of parallelization on evaluation board with the DPUCZDX8G model DPU, which is a configurable computation engine. Due to DPU parallelization, pooling operations such as maximum pooling, average pooling, convolution operations such as elementwise and depth wise convolution and transposed convolution, activation functions such as ReLU, Leaky ReLU, Sigmoid, Swish and other neural network and image processing operations for instances dilation and fully connected layer operations can be performed. In addition, convolutional neural network algorithms such as VGG, ResNet and GoogleNet is currently deployed as an example in DPU [92].

To achieve high throughput and efficiency, on-chip memory is employed to buffer input activations, intermediate feature-maps, and output meta-data. High speed data pipeline designed to use for computing engine. The relationship between High Speed Data Pipeline, External DDR RAM and Global Memory Pool relationship is demonstrated in Figure-3.11.

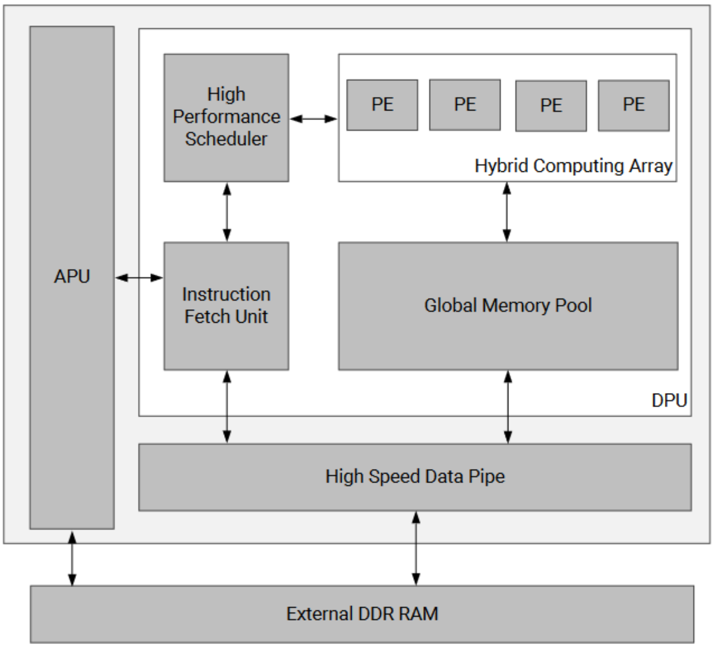


Figure 3. 11 – High Speed Data Pipeline of DPU [92]

DPUCZDX8G unit [92] integrated into the FPGA system with AXI Bus Structure to execute high performance tasks. Through the AXI Interface, PL and PS components are connected to each other. Figure-3.12 demonstrates the main AXI Interface and connection structure between PL and PS. The DPU contains two 32-bit memory mapped connections, three 128-bit memory map master interfaces, one AXI clock, one AXI reset, a Digital Signal Processor (DSP) clock and a DSP reset [92].

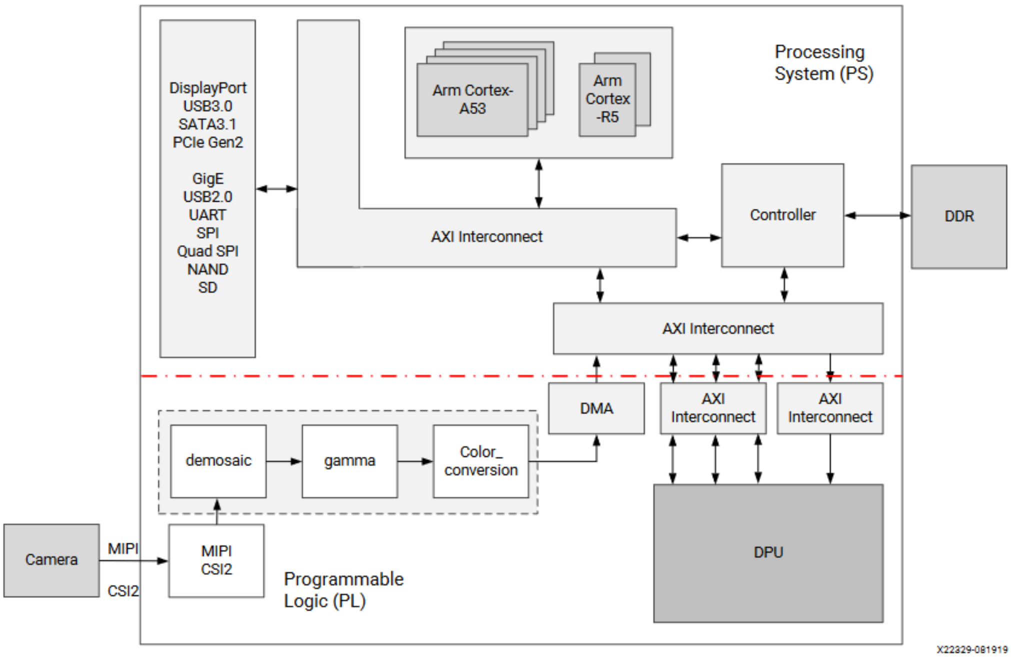


Figure 3. 12 – AXI Interface of DPU [92]

In the FPGA board, there are nine AXI interfaces between the PL and the PS components. General Purpose Ports have four 32-bit ports which are used for running Python codes. The detailed interfaces are shown in Figure-3.13. RISC-V architecture is configured over 32-bit General Purpose Ports in order to run Python code on the PS unit and control it over PL [90]. The M ports show the Master ports of the AXI and S ports show the Slave ports of the AXI buses. As can be seen in the figure, the first two of the four High Performance buses are directly connected to DRAM, while the remaining two are connected to DRAM via AXI Interconnect. For high performance buses, the master is on the PL side and the slave is on the PS side. On the other hand, two of the other 4 general purpose buses come out from the PL side and are connected to the peripherals. The other two connect to PL from PS ARM caches. The only buses in the AXI interface with master configuration on the PS side are general purpose buses connected to caches.

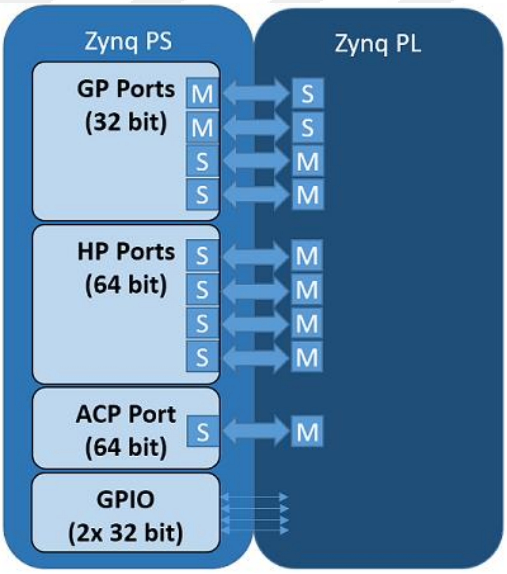


Figure 3. 13 – PL/PS Interface of FPGA [93]

Although there is no limit for data transfer in the AXI4 master bus structure, there is a limit on the maximum data that can be transferred per second in AXI4 slave buses and many data transfer protocols. The maximum data transfer rates and limits are given in Table-3.11. As shown in the table, data transfer protocols have its own limits. Since there is no limit for data transfer in this architecture created with RISC-V, there is no performance limitation regarding the transfer protocol. Since the AXI4 is, the most efficient way of transferring data buses, nevertheless in FPGA, AXI buses take 600 to 4000 LUT [94] due to the size and the complexity of the AXI4 bus design. Since the implementation of AXI4 bus is, too complex and resource consuming, the RISC-V core implementation is more efficient wherefore it contains AXI bus structure and is programmable and interchangeable for other process.

Table 3. 11 – Data Transfer Protocol Limitations [95]

Data Transfer Protocols	Data Bus Width	Address Bus Width
AMBA	32, 64, 128, 256 byte	32 bit
Avalon	1-128 bit	1-32 bit
CoreConnect	32, 64, 128, 256 byte	10 byte- 32 byte
Wishbone	8, 16, 32, 64 bit	1-64 bit
PI Bus	1-32	1-32 bit

3.3.1. Using Xilinx Board For RISC-V Implementation

This section describes the AXI bus core built using the RISC-V ISA. The RISC-V ISA is configured to run Atomic Instructions so it can run a transfer protocol. The cores created with the RISC-V ISA in the previous section such as BOOM Core and Rocket Core are also used in this section. The advantage of BOOM Core's out of order structure, which was used in the previous sections and tried to be processed directly, is that the instructions are not taken in a certain order, which makes the processor wait for less during the fetch-decode-execute-memory-write back five stage pipeline stages. SiFive's RISC-V core, the E2, has only a single connection between master and slave [96]. Thus, the number of connections is less, but there is a pause between read and write operations in the five stage pipeline. On the other hand, a GPU and a neural processor have been added on top of the 8 CPU cores in the Apple M2 processor [97]. Since these cores do not perform multi-threading operations in the same way and at the same level, their energy consumption and performance are also different from each other. The RISC-V based processor designed on FPGA is planned to work in AXI4 bus architecture. It is also planned to act as a master processor with write and read rights on Input/Output and internal memory. In the architecture where FPGA and embedded systems are targeted, system and application-specific customizations have been made and it is expected to provide power usage and performance advantages. The main structure of the design is shown in Figure-3.14. In this structure, CPU cores are RISC-V based and control the AXI bus as a master. DPU is the hardware for executing the neural network operations. PL Unit is the traditional FPGA and PS unit is the ARM core for Zynq FPGA.

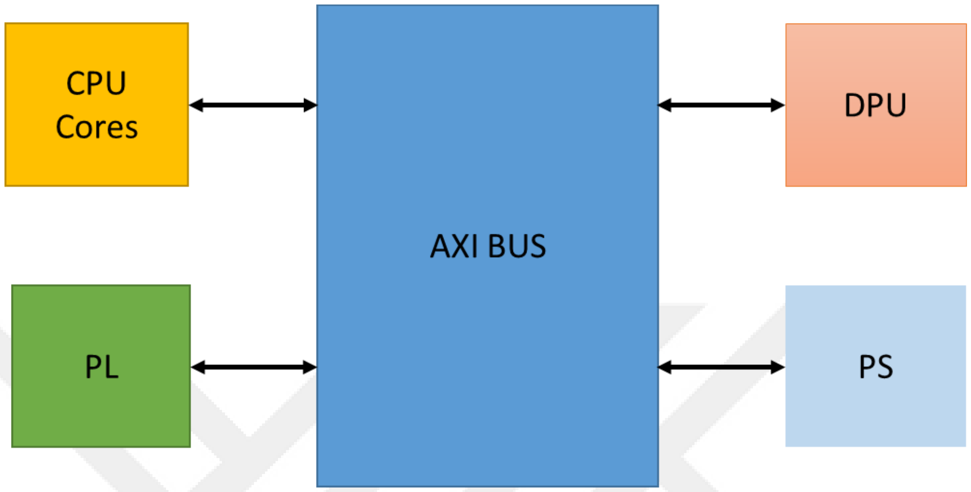


Figure 3. 14 – System Block Diagram

The Figure-3.15 is taken from Xilinx Vivado ISE. This figure shows the configurable components, Full Power Domain (FPD) components and Low-Power Domain (LPD) [98]. The FPD controls the majority of the digital logic and Input/Output resources in FPGA. LPD is also a power domain and operates when FPGA works in low power mode. It supplies the reduced clock frequency and lower voltage level. On the other hand, the Configurable part is controlled by 32-bit CPM_TOPSW_REF_CTRL register [99]. It is also dynamically configured and controlled by the user through software or hardware. The created RISC-V Core is the master for AXI bus seen in Figure-3.37 as axi_smc. axi_aclk is the main clock and the axi_aresetn is the main reset pins. There are thirteen write address channel pins such as axi_awiddr, axi_awcache, axi_awvalid; seven write data channel pins such as axi_wdata, axi_wuser, axi_wvalid and five write response channel pins such as axi_bresp, axi_buser are implemented. Additionally, there are thirteen read address channel pins for example axi_arid, axi_arsize and seven read response channel pins such as axi_rid and axi_ruser are implemented in the RISC-V core architecture. Since the performance effect of the memory is not concerned, Direct Memory Access (DMA) working on PL and DRAM controller working on PS were constant in order not to change the scope of the default memory working principle.

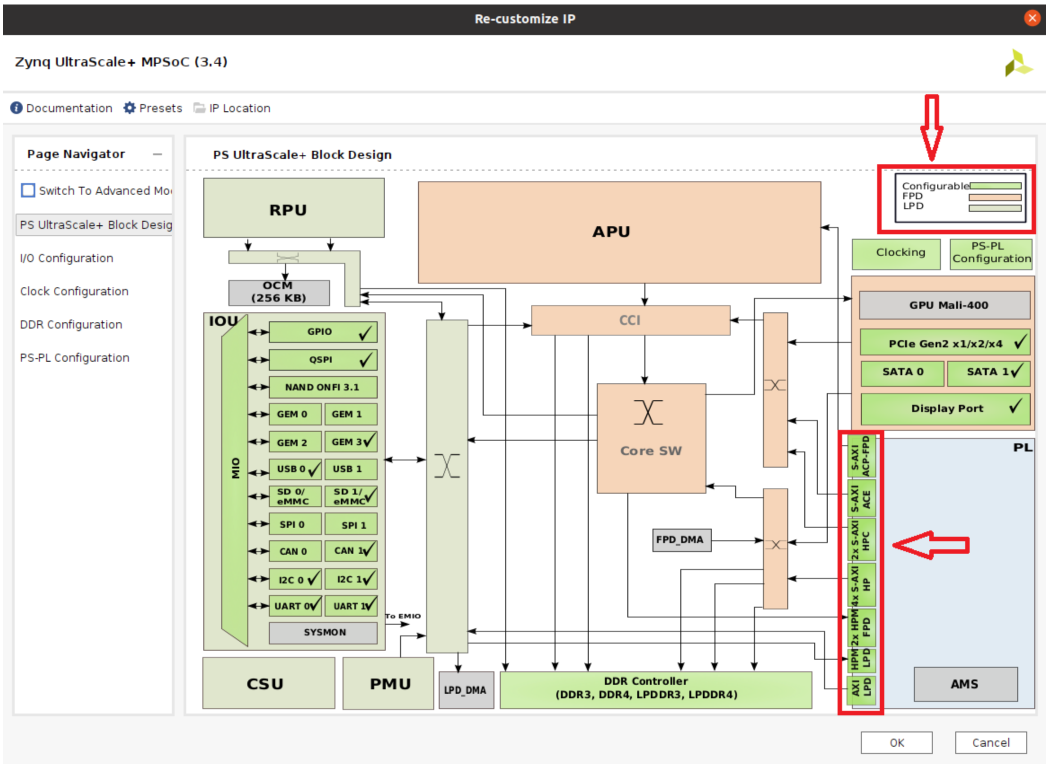


Figure 3. 15 – Vivado Customizable IPs

In the new architecture designed, the softcore processor FPGA, which was designed in Section-3.1.1 of the thesis work and produced with the Chipyard SoC Generator, is loaded into the PL section of Xilinx Zynq UltraScale+ MPSoC ZCU102. Since the Chipyard SoC Generator already has an AXI interface, RISC-V's AXI ports are connected to the main AXI ports of the FPGA. Thereby, DPU, ARM core and other peripherals can be managed by RISC-V via AXI. Thus, it is planned that RISC-V will be able to provide fast data processing with DPU instead of its own RAM and Arithmetic Logic Unit (ALU) components. On the other hand, since the processor produced with the Chipyard SoC Generator can only run C and C++ languages and does not have a visual interface, assistance was received from the ARM processor. With this method, after the ARM processor runs the Python codes that will start the operations in line with the instruction from RISC-V, the resulting data will be transferred to the DPU unit via AXI buses over the RISC-V core. The data processed on the DPU will be transferred back to the RISC-V processor, where it will be transmitted to the ARM core for visualization after passing the necessary processing. Figure-3.16 shows the newly designed schema and connections in Xilinx Zynq architecture. In this architecture, all AXI buses are designed to be passed over RISC-V. The implementation diagram of the designed system, taken from Vivado, is shown in detail in Appendix-3.

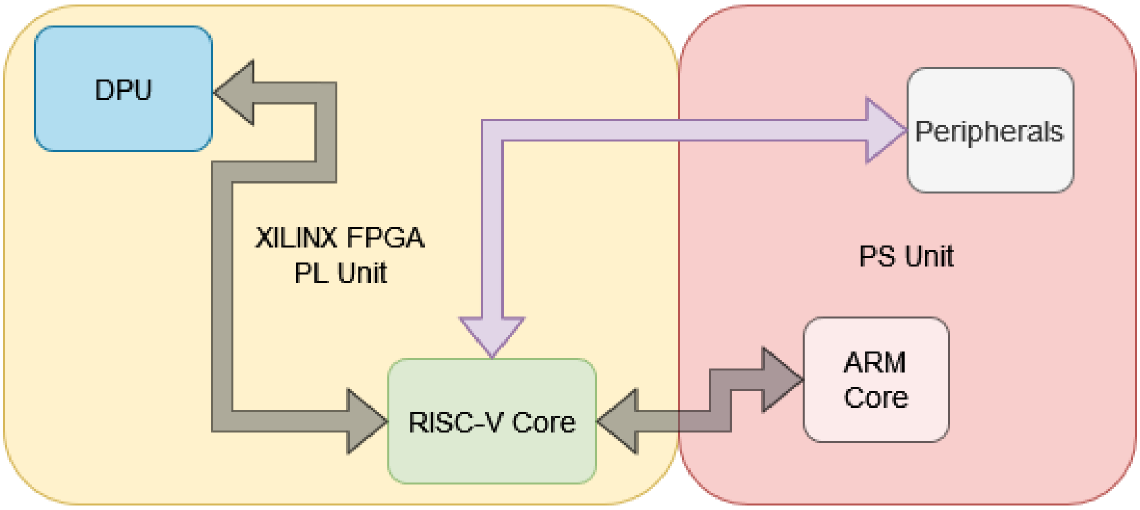


Figure 3. 16 – New FPGA Configuration with RISC-V

3.3.2. New Object Detection Approach

In this section, a neural network based object recognition algorithm designed to run on FPGA is explained. The designed model includes parts from popular models that were successful in ILSVRC2015 competition and widely studied in object recognition. It has been developed by using the methods in which the algorithms described in Section-3.2.1 provide solutions to the previous problems. In this context, the developed model was created using VGGNet, ResNet and DenseNet infrastructures. Multiple layers brought by VGGNet, smaller but higher dimensional convolution kernels and deeper network structures are used. The residual block structure, which was developed with ResNet and is a solution to the vanishing gradient problem, allows the networks to deepen and prevents the loss from increasing. In addition, the batch normalization structure, which was introduced with ResNet and made the model more stable, was used. On the other hand, the reduction in overfitting and neat hierarchical structure brought about by the dense layers introduced with DenseNet were utilized. The purpose of this section is not to cross compiling with OpenCV over RISC-V and use a pre-trained model as was done in the previous Section-3.2.2. The main purpose is to train an object recognition algorithm with The Canadian Institute for Advanced Research's CIFAR-10 dataset [100], to observe the performance and accurate test results on the FPGA with the resulting weight file, and to compare it with other object recognition models. The created model was developed in Python using the Tensorflow library.

Again, it is aimed to work on the bus and DPU unit controlled by RISC-V on Python on FPGA. The Python code written for the model and the detailed schema of the model is in the Appendix-1 section. The diagram describing the general structure of the model is in Figure-3.17. The learning epoch time initiated over a total of 100 epochs was shortened because the time of the epoch was too long and the test validation rate stopped increasing after a certain epoch, but the train validation rate was constantly increasing. In addition, the learning rate is designed in a form that decreases as the number of epochs increases, thus protecting it from overfitting. The details of the model is seen in Figure-3.17, its explanation in the detailed text is in the Appendix-2 section and the detailed schema is in Appendix-4.

In this training, it was observed that while the percentage of validation accuracy and test accuracy increased in each epoch, the results of validation loss and test loss decreased. Accordingly, it is thought that there is no overfitting during the training of the network. On the other hand, since the percentage of validation accuracy and test accuracy increases in each training epoch, it is thought that there is no vanishing gradient problem. When the number of training epochs is increased and the batch size is increased, the increase rate of the accuracy values decreases as the number of epoch increases. Therefore, another way to avoid overfitting is to keep the epoch number balanced and to keep the batch number low, based on the size of the images in the dataset and the depth of the network.

In this part of the study, after the RISC-V processor was implemented on the FPGA, DenseNet, one of the widely used neural network applications, and the Inspired model introduced in this section were run with the RISC-V processor on the FPGA and compared in terms of time performances. The main purpose of this section is to evaluate the performance of the RISC-V processor on neural network applications and its ability to manage the Xilinx Zynq architecture. The reason for creating and using another model other than DenseNet is that DenseNet has a suitable architecture for parallel processing. Therefore, the second and simple but successful network is to demonstrate the performance of RISC-V. The RISC-V processor positioned on the FPGA is designed to be used to test a trained model to train a model since it uses a DPU instead of its own ALU unit and RAM module. Therefore, after training the model on the GPU, a neural network application can be performed by running it with the weight file and dataset. Table-3.12 gives the time results and accuracy rates of DenseNet trained on CIFAR dataset and newly created InspiredNet on RISC-V and ARM cores, respectively. Since the two models are trained on the same GPU and with the same dataset, not on the RISC-V core, the accuracy rates are shown to show the success rate of the created model, although there is no direct connection with the RISC-V processor.

Table 3. 12 – FPGA and RISC-V Comparison with Object Classification

Model Name	Accuracy	Time	Implementation
DenseNet	86.77%	21.12 sec	RISC-V on FPGA
InspiredNet	82.59%	27.73 sec	RISC-V on FPGA
DenseNet	86.77%	19.02 sec	ARM Core
InspiredNet	82.59%	25.74 sec	ARM Core

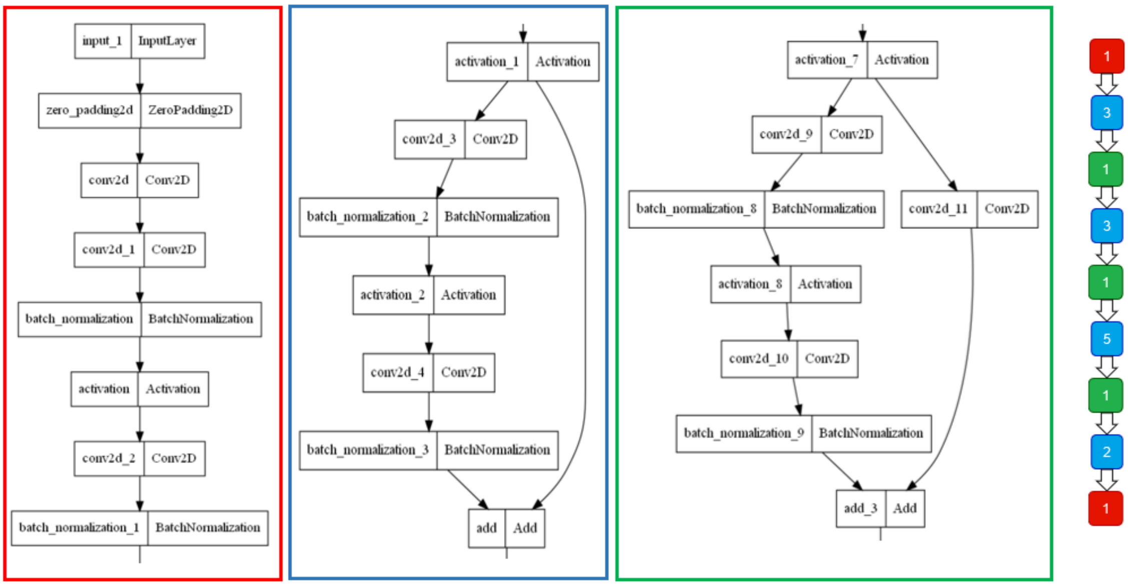


Figure 3. 17 – General Design Flow of New Model

4. RESULTS AND DISCUSSION

In this thesis, different RISC-V kernels and architectures have been implemented in different systems. In addition, image processing and neural network algorithms are run in each implemented RISC-V architecture and time results are obtained. Three different paths have been followed regarding RISC-V. First, a virtual processor was created on the Linux operating system with Chipyard SoC Generator. Then a virtual processor was created on QEMU and the operating system was installed on it. In the next section, a widely used image processing library is run with RISC-V architecture by cross-compiling method. Performance analyzes of these three methods were obtained with different methods and compared. In the last part of the study, a new and effective method based on RISC-V is presented to perform a neural network application on the FPGA, so that the DPU and ARM cores in the FPGA communicate over the RISC-V architecture and the AXI bus structure is not occupied and the communication infrastructure is customized. In addition, a neural network supported image processing algorithm was created by using the superior aspects of the existing algorithms and it was run on the FPGA in the configuration where the RISC-V architecture works as accelerator.

4.1. Results

In this thesis, different RISC-V kernels and architectures have been implemented in different systems. In addition, image processing and neural network algorithms are implemented and executed on each implemented RISC-V architecture and time results are obtained. Three different paths have been followed regarding RISC-V. After the installations were completed, a network layer simulation was run. Since the time results of the softcore processor produced with the Chipyard SoC Generator are thought to be slow compared to the other implementation results, the first layer results are compared instead of a full network due to the limited support of the external library and the inability to fully run C and C++ languages. Then, a full network simulation and traditional image processing application was run with architectures that were found to be more successful in terms of time. First, a virtual processor was created on the Linux operating system with Chipyard SoC Generator. Then a virtual processor was created on QEMU and the operating system was installed on it. Then, a widely used image processing library that is OpenCV is installed with RISC-V architecture by cross-compiling method. Performance analyzes of these three methods were obtained and compared. In the second part of the study, an improved method for traditional object detection is designed. After that, the traditional object detection method is implemented in RISC-V core with QEMU Linux and OpenCV and RISC-V cross compiling architectures. In the third part of the study, a full neural network simulation is try to be executed in RISC-V core with QEMU Linux and OpenCV and RISC-V cross compiling architectures again. In the last part of the study, a new and effective method based on RISC-V is presented to perform a neural network application on the FPGA, so that the DPU and ARM cores in the FPGA communicate over the RISC-V architecture and the communication infrastructure is customized over RISC-V as main processor. In addition, a neural network supported image-processing algorithm was created by using the superior aspects of the existing algorithms and it was run on the FPGA in the configuration where the RISC-V architecture works as a main processor.

4.1.1. Results for Installing and Sample Network Layer Application

The selected techniques for testing the performance of the RISC-V architecture are established in the first part of the experimental work, and the first layer of a basic neural network is simulated. This procedure was chosen because it incorporates several matrix operations, including resizing, filtering, convolution, ReLU, and max pooling. A full neural network is not simulated since the Chipyard SoC Generator does not support libraries and the RISC-V processor used to run Linux with QEMU does not enable vector extension. Table-4.1 displays all of the operation's outcomes.

Table 4. 1 – Installing and Sample Network Layer Outputs

Operation	**Architecture**	**Time Result**
Convolution with 32, 3x3 Kernel ReLU Operation Max Pooling Operation	Chipyard SoC Generator RocketCore + BOOMCore	134 min 46.237 sec
	Chipyard SoC Generator RocketCore + BOOMCore + Gemmini Accelerator	92 min 76.383 sec
	Linux on RISC-V with QEMU	1 min 10.152 sec
	OpenCV with RISC-V ISA	0 min 6.619 sec

As can be seen from the table, two different configurations produced with the Chipyard SoC Generator took longer time than the others. This is because the Chipyard SoC Generator has been designed for educational purposes as a processor generator that can work on all systems in general. SiFive is able to achieve high performance in its FPGAs by customizing RocketCore. However, it has been shown that a ready-made and non-customized Chipyard SoC Generator processor cannot be implemented as softcore on FPGA or run on a computer. On the other hand, although the results were improved when the Gemmini Accelerator was used, the improvement rate seen between 60% and 70% in simple operations decreased to around 31%. The main reason for this is Gemmini's ability to use 16x16 matrices as inputs, and the performance decreases, as a tile mechanism is required for larger matrices. Traditional object detection application and full neural network simulation will not be simulated with the Chipyard SoC Generator, as it is understood that the operations on the matrix take a long time with the cores produced with the Chipyard SoC Generator.

On the other hand, there is a delay due to the fact that the vector extension could not be installed in the same example running with Linux on the RISC-V kernel with QEMU. However, it is seen that the results obtained are much better than the results obtained with the Chipyard SoC Generator. For this reason, it was deemed appropriate to perform image processing and full network simulation with the traditional method with this architecture and examine the results.

After the installation of the OpenCV libraries, which is the last work of this section, for the RISC-V ISA with the cross compiling mechanism, the library paths were specified for RISC-V and the sample application was run. The main reason why the most appropriate results were obtained with this architecture in the tests carried out is that this architecture is not implemented on a real RISC-V processor, but only the instructions are compiled for OpenCV. However, traditional object detection application and full network simulation will be performed to fully understand the capabilities of the architecture.

4.1.2. Results for Traditional Object Detection Application

In this part of the thesis, firstly, traditional object detection methods were examined and it was noticed that these methods consisted of shallow networks and hand crafted features. Later on, it was understood that the object detection methods used the sliding window method frequently and searched for where the object might be in the entire input image. Since this approach is thought to affect both accuracy and time results, a single object detection application was considered on the histogram map of the input image. When the outputs obtained with this model developed on Matlab are applied to a dataset, a 3% delay was detected in terms of time, but it was determined that the accuracy increased by 8%. The developed single object detection method was applied to the successful RISC-V architectures in the previous section and the time results were compared. Since the same kernels and the same filter parameters are given to the codes run in C++ for OpenCV, Python for QEMU Linux, the results are seen to be almost the same. The results obtained with the Linux operating system running with RISC-V kernel on QEMU and the results obtained with the OpenCV library installed according to the cross compiling method with RISC-V are given in Table-4.2. The main reason why QEMU Linux and RISC-V results take longer than OpenCV RISC-V results is that they run on a softcore processor, the vector extension is not supported, and all operations are done with manually written functions due to the lack of Python libraries.

Table 4. 2 – Improved Traditional Object Detection Outputs

Operation	Architecture	Time Result
Traditional Object Detection with Improved Histogram Comparison Approach	Linux on RISC-V with QEMU	33.212 sec
	OpenCV with RISC-V ISA	4.714 sec

4.1.3. Results for Neural Network Application

In the third part of the thesis, a complete neural network application has been implemented on the RISC-V integration architectures that were successful in the previous parts. In this context, GoogleNet, which uses the InceptionNet infrastructure for the neural network planned to be implemented, due to obtain fast results and is the winner of the ILSVRC 2014 competition. Since vector extension and neural network libraries are not supported in RISC-V and Linux architectures built with QEMU, VGG-19 network is implemented instead of GoogleNet for QEMU Linux architecture. For OpenCV, a C++ code was executed to which DNN libraries and CUDA libraries were added. A ready-trained model was used for GoogleNet convolution kernels. On the other hand, since the fully connected layer modules in the classification layer are very difficult to implement without libraries such as TensorFlow, the classification layer is not included in the QEMU Linux architecture and VGG-19 layers are created. Table-4.3 shows the results of full neural network simulation.

Table 4. 3 – Full Neural Network Implementation Outputs

Operation	Architecture	Time Result
VGG-19 Network	Linux on RISC-V with QEMU	601.88 sec
GoogleNet	OpenCV with RISC-V ISA	32.57 sec
VGG-19 Network	AMD Ryzen-5 Pro 4650U	

	2.1GHz, MATLAB 2021b and MATLAB Live Editor	4.21 sec
GoogleNet	AMD Ryzen-5 Pro 4650U 2.1GHz, MATLAB 2021b and MATLAB Live Editor	3.93 sec

Although the classification layers are not implemented, it has been found that the VGG-19 network, which is run on QEMU Linux and RISC-V architecture, takes more than 10 minutes in total. When Linux and Python fully support RISC-V instructions, it is expected that a full neural network implementation will be created with the help of the Tensorflow and Mathplotlib libraries, reducing the time required. However, it has been noticed that today's computers are far from the findings in their current form. Although it can conduct functionally effective tasks by putting an operating system on the Softcore RISC-V processor, it is not time-efficient. The time results, on the other hand, were the best in this example since the OpenCV library was constructed by simply cross-compiling the RISC-V instructions.The results produced using GoogleNet, which uses pre-trained weights, are nonetheless equivalent to those obtained with modern computers. However, as previously stated, the OpenCV architecture's results should not be directly compared to the CPU architecture because there is no real processor and it only works with cross compile.

4.1.4. Results for a New Object Classification Approach, The Inspired Model

In this part of the thesis, a neural network was created, which is thought to be realized within the scope of RISC-V and FPGA integration. This model has been developed to use the positive and positive aspects of VGGNet, ResNet and DenseNet. It is aimed to have a simple design with VGGNet structure. However, since the number of parameters of VGGNet is higher than other networks, the model size is very large and there may be a vanishing gradient problem, ResNet's convolutional loop structure has been used. Thus, the vanishing gradient problem was solved without network complexity and the size of the model was reduced. Finally, since ResNet has insufficient points for a small dataset, the model is completed using DenseNet's dense block structure. The outputs of the model are as in Table-4.X. The purpose of designing this model is to make a separate test with RISC-V architecture in FPGA. Since DenseNet has a structure that is very suitable for parallelization, it is aimed to understand the real performance of RISC-V with this model, which is simpler and whose speed cannot be increased much with parallel processing. The entire training phase of the model was carried out on the GPU and with the CIFAR-10 dataset. Although the dimensions of the images in the dataset are 32x32, no overfitting and vanishing gradient problems were observed after 20 epochs while the model was being trained. Table-4.4 shows the training and testing results on GPU.

Table 4. 4 – Inspired Model Design Outputs

Validation Accuracy	Epoch	Environment	Hardware
82.59%	100 Epoch	PyCharm Python 3.9	Intel Xeon Silver Nvidia RTX 3080TI

4.1.5. Results for RISC-V FPGA and Neural Network Application

In the last section of the thesis work, the RISC-V ISA is configured as a softcore processor on an FPGA. The Chipyard SoC Generator produces delayed output, barring it from being used in sophisticated processes. With the deficiencies in Python and vector extension support of the QEMU Linux architecture, this architect is not fully usable for now. Although the OpenCV library produced successful results, the fact that it did not function as a RISC-V processor in this architecture necessitated a hardware implementation to evaluate the capabilities of a processor designed using the RISC-V ISA. As a result, the AXI interface connects the RISC-V processor on the PL side of the Xilinx Zynq UltraScale+ MPSoC ZCU102 board to the ARM core and DPU unit. Thus, the ARM core handled code execution and visualization, while the DPU units performed data processing. In this arrangement, the RISC-V processor's performance is boosted by utilizing the DPU instead of the RISC-V processor's ALU and memory unit, and by using ARM instead of the RISC-V processor's cache and peripheral units. Table-4.5 shows the results of the tests performed on DenseNet and the built network (InspiredNet).

Table 4. 5 – DenseNet and InspiredNet RISC-V Results

Operation	Architecture	Accuracy	Time Result
DenseNet CIFAR Test	Xilinx Zynq UltraScale+ MPSoC ZCU102 with RISC-V Core	86.77%	21.12 sec
InspiredNet CIFAR Test	Xilinx Zynq UltraScale+ MPSoC ZCU102 with RISC-V Core	82.59%	27.73 sec

As shown in Table-4.5, when the processor constructed with RISC-V ISA is utilized as the main control unit, the results are enhanced and the method's usability is boosted. In order to evaluate the regular performance of FPGA, DenseNet and InspiredNet networks are operated on the ARM core in the default configuration of FPGA. The results of the network application on the ARM processor are shown in Table-4.6. As a consequence, the ARM processor achieved about 9.8% faster than DenseNet and 7.1% faster than InspiredNet results. The difference was deemed acceptable, and it was demonstrated that the RISC-V processor could compete with today's CPUs in this arrangement. Appendix-4 contains the InspiredNet design diagram and console outputs, as well as the DenseNet console outputs. Appendix-2 contains detailed design procedures for InspiredNet as well as source code. Since DenseNet has a structure that is suitable for multi-thread operation due to the dense blocks in its structure, the results are reduced to 11 seconds when the ARM processor is run with five multi-thread capabilities. However, multi-threading was not used in this study in order to compare both the RISC-V kernel and the ARM kernel and the comparison of InspiredNet and DenseNet.

Table 4. 6 – DenseNet and InspiredNet ARM Results

Operation	Architecture	Accuracy	Time Result
DenseNet CIFAR Test	Xilinx Zynq UltraScale+ MPSoC ZCU102 Standard Configuration	86.77%	19.02 sec
InspiredNet CIFAR	Xilinx Zynq UltraScale+ MPSoC ZCU102 Standard	82.59%	25.74 sec

Test	Configuration		

4.2. Discussion

In this part of the thesis, the results of the experiments that have been carried out are compared and evaluated. Then, a use case was created with the RISC-V architecture integrated on the FPGA used in the thesis. The possible results of the created use case in terms of time performance, accuracy and data transfer were examined. In the last part of the Discussion section, the software and hardware implementations carried out in the world on the basis of RISC-V ISA were examined and the architecture used in the thesis was supported. After examining these methods, it has been revealed that a RISC-V processor used without customization will perform slowly.

4.2.1. Discussion of the Thesis Experiments Results

The experiments carried out in the thesis work can be divided into two main software and hardware implementations. Software implementations include building a softcore processor using the RISC-V ISA. The first study implemented in this context is the structure containing RocketCore and BOOMCore, which was created with the Chipyard SoC Generator. Later, the Gemmini Accelerator was added to this structure and the first layer of a simple neural network was run. Since the time results obtained were observed to be more than one hour, it was understood that these architectures could not be continued.

Later, a virtual processor was created with the RISC-V ISA via the QEMU emulator and the Linux operating system was installed. Thus, an operating system was installed on the RISC-V softcore processor. The first layer of a neural network application was implemented on this architecture and a successful result was obtained. Then, the traditional object detection method application, which was created based on histogram comparison, was carried out. After applying this architecture, it is aimed to implement a complete neural network-based object classification application. However, since Linux's vector extention support is not provided as open source yet, Python libraries cannot fully work. For this reason, Tensorflow and Mathplotlib libraries could not be installed and all convolution layers in the network were written manually. In addition, the classification layer of the neural network part does not work because fully connected layers cannot be implemented. The final time result was measured as over 10 minutes. It is predicted that performance will increase when Linux provides vector extension support. For example, the Linux kernel implemented by SiFive has vector extensions.

In the final part of the software implementation, the OpenCV libraries were installed using the RISC-V ISA. Again, after the first layer of a neural network application was implemented, the histogram supported object detection application was also run. Since the time outputs of this application are also successful compared to other architectures, it is aimed to run the neural network supported object classification application. In this context, a pre-trained GoogleNet application has been successfully implemented. Again, the time results of this application were compared with today's computers and were found successful. The main reason why this architecture is the most successful is that OpenCV doesn't actually run on a RISC-V processor, it is only cross compiled. In other words, OpenCV libraries were installed with the RISC-V ISA and operations were performed on the host's processor.

The insufficient time results of the architectures where the tests were made by creating a RISC-V processor and the problems such as extension, library and language support made it necessary to test the RISC-V architecture on a hardware. In this context, with the RISC-V processor implemented on the Xilinx Zynq UltraScale+ MPSoC ZCU102 board, all modules in the FPGA and connected to the AXI interface are controlled. In other words, DPU is used instead of memory units and ALU in the implemented RocketCore architecture. In addition, the ARM kernel is used as a code executor and visualizer, since RISC-V is not yet supported by all operating systems and languages. Tests using DenseNet and the created InspiredNet networks were compared with the results of the architecture without RISC-V architecture. The results showed that the RISC-V processor can be as powerful as the ARM processor.

4.2.2. Discussion on a Use Case for Future Work

In this section, a use case review has been made with the artificial intelligence infrastructure and RISC-V architecture hardware. In this use case, each member of a 10-

person military team has clothing equipped with wearable technology products and containing different sensors. The amplitude values of the sensors used are thought to be between 1mV and 5mV. In addition, it is thought that the operating frequency of the sensors used will be 14400Hz and the size of the data to be transmitted from the sensors will be 16 bits. The number and usage area of wearable technology products is increasing with the development of IoT devices. With the wearable technology products, which are frequently used in military technologies, it is aimed to develop a system that can decide whether it is suitable for operational activities according to the health and psychological status of the soldiers. The following data is obtained from each soldier through wearable technology products.

- EEG
- EMG
- ECG
- Body Temperature
- Hydration Sensor
- Oxygen Rate in Blood
- Blood Pressure
- Fatigue Sensor
- Electrophysiology

Artificial intelligence technology was used to make sense of all the data received from the soldiers through sensors. In this context, data were obtained from a large number of soldiers who were confirmed to be healthy with the same sensors and whose body temperature and water ratio were normal. Then, with the same sensors, data is recorded from the same soldiers during military training. All recorded data is tagged and fed to the neural network algorithm as a dataset. Recurrent Neural Network (RNN) can be used because the data received from the soldiers through sensors have certain time intervals. RNNs seem suitable for data coming in a sequence. The Figure-4.1 shows the DNN training flow.

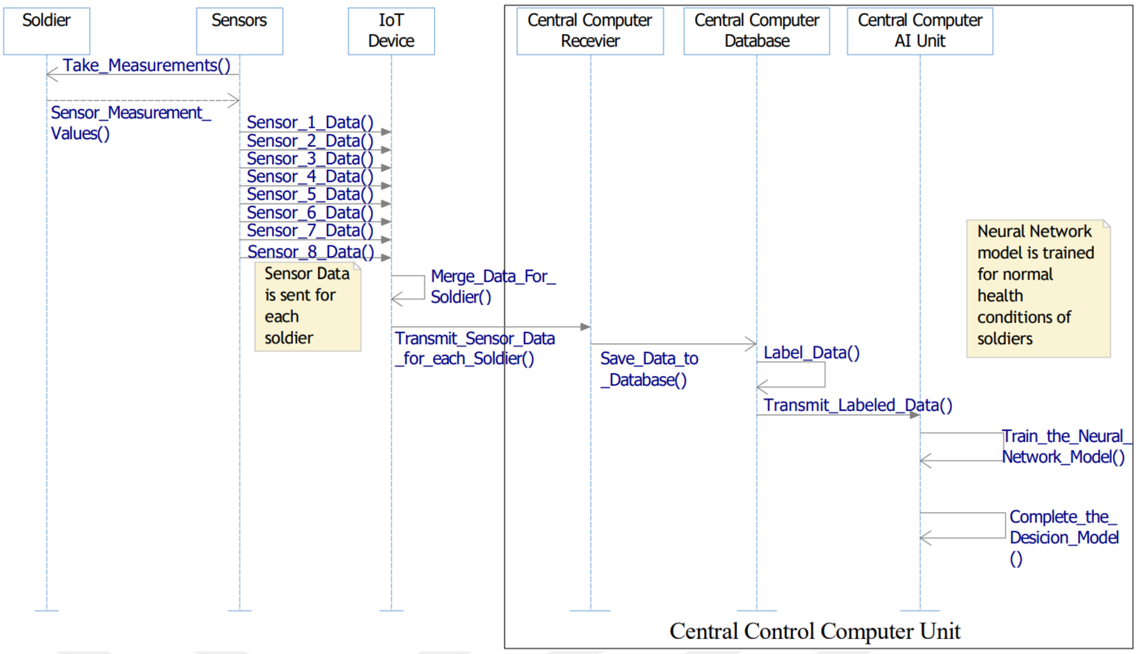


Figure 4. 1 – General Design Flow for Data Training

Electroencephalogram (EEG), one of the sensors used, can examine electrical movements in the brain. In this way, many problems from sleep disorders to brain trauma can be understood. Electromyography (EMG), on the other hand, transmits the electrical signal received from the sensory nerves to the brain via the spinal cord. With EMG, electrical conduction in sensory and motor conduction pathways and electrical behavior of muscles can be examined. Electrocardiography (ECG) is used to measure heart rhythm and frequency. With ECG, rhythm disorders in the heart can be detected. The body temperature sensor measures the body temperature on the skin. In this way, the soldier's suitability for duty can be examined by recording the increase or decrease in body temperature compared to the normal situation. By positioning the temperature sensor to measure temperature from different parts of the body, it can also give an average body temperature value according to the number of sensors. The hydration sensor measures the water level in the body. Body water level is an important parameter since the decrease in water ratio has many different effects on the body, from loss of reflexes to distraction. The oxygen level in the blood is a very important parameter for the proper functioning of the organs. By measuring the oxygen level in the blood, many inferences can be made regarding the general health status of the soldier, from the fact that the soldier may be in a place with limited oxygen and may have pneumonia. Blood pressure, like the oxygen level in the blood, is an important parameter for the proper functioning of body functions. Therefore, blood pressure and blood oxygen level should be measured separately. On the other hand, the fatigue sensor is one of the most important sensors measuring physical endurance. This sensor, which measures movement, kinetic energy, and yawning in general, can be decided that the soldier is tired by getting help from the ECG sensor data. The last sensor to be used is the Electrophysiology sensor. Since unusual heartbeats and electrical signals can be measured with this sensor, it was thought that it could be used directly in conjunction with measuring the stress level [101] [102].

The amplitude values of all sensors are planned to be between 1mV and 5mV. For this reason, the selection of sensors with this sensitivity will be sufficient for the operation of the system. It also means that this amount of data will be measured per second, assuming that all sensors operate at a frequency of 14400 Hz in general. Since this means that approximately 20 GB of data per day will be transferred to the central control computer, it is thought that the central control computer should be a computer capable of performing high-performance operations. On the other hand, assuming that all sensors operate at a frequency of 14400 Hz, the size of data to be sent per second can be calculated. Considering the general IoT device mechanism, sending the data from all sensors to the central control computer at different times will consume the device's battery fast, so sending the average of the data received from all sensors once a minute will be a good choice to save battery life. Assuming that all sensor data is 16-bit, 128-bit data from eight sensors and 144-bit data from nine sensors are transferred per minute. Thus, approximately 23 KiloByte (KB) of data is sent from a soldier per day. A team of 10 soldiers will have sent less than 1 MegaByte (MB) of data in total. It is thought that this data size can be processed and transmitted by a RISC-V based IoT device and a central control computer with RISC-V processor can easily process this size of data.

These sensors on the soldier will collect data for 24 hours at certain time intervals and the collected data will be transferred to the central control computer at certain time intervals via an IoT device. Military data transmission must be encrypted and secure. For this reason, the end point IoT device to be positioned on the soldiers must support the Scalar extension, Entropy, extension and Vector extension for the symmetric encryption to be used to contain the RISC-V processor [103]. The transmission module on the IoT board, which will be designed for the safe transfer of data, will communicate with RISC-V over the Universal asynchronous receiver-transmitter (UART). For example, Chipyard SoC Generator supports UART protocol by default. Since all the data of the soldiers will be encrypted with the symmetric encryption method and the cryptographic keys will be stored on the secure Read Only Memory (ROM) side. That cannot be accessed from the outside; it is thought that the data transfer between the end point IoT device on the soldiers and the central control computer can be done through the antenna relay mechanism. Thus, it is foreseen that there will be no data loss and a successful and fast data transfer will be ensured. For lossless data transfer to a center 100 kilometers (km) away, establishing a receiver-transmitter connection between the antennas of other devices in the field and establishing a relay mechanism for data transmission seems to be a suitable solution in terms of data security. On the other hand, cellular data usage will not comply with military standards and data loss will occur. In addition, considering that the data will be sent 24 hours a day for 30 days, it is thought that the IoT device will only be turned on when sending data and remain inactive when not sending data, so that the battery can last for a long time. In addition, considering that the IoT device will be prepared for military duty, it would be appropriate not to have any light or sound output on it.

The central control unit computer, on the other hand, contains a ready-made model that has been trained on sequential data previously collected and labeled from soldiers. This model is trained on GPU as data training with RISC-V architecture has not been developed yet. Since the central control computer contains a lot of data from many soldiers and will contain a neural network application, it was thought that it should be a computer with GPU and high processing capacity. However, the RISC-V architecture integrated on the FPGA created in this thesis is usable. For this, the RISC-V core, DPU unit and coprocessor that will provide the visual interface must be fabricated as an ASIC. Because the RISC-V core, which will work on the FPGA, has to process thousands of data coming from military teams of 10 people for 24 hours and transmit the decisions given to each team. In addition, since the RISC-V architecture does not currently have an open-source solution for end-to-end fast data transfer, this protocol will need to be produced separately. Figure-4.2 shows the design scheme and general flow diagram for a military system use case created with RISC-V and neural network architecture with trained neural network method. Table-4.7 contains the list of main devices required to implement such a use-case scenario.

Table 4. 7 – List of Devices in Use-Case Scenario

Number	System	Purpose of Use
1	Wearable IoT Device	
1.1	Sensors	It will be used to collect data from soldiers.
1.2	RISC-V Core	It will carry out all operations related to the collected data.
1.3	Battery and Key Storage Battery	It will ensure the operation of the device and the storage of the crypto key.
1.4	Transmitter and Receiver Unit	It will enable the device to send and receive data.
1.5	Embedded Memory	It will enable the device to store data.
1.6	Embedded Flash Memory	It will be used to store the cryptographic key and algorithm.
2	Central Control Computer Unit	
2.1	RISC-V Core	It will carry out all operations related to the received data.
2.2	GPU	It will be used to train the Neural Network model.
2.3	Storage	It will be used to store incoming data and neural network results.
2.4	Transmitter and Receiver Unit	It will enable the device to send and receive data and decisions.
2.5	Embedded Flash Memory	It will be used to store the key that will be used to decrypt the encrypted incoming data.
2.6	Key Storage Battery	It will be used to store the crypto key.

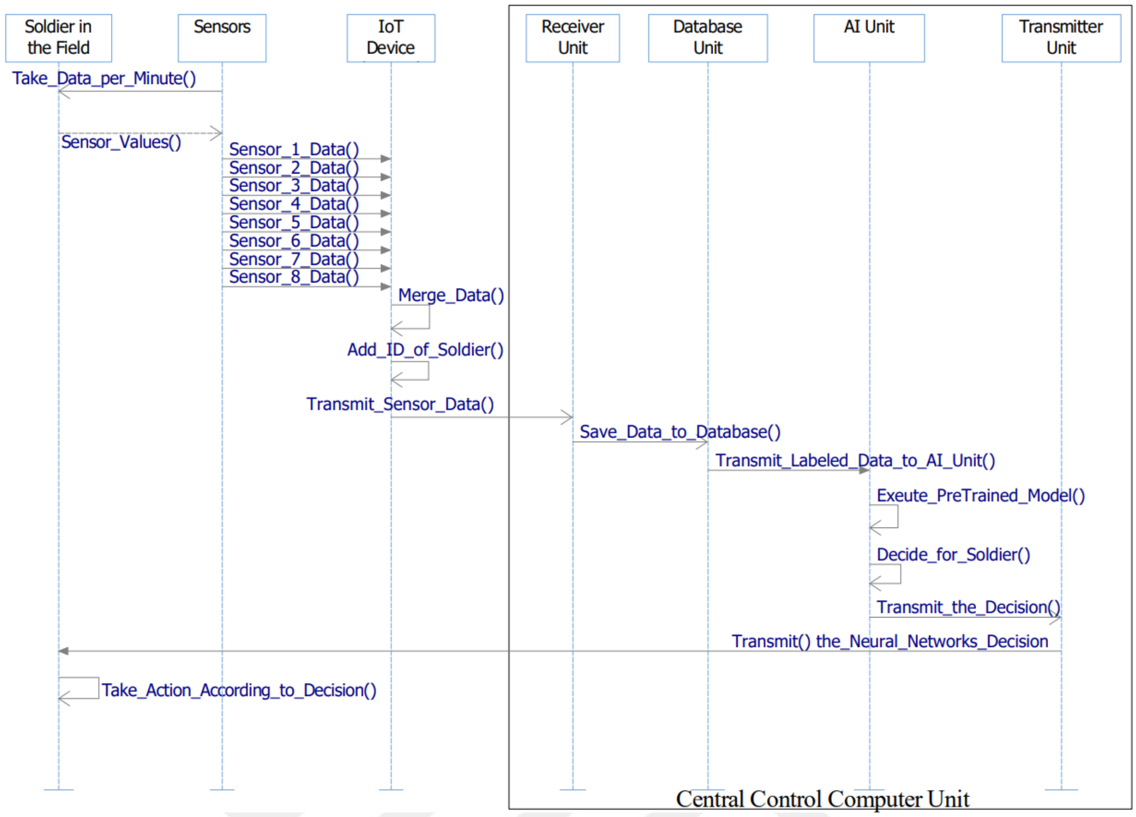


Figure 4. 2 – Use Case Model Sequence Diagram, General Flow

4.2.3. Discussion about RISC-V Software Implementations

The implementation of the RISC-V ISA on software has shown that this architecture can be used on computers. In this context, RISC-V ISA can be used by embedded systems and IoT devices which run operating systems and use processors. As can be seen in Table-4.1, it takes a long time for RISC-V ISA to run a neural network application and a complex image-processing algorithm using its own instructions. It took more than 134 minutes to filter a single image on a single image of the two-core RISC-V-based processor produced with the Chipyard SoC Generator built on the Linux operating system. It shows that performing complex operations directly on a processor manufactured using RISC-V does not make sense in terms of time performance. The reason for this situation is that although the RISC-V architecture uses an accelerator called Chipyard SoC Generator, Gemmini, current neural network applications perform hundreds of matrix convolution operations. It would take days for a single image to do this on a RISC-V-generated kernel.

On the other hand, the results obtained by simulating the structure created using the cross compiling method for the RISC-V ISA on a softcore processor and installing a Linux-based operating system on it, have a more innovative perspective as it is both more time-efficient and allows working on Python. Again, in this case, a neural network-based object recognition application will take a very long time to run for a single image, so a viable structure will not be formed.

In addition, it has been seen that it is possible to integrate the OpenCV architecture into the RISC-V ISA and to run neural network and image processing algorithms with C++. Running a widely used neural network and/or image-processing algorithm with softcore processors developed on RISC-V ISA and cross-compiling method is a more effective solution than RISC-V cores produced with Chipyard SoC Generator. However, in terms of performance, it is not an effective solution when compared to products manufactured by SiFive. The features of ready-made processors are shown in Table-4.8 [91] [104] [105]. Therefore, it would be more appropriate to use the RISC-V architecture for hardware driver, hardware accelerator and/or data transfer purposes instead of the main processor in large-scale operations that will run on a piece of hardware.

Table 4. 8 – RISC-V Processors

PicoRV32	**SiFive X280 Vector Processor**	**Microchip PolarFire**
1-stage pipeline	8-stages pipeline	5-stages pipeline
32-bit RISC-V ISA	64-bit RISC-V ISA	64-bit RISC-V ISA
4 Cores	8 Cores	5 Cores
Contains multiplier, divider, barrel shifter	32KB Level-1 Cache 256KB Level-2 Cache	2MB L2 Cache
AXI-4	AXI-4 256-bit	AXI-4 64-bit

4.2.4. Discussion about RISC-V Hardware Implementations

After seeing that the RISC-V ISA running as softcore on software causes performance losses in terms of time, it has been tested on hardware. In this context, studies on RISC-

V implementation on FPGA have been examined. Projects such as lowRISC, SweRV, NX25 and SiFive [106] [107] [108] are all about RISC-V buildings based on an FPGA model accordingly.

It has been seen that the studies are customizing the instructions required for a particular FPGA and that these studies can only work in selected FPGAs, which can be seen in Table-4.9. RISC-V based hardware processors and most of them have sufficient performance to perform neural network and/or image processing algorithms. However, since RISC-V-based processors are not widely used and are not used in computers or IoT devices with general use, the need for different RISC-V-based solutions on hardware has arisen.

Table 4. 9 – RISC-V Based FPGAs and Soft Processors

lowRISC	**SweRV**	**NX25**
University of Cambridge	Western Digital	Andes Technologies
Nexys4 FPGA	Nexys4 FPGA	GOWIN GW-2A FPGA or Corvette F1 Board Compatible
64-bit	32-bit	32-bit and 64-bit
Contains UART, SDCard, AXI4, Screen Monitor Support	Contains PCIE, AXI4	AXI4, Screen Monitor Support, First ThreadX port supported RISC-V

With the method described in Section-3.3.1, a viable method for every Xilinx architecture containing the AXI bus structure has been introduced. Neural network applications and image processing algorithms running on the processors developed on the software were thus run on the DPU unit; the RISC-V ISA managed the data transfer between the PL and PS units, in other words, it acted as a hardware accelerator for the DPU. After the core created with RISC-V ISA has been implemented, it is aimed to use the units on the FPGA instead of RISC-V peripheral and memory units by using the RISC-V AXI bus structure.

5. CONCLUSION

In this thesis, RISC-V ISA was analyzed with different implementation methods and implemented in software and hardware, and performance values were examined by running different applications. With the QEMU application, it has been shown that an operating system configured on ISA can operate more efficiently than a core structure produced with an SoC Generator. Creating a RISC-V based processor using the SoC Generator and comparing the application and code execution times of the RISC-V ISA with the cross compiling method, the method of realizing it in the RISC-V processor with AXI bus configuration on the FPGA is proposed. Accordingly, using the proposed method, fast data transfer has been achieved without the AXI buses responsible for data transmission on the FPGA being busy, and it has been shown that RISC-V architecture can work as a main processor, hardware accelerator and hardware driver. It has been shown that running a neural network or image processing application is faster in ARM architecture instead of RISC-V. Therefore, the neural network application running on the FPGA is not run on RISC-V, but on ARM architecture using RISC-V architecture. In addition, the proposed method worked together with a new method that uses advanced aspects of accepted applications in neural networking such as ResNet, VGGNet and DenseNet. The model taught on a dataset was first integrated into the computer environment and then into the FPGA and performance values were calculated. In terms of time and accuracy values, the use of RISC-V architecture as a driver and/or hardware accelerator turned out to be promising.

To summarize, it has been shown that RISC-V architecture offers its users a flexible development environment and can take on different roles on different platforms, as technology evolves, demanding more flexible applications and hardware in terms of performance critical operations and missions. Based on this study, it can be said that the RISC-V architecture will take on a structure that can be developed in the future to take on many operations that can facilitate today's processors.

In addition to this work, a structure can be developed for RISC-V that can work directly with convolution on matrices or with kernels of different sizes. Since the processor produced with the SoC generator used in the study works on a virtual machine or because the cross compiling method is tried in the QEMU simulator environment, a processor can be developed and converted into a real hardware. In addition, a RISC-V core can be run on the FPGA and different peripheral drivers can be written for this structure, allowing the hardware to be an IoT device.

In terms of performance improvements, the neural network application can be trained with more powerful datasets and longer epochs using lower learning rate values, lower batch sizes, and a customized convolution instruction for RISC-V ISA is considered feasible. It also showed that the use of more advanced peripherals instead of the RISC-V cache, RAM and ALU architecture offered as open-source using the RISC-V ISA improves the results and that the RISC-V ISA can be a competitor to the ARM architecture. On the other hand, since RISC-V architecture is open-source and open-standard, it is thought that developments will accelerate and hardware developments such as RISC-V-based controller, CPU, GPU will accelerate.

6. REFERENCES

[1] Internet of Things (IoT) and non-IoT active device connections worldwide from 2010 to 2025, https://www.statista.com/statistics/1101442/iot-number-of-connected-devices-worldwide/ (Date of Access: 2022-12-30)

[2] Number of Internet of Things (IoT) connected devices worldwide from 2019 to 2021, with forecasts from 2022 to 2030, https://www.statista.com/statistics/1183457/iot-connected-devices-worldwide/ (Date of Access: 2022-12-30)

[3] Mohammad, W., Elomri, A. and Kerbache, L., The Global Semiconductor Chip Shortage: Causes, Implications, and Potential Remedies, *IFAC-PapersOnLine*, 55(10), **(2022)** pp. 476–483.

[4] Celio, C., Patterson, D. A. and Asanović, K., The Berkeley Out-of-Order Machine (BOOM): An Industry-Competitive, Synthesizable, Parameterized RISC-V Processor., EECS Department, University of California, Berkeley, **(2015)**

[5] Apple Newsroom, Apple unveils M2, taking the breakthrough performance and capabilities of M1 even further, https://www.apple.com/newsroom/2022/06/apple-unveils-m2-with-breakthrough-performance-and-capabilities/ (Date of Access: 2022-12-30)

[6] Akram, A., A Study on the Impact of Instruction Set Architectures on Processor's Performance, Master Thesis, University of California, Davis, **2017**

[7] Harris, D. M., & Harris, S. L., Digital Design and Computer Architecture, Chapter 1 - From zero to one, Morgan Kaufmann Books, Massachusetts, USA, **(2007)** pp. 3–48.

[8] George, Alan D., An overview of RISC vs. CISC, [1990] Proceedings. The Twenty-Second Southeastern Symposium on System Theory, **(1990)** pp. 436–438.

[9] Bhandarkar, D. and Douglas W. C. "Performance from architecture: comparing a RISC and a CISC with similar hardware organization." ASPLOS IV **(1991)**

[10] Hennessy, J. L. and Patterson, D. A., Computer Architecture, Fifth Edition: A Quantitative Approach, Vol. 9, Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, **(2011)**

[11] Can Arm Survive RISC-V Challenge?, https://www.eetimes.eu/can-arm-survive-risc-v-challenge/ (Date of Access: 2022-12-30)

[12] Bibi, S. et al., A Review of ARM Processor Architecture History, Progress and Applications, 10, **(2021)** pp. 171–179.

[13] Waterman A., Lee Y., Patterson D.A. and Asanovic K., The RISC-V Instruction Set Manual, Volume I: Base User-Level ISA, Electrical Engineering and Computer Sciences

University of California at Berkeley, UCB/EECS-2016-118, **2016**

[14] Asanovic K. and Patterson D.A, Instruction Sets Should Be Free: The Case For RISC-V, Electrical Engineering and Computer Sciences University of California at Berkeley, UCB/EECS-2014-146, **2014**

[15] RISC-V International Members, https://riscv.org/members/ (Date of Access: 2022-12-30)

[16] Xu, Y. et al., Machine learning in construction: From shallow to deep learning, Developments in the Built Environment, Vol. 6 (**2021)** p. 100045.

[17] Samuel, A. L., Some studies in machine learning using the game of checkers, IBM Journal of Research and Development, 3(3) (**1959)** 211–229.

[18] Buscema, M., Back Propagation Neural Networks, Substance Use & Misuse, 33, (**1998**) 233–270.

[19] Ranganathan, V. and Natarajan, S., A New Backpropagation Algorithm without Gradient Descent, **(2018)** ArXiv, abs/1802.00027.

[20] Rokach, L. and Maimon, O., Decision Trees, The Data Mining and Knowledge Discovery Handbook, Vol.6 (**2005**) p. 165-192

[21] Schmidhuber, J., Deep learning in neural networks: An overview, Neural Networks, Elsevier BV, 61, pp. **(2015)** 85–117.

[22] Sarker, I. H., Machine Learning: Algorithms, Real-World Applications and Research Directions, SN Computer Science **(2021)**

[23] Schaefer, M., Michelin, L. and Kepner J., 'Naming Schema for a Human Brain-Scale Neural Network **(2021)**

[24] Herculano- Houzel, Suzana. “The Human Brain in Numbers: A Linearly Scaled-up Primate Brain.” Frontiers in Human Neuroscience 3 **(2009)**

[25] Schaefer, Morgan, Lauren Michelin and Jeremy Kepner. “Naming Schema for a Human Brain-Scale Neural Network.” ArXiv abs/2109.10951 **(2021)**

[26] Google Announces New Open-source OS for RISC-V Chips | Darshil Patel, All About Circuits, https://riscv.org/news/2022/10/google-announces-new-open-source-os-for-risc-v-chips-darshil-patel-all-about-circuits/, (Date of Access: 2023-01-25)

[27] NVidia Shuts Out RISC-V Software Support for GPUs, September 23, 2022, https://www.hpcwire.com/2022/09/23/nvidia-shuts-out-risc-v-software-support-for-gpus/, (Date of Access: 2023-01-25)

[28] Viola, P. A. and Jones M., Robust Real-time Object Detection, International Journal of Computer Vision (2001)

[29] LeCun, Y., Bottou, L., Bengio, Y. and Haffner, P., Gradient-based learning applied to document recognition, Proc. IEEE, 86, pp. **(1998)** 2278–2324

[30] Krizhevsky, A., Sutskever, I., & Hinton, G.E., ImageNet classification with deep convolutional neural networks, Communications of the ACM, 60 **(2012)** 84 - 90.

[31] Gonzalez, A., Korpan B., Zhao J., Replicating and Mitigating Spectre Attacks on a Open Source RISC-V Microarchitecture, **(2019)**

[32] Genç, H., Kim, S., Amid, A., Haj-Ali, A., Iyer, V., Prakash, P.R., Zhao, J., Grubb, D., Liew, H., Mao, H., Ou, A.J., Schmidt, C., Steffl, S., Wright, J.C., Stoica, I., Ragan-Kelley, J., Asanović, K., Nikolić, B. and Shao, Y.S., Gemmini: Enabling Systematic Deep-Learning Architecture Evaluation via Full-Stack Integration. 2021 58th ACM/IEEE Design Automation Conference (DAC), **(2019)**, 769-774.

[33] Amid, A., Biancolin, D., Gonzalez, A., Grubb, D., Karandikar, S., Liew, H., Magyar, A., Mao, H., Ou, A.J., Pemberton, N., Rigge, P., Schmidt, C., Wright, J.C., Zhao, J., Shao, Y.S., Asanović, K. and Nikolić, B., Chipyard: Integrated Design, Simulation, and Implementation Framework for Custom SoCs. IEEE Micro, 40 **(2020)** 10-21.

[34] Waterman, A., Lee, Y., Asanovic, K., The RISC-V Instruction Set Manual. Volume 1: User-Level ISA, Version 2.0. **(2019)** Document Version 20190608-Base-Ratified

[35] Garlati C., Pinto S., A Clean Slate Approach to Linux Security RISC-V Enclaves, **(2020)**

[36] Verboven, S., Bossche, R.V., Berghmans, O., Vanmechelen, K., & Broeckhove, J., Evaluating nested virtualization support, Parallel and distributed computing and networks Journal, **(2011)**

[37] Waterman A., Design of the RISC-V Instruction Set Architecture, Electrical Engineering and Computer Sciences University of California at Berkeley, UCB/EECS-2016-1, **(2016)**

[38] Waterman A., Lee Y., Patterson D.A., Asanovic K., The RISC-V Instruction Set Manual, Volume I: User-Level ISA, Electrical Engineering and Computer Sciences University of California at Berkeley, Version 2.1, UCB/EECS-2016-118, **(2016)**

[39] Hennessy J. L. and Patterson D. A., Computer Architecture, Oxford, England: Morgan Kaufmann, 5th Edition, **(2011)**

[40] Yul C, A Simple Project Paradigm for Teaching Computer Architecture, University of Texas Rio Grande Valley, Electrical and Computer Engineering Faculty Publications and Presentations, 6, **(2006)**

[41] Dulay, N, Memory Organization CPU Organisation and Operation Introduction to Assembly Programming, Imperial College London, Department of Computing Computer Systems 113 Architecture 110, **(2007)**

[42] Cheikh, A., Cerutti, G., Mastrandrea, A., Menichelli, F., and Olivieri, M., The microarchitecture of a multi-threaded RISC-V compliant processing core family for IoT end-nodes, Applications in Electronics Pervading Industry, Environment and Society, **(2017)**

[43] Zengin S., Systemc Implementation of a RISC-Based Microcontroller Architecture, Master Thesis, Middle East Technical University, Graduate School of Natural And Applied Sciences, **(2006)**

[44] Waterman, A., Asanovic, K., The RISC-V Instruction Set Manual, Volume I: User-Level ISA, Version 2.2, **(2017)**

[45] Patterson, D. A. and Hennessy J.L., Computer Organization and Design, Fifth Edition: The Hardware/Software Interface, **(2013)**

[46] Asanovic, K., Avizienis, R., Bachrach, J., Beamer, S., Biancolin, D., Celio, C., Cook, H., Dabbelt, P., Hauser, J.R., Izraelevitz, A.M., Karandikar, S., Keller, B., Kim, D., Koenig, J., Lee, Y., Love, E., Maas, M., Magyar, A., Mao, H., Moretó, M., Ou, A.J., Patterson, D.A., Richards, B.C., Schmidt, C., Twigg, S., Vo, H.D., and Waterman, A., The Rocket Chip Generator, **(2016)**

[47] Olanrewaju, R.F., Fajingbesi, F.E., Junaid, S.B., Alahudin, R., Anwar, F., and Pampori, B.R., Design and Implementation of a Five Stage Pipelining Architecture Simulator for RISC-16 Instruction Set, Indian journal of science and technology, 10, **(2017)**, 1-9.

[48] Genç, H., Haj-Ali, A., Iyer, V., Amid, A., Mao, H., Wright, J.C., Schmidt, C., Zhao, J., Ou, A.J., Banister, M., Shao, Y.S., Nikolić, B., Stoica, I. and Asanović, K., Gemmini: An Agile Systolic Array Generator Enabling Systematic Evaluations of Deep-Learning Architectures, **(2019)**

[49] Bellard, F., QEMU, a fast and portable dynamic translator, USENIX Annual Technical Conference, **(2005)**, pp. 41–46.

[50] Ongsulee, P., Artificial intelligence, machine learning and deep learning, 2017 15th International Conference on ICT and Knowledge Engineering (ICT&KE) **(2017)** pp. 1-6

[51] Deng, J., Dong, W., Socher, R., Li, L., Li, K., & Fei-Fei, L., ImageNet: A large-scale hierarchical image database. 2009 IEEE Conference on Computer Vision and Pattern Recognition, **(2009)** pp. 248-255.

[52] Janiesch, C., Patrick Z. and Kai H., Machine learning and deep learning, Electronic Markets 31 **(2021)** pp. 685-695.

[53] Nasteski, V., An overview of the supervised machine learning methods, Horizons 4 **(2017)** pp. 51-62.

[54] Schmidhuber, J., Deep learning in neural networks: An overview, Neural networks: the official journal of the International Neural Network Society 61 **(2014)** pp. 85-117 .

[55] Buscema, M., Back propagation neural networks, Substance use & misuse, 33-2 **(1998)** pp. 233-70.

[56] O'Shea, K. and Ryan N., An Introduction to Convolutional Neural Networks, ArXiv abs/1511.08458 **(2015)**

[57] Wang, S., A Review of Gradient-Based and Edge-Based Feature Extraction Methods for Object Detection, 2011 IEEE 11th International Conference on Computer and Information Technology **(2011)** pp. 277-282.

[58] Hohman, F., Haekyu P., Caleb R. and Duen H. C., Summit: Scaling Deep Learning Interpretability by Visualizing Activation and Attribution Summarizations, IEEE Transactions on Visualization and Computer Graphics 26 **(2019)** pp. 1096-1106.

[59] Dumoulin, V. and Visin, F., A guide to convolution arithmetic for deep learning, ArXiv, abs/1603.07285 **(2016)**

[60] Gholamalinezhad, H. and Khosravi H., Pooling Methods in Deep Neural Networks, a Review, ArXiv abs/2009.07485 **(2020)**

[61] Tammina, Srikanth. "Transfer learning using VGG-16 with Deep Convolutional Neural Network for Classifying Images." International Journal of Scientific and Research Publications (IJSRP) Vol.9, (2019), pp.9420

[62] Murthy, C.B., Hashmi, M.F., Bokde, N.D., Geem, Z.W., Investigations of Object Detection in Images/Videos Using Various Deep Learning Techniques and Embedded Platforms, A Comprehensive Review **(2020)** Applied Sciences 10, 3280.

[63] Xiao, Y., Zhiqiang T., Jiachen Y., Yinshu Z., Shuai L., Du S. and Xuguang L., A review of object detection based on deep learning, Multimedia Tools and Applications 79, 23729–23791 **(2020)** pp. 1-63.

[64] Zou, Z., Shi, Z., Guo, Y. and Ye, J., Object Detection in 20 Years: A Survey, ArXiv abs/1905.05055 (2019).

[65] Du, M., Ninghao, L. and Xia, H., Techniques for interpretable machine learning, Communications of the ACM, 63, **(2018)** pp. 68 - 77.

[66] Large Scale Visual Recognition Challenge 2012 (ILSVRC2012), https://www.image-net.org/challenges/LSVRC/2012/results.html (Date of Access: 2022-12-30)

[67] Krizhevsky A., Sutskever I., and Geoffrey E. H., ImageNet classification with deep convolutional neural networks, In Proceedings of the 25th International Conference on Neural Information Processing Systems, Volume 1 (NIPS'12), Curran Associates Inc., Red Hook, NY, USA, **(2012)** pp. 1097–1105.

[68] Agarap, A. F., Deep Learning using Rectified Linear Units (ReLU), ArXiv abs/1803.08375 **(2018)**

[69] Chen, L., Shaobo L., Qiang B., Jing Y., Sanlong J. and Yanming M., Review of Image Classification Algorithms Based on Convolutional Neural Networks, Remote. Sens. 13 **(2021)** 4712.

[70] Girshick, R. B., Jeff D., Trevor D. and Jitendra M., Rich Feature Hierarchies for Accurate Object Detection and Semantic Segmentation, 2014 IEEE Conference on Computer Vision and Pattern Recognition, **(2013)** pp. 580-587.

[71] Simonyan, K., & Zisserman, A., Very Deep Convolutional Networks for Large-Scale Image Recognition, CoRR, abs/1409.1556, **(2014)**

[72] Szegedy, C., Liu, W., Jia, Y., Sermanet, P., Reed, S.E., Anguelov, D., Erhan, D., Vanhoucke, V., & Rabinovich, A., Going deeper with convolutions, 2015 IEEE Conference on Computer Vision and Pattern Recognition (CVPR), **(2014)** pp. 1-9.

[73] He, K., Zhang, X., Ren, S., & Sun, J., Spatial Pyramid Pooling in Deep Convolutional Networks for Visual Recognition, IEEE Transactions on Pattern Analysis and Machine Intelligence, 37, **(2014)** pp. 1904-1916.

[74] Girshick, R. B., Fast R-CNN, 2015 IEEE International Conference on Computer Vision (ICCV), **(2015)** pp. 1440-1448.

[75] He, K., Zhang, X., Ren, S. and Sun, J., Deep Residual Learning for Image Recognition, 2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR), **(2014)** pp. 770-778.

[76] He, K., Zhang, X., Ren, S. and Sun, J., Delving Deep into Rectifiers: Surpassing Human-Level Performance on ImageNet Classification, 2015 IEEE International Conference on Computer Vision (ICCV) **(2015)** pp. 1026-1034.

[77] Leong, M.C., Prasad, D., Lee, Y.T. and Lin, F., Semi-CNN Architecture for Effective Spatio-Temporal Learning in Action Recognition, Applied Sciences. *10*(2) **(2019)** pp. 557

[78] Pascanu, R., Mikolov T. and Bengio Y., On the difficulty of training recurrent neural networks, International Conference on Machine Learning, Vol.28, (ICML'13), **(2012)** pp.1310-1318

[79] Huang, G., Liu, Z. and Weinberger, K. Q., Densely Connected Convolutional Networks, 2017 IEEE Conference on Computer Vision and Pattern Recognition (CVPR), **(2016)**, pp. 2261-2269.

[80] AMBA AXI and ACE Protocol Specification, AXI3, AXI4, and AXI4-Lite, ARM, http://www.gstitt.ece.ufl.edu/courses/fall15/eel4720_5721/labs/refs/AXI4_specification.pdf (Date of Access: 2022-12-30)

[81] AMBA AXI4 Interface Protocol, https://www.xilinx.com/products/intellectual-property/axi.html (Date of Access: 2022-12-30)

[82] RISC-V GNU Compiler Toolchain, https://github.com/riscv-collab/riscv-gnu-toolchain (Date of Access: 2022-12-30)

[83] Stallman, R.M. and GCCDeveloperCommunity, Using The Gnu Compiler Collection: A Gnu Manual For Gcc, Version 4.3.3, **(2009)**

[84] riscv-tools, Spike, riscv-pk, https://github.com/ucb-bar/esp-tools, (Date of Access: 2022-12-30)

[85] Wilson, S., Verilator Release 5.005, https://veripool.org/ftp/verilator_doc.pdf (Date of Access: 2022-12-30)

[86] Chipyard's documentation, Chipyard Basics, Simulation, Included RTL Generators (version "1.8.1"), https://chipyard.readthedocs.io/en/stable/index.html (Date of Access: 2022-12-30)

[87] pip - The Python Package Installer, https://github.com/pypa/pip (Date of Access: 2022-12-30)

[88] Dalal, N. and Triggs, B., Histograms of oriented gradients for human detection, 2005 IEEE Computer Society Conference on Computer Vision and Pattern Recognition (CVPR'05) 1 **(2005)** pp. 886-893, vol.1

[89] ZCU102 Evaluation Board, User Guide, UG1182 (v1.6) June 12, 2019, https://docs.xilinx.com/v/u/en-US/ug1182-zcu102-eval-bd (Date of Access: 2022-12-30)

[90] Kazaz, T., Praet, C.V., Kulin, M., Willemen, P. and Moerman, I., Hardware Accelerated SDR Platform for Adaptive Air Interfaces, ETSI Workshop On Future Radio Technologies – Air Interfaces, ArXiv, abs/1705.00115, **(2017)**

[91] PicoRV32 - A Size-Optimized RISC-V CPU, https://github.com/YosysHQ/picorv32 (Date of Access: 2022-12-30)

[92] DPUCZDX8G for Zynq UltraScale+ MPSoCs Product Guide (PG338), Xilinx Documentation Portal, 2022-06-24, v.4.0., https://docs.xilinx.com/r/en-US/pg338-dpu/Introduction?tocId=Bd4R4bhnWgMYE6wUISXDLw (Date of Access: 2022-12-30)

[93] Python Productivity for Zynq, (Pynq) Documentation, PS/PL Interfaces, https://pynq.readthedocs.io/en/v2.3/overlay_design_methodology/pspl_interface.html (Date of Access: 2022-12-30)

[94] AXI Interconnect v2.1 LogiCORE IP Product Guide Vivado Design Suite, https://www.xilinx.com/content/dam/xilinx/support/documents/ip_documentation/axi_interconnect/v2_1/pg059-axi-interconnect.pdf (Date of Access: 2022-12-30)

[95] Mitic, M. and Stojcev, M.K., An Overview of On-Chip Buses, Facta Universitatis, Series electronics and energetics, 19 **(2006)** pp.405-428.

[96] SiFive TileLink Specification, https://static.dev.sifive.com/docs/tilelink/tilelink-spec-1.7-draft.pdf, (Date of Access: 2022-12-30)

[97] MacBook Air (M2, 2022) - Technical Specifications, Year introduced: 2022, https://support.apple.com/kb/SP869?locale=en_US (Date of Access: 2022-12-30)

[98] Zynq UltraScale+ MPSoC Processing System Product Guide (PG201), 2022-05-11 v.3.4.,https://docs.xilinx.com/r/en-US/pg201-zynq-ultrascale-plus-processing-system/ Zynq-UltraScale-MPSoC-Processing-System-v3.4-LogiCORE-IP-Product-Guide-PG201 (Date of Access: 2022-12-30)

[99] Versal ACAP Technical Reference Manual (AM011), 2022-12-12, v.1.5. https://docs.xilinx.com/r/en-US/am011-versal-acap-trm/LPD-Reference-Clocks (Date of Access: 2022-12-30)

[100] Krizhevsky, A. and Hinton, G., Learning multiple layers of features from tiny images, Toronto, Ontario: University of Toronto, **(2009)**.

[101] Shi, H.; Zhao, H.; Liu, Y.; Gao, W. Dou, S., Systematic Analysis of a Military Wearable Device Based on a Multi-Level Fusion Framework: Research Directions. Sensors, **(2019)**, 19, 2651

[102] Callan, C., Goodman, J., Cornwall, M., Fortson, N., Henderson, R., Katz, J., Long, Darrell, Muller, R., Ruderman, M., Vesecky, J., Sensors to Support the Soldier, **(2005)**, 106.

[103] Nişancı G, Flikkema PG, Yalçın T. Symmetric Cryptography on RISC-V: Performance Evaluation of Standardized Algorithms. Cryptography. (**2022**), 6(3):41.

[104] Introduction to the SiFive Intelligence X280, June 01, 2022, https://www.sifive.com/blog/introduction-to-the-sifive-intelligence-x280 (Date of Access: 2022-12-30)

[105] Microchip PolarFire SoC Product Overview, https://ww1.microchip.com/downloads/en/DeviceDoc/Polarfire_SOC_Product_Overview.pdf (Date of Access: 2022-12-30)

[106] Mohseni, Z., and Reviriego. P., Reliability Characterization And Activity Analysis Of Lowrisc Internal Modules Against Single Event Upsets Using Fault Injection And RTL Simulation, Microprocess, Microsystems, 71 **(2019)** 102871.

[107] Marena, T., RISC-V: high performance embedded SweRV core microarchitecture, performance and CHIPS Alliance, Western Digital Corporation **(2019)**

[108] Su, C., Andes RISC-V Processor Solutions for Diversified Applications, 2019/05/09 RISC-V CON Hsinchu **(2019)**

APPENDICES

Appendix 1 – Thesis Work Codes

C++ Image Filtering Code

```
#include <stdio.h>
#include <stdlib.h>
#define KERNEL_WIDTH 3
#define KERNEL_HEIGHT 3
#define WIDTH 5
#define HEIGHT 5

typedef struct Image {
  int width;
  int height;
  int** pixels;
} Image;

Image* create_image(int width, int height) {
      Image* image = (Image*)malloc(sizeof(Image));
      image->width = width;
      image->height = height;
      image->pixels = (int**)malloc(height * sizeof(int*));
      for (int i = 0; i < height; i++) {
            image->pixels[i] = (int*)malloc(width * sizeof(int));
            }
      return image;}

// convolution kernel
int kernel[3][3] = {
  {1, 0, 1},
  {1, 0, 1},
  {0, 1, 0}
};

// image data
Image = fopen("image_to_matrix.txt", "r");

void convolve3x3(const Image* input, Image* output, const int*
kernel) {
      for (int y = 0; y < input->height; y++) {
            for (int x = 0; x < input->width; x++) {
            int sum = 0;
                  for (int ky = -1; ky <= 1; ky++) {
                        for (int kx = -1; kx <= 1; kx++) {
                        int iy = y + ky;
                        int ix = x + kx;
if (iy >= 0 && iy < input->height && ix >= 0 && ix < input->width) {
      sum += input->pixels[iy][ix] * kernel[ky + 1][kx + 1];
                        }}}
            output->pixels[y][x] = sum;
```

```
      } } }
int main() {
  Image* input = create_image(600, 600);
  Image* output = create_image(600, 600);
      //2d image with width and height
  for (int y = 0; y < input->height; y++) {
    for (int x = 0; x < input->width; x++) {
      input->pixels[y][x] = rand();
    }
  }
  // Define the kernel for the convolution
  int img_kernel[KERNEL_HEIGHT][KERNEL_WIDTH] = {
    {1, 2, 1},
    {2, 4, 2},
    {1, 2, 1}
  };
  convolve3x3(input, output, kernel);
  return 0;
}
int main() {
  // output image
  int output[WIDTH][HEIGHT];

  // perform convolution
  for (int y = 0; y < HEIGHT; y++) {
    for (int x = 0; x < WIDTH; x++) {
      int sum = 0;
      for (int ky = -1; ky <= 1; ky++) {
        for (int kx = -1; kx <= 1; kx++) {
          // handle image boundaries
          int yy = y + ky;
          int xx = x + kx;
          if (yy < 0) { yy = 0; }
          if (yy >= HEIGHT) { yy = HEIGHT - 1; }
          if (xx < 0) { xx = 0; }
          if (xx >= WIDTH) { xx = WIDTH - 1; }
          // perform convolution
          sum += image[xx][yy] * img_kernel[kx + 1][ky + 1];
        }
      }
      output[x][y] = sum;
    }
  }

  // print output image
  for (int y = 0; y < HEIGHT; y++) {
    for (int x = 0; x < WIDTH; x++) {
      printf("%d ", output[x][y]);
    }
    printf("\n");
  }
  return 0;
}
```

Python Image Convolution Code

```
import time
start_time = time.time()

from PIL import Image, ImageFilter

# Open the image and convert it to grayscale
image = Image.open('blk.png').convert('L')

# Define the convolution kernel
kernel = [-1, -1, -1, -1, 8, -1, -1, -1, -1]
# Apply the convolution with the 'convolve' filter
image = image.filter(ImageFilter.Kernel((3, 3), kernel, scale=1,
offset=0))

# Save the convolved image
image.save('image_convolved.jpg')

print("--- %s seconds ---" % (time.time() - start_time))
```

C++ Edge Detection Code

```
#include<iostream>
using namespace std;

int main(int argc, char * argv []) {
        cv::Mat imgOriginal;
        cv::Mat imgGrayscale;
        cv::Mat imgBlurred;
        cv::Mat imgCanny;
        std::string img_addr = argv[1];
        std::cout << "loaded image  " + img_addr << std::endl;
        imgOriginal = cv::imread(img_addr);
        if (imgOriginal.empty() ) {
                std::cout << "error: image not read from file \n\n";
                return 0; }
        cv::cvtColor(imgOriginal, imgGrayscale, cv::COLOR_BGR2GRAY);
        cv::GaussianBlur(imgGrayscale, imgBlurred, cv::Size(3,3),
1.2);
        cv::Canny(imgBlurred, imgCanny, 100, 200);
        cv::imwrite("img_Edge.jpg", imgCanny);
        return 0;
        }
```

Matlab Image Processing Code

```
clc;
clear all;
close all;

%%
```

```
%IMREAD
tic
img=imread("goldfish.JPEG");
figure;
imshow(img)
%%
%RESIZE
img=imresize(img, [600,600]);
figure;
imshow(img)
%%
%RGB2GRAY IMAGE
img_rgb_2=rgb2gray(img);
figure;
imshow(img_rgb_2);

%%
%PRE-SETTINGS
GAUSE_INDEX=0.6;
MED_INDEX=[13 13];
BIN_INDEX=0.48; %48;
STREL_INDEX=5;
STEL_PARAM='disk';

%%
%GAUSSIAN FILTER
C=imgaussfilt(img_rgb_2,GAUSE_INDEX);
figure;
imshow(C)

%%
%MEDIAN FILTER
J=medfilt2(C);
J2=medfilt2(C,MED_INDEX);
figure;
montage ({J,J2});
title("med filt")
%%
%BINARY IMAGE
bin1=im2bw(J,BIN_INDEX);
bin2=im2bw(J2,BIN_INDEX);
figure;
montage ({bin1,bin2});
title("bin ")

%%
%IMAGE OPENING
se=strel(STEL_PARAM,5);
open1=imopen(bin1,se);
open2=imopen(bin2,se);
figure;
montage ({open1,open2});
title("open")
```

```
%%
%HISTOGRAM CHECK
H1=histogram(img_rgb_2);
R1=sum(H1.Values(find(H1.BinEdges > H1.NumBins/2)-1));
L1=sum(H1.Values(find(H1.BinEdges <= H1.NumBins/2)));
if R1>L1
    open1=1-open1;
    open2=1-open2;
elseif R1<L1
    open1=open1;
    open2=open2;
else
    error("Undefined Object");
end
figure;
montage ({open1,open2});
title("Open After Histogram")

%%
%BWLABEL
bw1=bwlabel(open1,8);
[bw2,n2]=bwlabel(open2,8);
figure; montage({bw1, bw2});
title("bwlabel")

%%
%BWLABEL COMPARE
for i = 1:1:n2
    param1(i,1) = size(find(bw2 == i),1);
end
param2=find(param1 == max(param1))
bw1(bw1 ~= param2) = 0;
bw1(bw1 == param2) = 1;
bw2(bw2 ~= param2) = 0;
bw2(bw2 == param2) = 1;
figure; montage({bw1, bw2});
title("bwlabel after compare")

%%
%IMFILL
fill1=imfill(bw1,'holes');
fill2=imfill(bw2,'holes');
figure;
montage ({fill1,fill2});
title("fill")

%%
%CLEARBORDER
imclbr1=imclearborder(fill1,8);
imclbr2=imclearborder(fill2,8);
figure;
montage ({imclbr1,imclbr2});
```

```
title("noboard")

%%
%BOUNDING BOX
select=regionprops(imclbr2, 'BoundingBox');
bboxCoords = reshape([select.BoundingBox], 4, []).';

%top-bottom-right-left
top_Left = bboxCoords(:,1:2);
top_Right = [top_Left(:,1) + bboxCoords(:,3) top_Left(:,2)];
bottom_Left = [top_Left(:,1) top_Left(:,2) + bboxCoords(:,4)];
bottom_Right = [top_Left(:,1) + bboxCoords(:,3) top_Left(:,2) +
bboxCoords(:,4)];
Coordinates = [top_Left; top_Right; bottom_Left; bottom_Right];
%max-min
minX = min(Coordinates(:,1));
maxX = max(Coordinates(:,1));
minY = min(Coordinates(:,2));
maxY = max(Coordinates(:,2));
%width-height
w = (maxX - minX + 1);
h = (maxY - minY + 1);
%box sketch
box1 = [minX minY w h];

imshow(img);
hold on;
rectangle('Position', box1, 'EdgeColor', 'blue',LineWidth=2);

img_crop=imcrop(img,box1);
param2=size(img_crop,1);
N=size(img_crop,2);

threshold=2;

if (max(param2,N)/min(param2,N))>threshold
    a=max(size(rgb2gray(img_crop)));
% pad the image
    pad=padarray(img_crop, [floor((a-param2)/2) floor((a-N)/2)],
'replicate','post');
    pad=padarray(pad, [ceil((a-param2)/2) ceil((a-N)/2)],
'replicate','pre');
    figure;imshow(pad)
elseif (max(param2,N)/min(param2,N)) <= threshold
    pad=imresize(img_crop, [400,400]);
else
    error("padding error");
end

%%
net=nasnetlarge %selected net

image_new=imclbr1;
```

```
%image_new=imclbr2;
image_new=imresize(image_new, [331,331]);

net.Layers

I_double = double(image_new);

% Adjust size of the image
sz = net.Layers(1).InputSize
I_double = I_double(1:sz(1),1:sz(2),1:sz(3));

% Classify the image using ResNet-18
label = classify(net, image_new)

% Show the image and the classification results
figure
imshow(img)
text(10,20,char(label),'Color','red', 'FontSize',24,'FontWeight',
'Bold')

m = 331; n = 3 ; p = 331 ; % NasNet input Dimensions
F = [repmat(' %0.0f',1,331),'\n'];  % NasNet format

toc
```

FPGA Verilog Code for AXI Interface- Module Definer

```
module RISCVCORE1(
   input S00_AXI_awid,
   input S00_AXI_awaddr,
   input S00_AXI_awlen,
   input [2:0] S00_AXI_awsize,
   input [1:0] S00_AXI_awburst,
   input S00_AXI_awlock,
   input [3:0] S00_AXI_awcache,
   input [2:0] S00_AXI_awprot,
   input [3:0] S00_AXI_awregion,
   input [3:0] S00_AXI_awqos,
   input S00_AXI_awuser,
   input [0:0] S00_AXI_awvalid,
   output [0:0] S00_AXI_awready,
   input S00_AXI_wid,
   input S00_AXI_wdata,
   input S00_AXI_wstrb,
   input [0:0] S00_AXI_wlast,
   input S00_AXI_wuser,
   output [0:0] S00_AXI_wvalid,
   output [0:0] S00_AXI_wready,
   output S00_AXI_bid,
   output [1:0] S00_AXI_bresp,
   output S00_AXI_buser,
   input [0:0] S00_AXI_bvalid,
   input [0:0] S00_AXI_bready,
```

```
input S00_AXI_arid,
input S00_AXI_araddr,
input S00_AXI_arlen,
input [2:0] S00_AXI_arsize,
input [1:0] S00_AXI_arburst,
input S00_AXI_arlock,
input [3:0] S00_AXI_arcache,
input [2:0] S00_AXI_arprot,
input [3:0] S00_AXI_arregion,
input [3:0] S00_AXI_arqos,
input S00_AXI_aruser,
input [0:0] S00_AXI_arvalid,
output [0:0] S00_AXI_arready,
output S00_AXI_rid,
output S00_AXI_rdata,
output [1:0] S00_AXI_rresp,
output [0:0] S00_AXI_rlast,
output S00_AXI_ruser,
output [0:0] S00_AXI_rvalid,
output [0:0] S00_AXI_rready,


output [15:0] M00_AXI_awid,
output [31:0] M00_AXI_awaddr,
output [7:0] M00_AXI_awlen,
output [2:0] M00_AXI_awsize,
output [1:0] M00_AXI_awburst,
output [1:0] M00_AXI_awlock,
output [3:0] M00_AXI_awcache,
output [2:0] M00_AXI_awprot,
output [3:0] M00_AXI_awregion,
output [3:0] M00_AXI_awqos,
output [15:0] M00_AXI_awuser,
output [0:0] M00_AXI_awvalid,

input [0:0] M00_AXI_awready,

output [15:0] M00_AXI_wid,
output [31:0] M00_AXI_wdata,
output [3:0] M00_AXI_wstrb,
output [0:0] M00_AXI_wlast,
output [0:0] M00_AXI_wvalid,

input [0:0] M00_AXI_wready,
input [15:0] M00_AXI_bid,
input [1:0] M00_AXI_bresp,
input [0:0] M00_AXI_bvalid,

output [0:0] M00_AXI_bready,
output [15:0] M00_AXI_arid,
output [31:0] M00_AXI_araddr,
output [7:0] M00_AXI_arlen,
output [2:0] M00_AXI_arsize,
output [1:0] M00_AXI_arburst,
```

```
    output [1:0] M00_AXI_arlock,
    output [3:0] M00_AXI_arcache,
    output [2:0] M00_AXI_arprot,
    output [3:0] M00_AXI_arregion,
    output [3:0] M00_AXI_arqos,
    output [15:0] M00_AXI_aruser,
    output [0:0] M00_AXI_arvalid,

    input [0:0] M00_AXI_arready,
    input [15:0] M00_AXI_rid,
    input [31:0] M00_AXI_rdata,
    input [1:0] M00_AXI_rresp,
    input [0:0] M00_AXI_rlast,
    input [0:0] M00_AXI_rvalid,

    output [0:0] M00_AXI_rready,


   input clk,
   input rst
    );

endmodule
```

Python Code for New Object Detection Code

```
import tensorflow as tf

# Display the version
print(tf.__version__)

# other imports
import numpy as np
import matplotlib.pyplot as plt
from tensorflow.keras.layers import Input, Conv2D, Dense, Flatten,
Dropout
from tensorflow.keras.layers import GlobalMaxPooling2D, MaxPooling2D
from tensorflow.keras.layers import BatchNormalization
from tensorflow.keras.models import Model

cifar10 = tf.keras.datasets.cifar10

(x_train, y_train), (x_test, y_test) = cifar10.load_data()
print(x_train.shape, y_train.shape, x_test.shape, y_test.shape)

x_train, x_test = x_train / 255.0, x_test / 255.0
y_train, y_test = y_train.flatten(), y_test.flatten()


def identity_block(x, filter):
    # copy tensor to variable called x_skip
    x_skip = x
    # Layer 1
    x = tf.keras.layers.Conv2D(filter, (3,3), padding = 'same')(x)
```

```
    x = tf.keras.layers.BatchNormalization(axis=3)(x)
    x = tf.keras.layers.Activation('relu')(x)
    # Layer 2
    x = tf.keras.layers.Conv2D(filter, (3,3), padding = 'same')(x)
    x = tf.keras.layers.BatchNormalization(axis=3)(x)
    # Add Residue
    x = tf.keras.layers.Add()([x, x_skip])
    x = tf.keras.layers.Activation('relu')(x)
    return x

def convolutional_block(x, filter):
    # copy tensor to variable called x_skip
    x_skip = x
    # Layer 1
    x = tf.keras.layers.Conv2D(filter, (3,3), padding = 'same',
strides = (2,2))(x)
    x = tf.keras.layers.BatchNormalization(axis=3)(x)
    x = tf.keras.layers.Activation('relu')(x)
    # Layer 2
    x = tf.keras.layers.Conv2D(filter, (3,3), padding = 'same')(x)
    x = tf.keras.layers.BatchNormalization(axis=3)(x)
    # Processing Residue with conv(1,1)
    x_skip = tf.keras.layers.Conv2D(filter, (1,1), strides =
(2,2))(x_skip)
    # Add Residue
    x = tf.keras.layers.Add()([x, x_skip])
    x = tf.keras.layers.Activation('relu')(x)
    return x

def NewModel(shape = (32, 32, 3), classes = 10):
    x_input = tf.keras.layers.Input(shape)
    x = tf.keras.layers.ZeroPadding2D((3, 3))(x_input)

    x = tf.keras.layers.Conv2D(32, (3, 3), padding='same')(x)
    x = tf.keras.layers.BatchNormalization(axis=3)(x)
    x = tf.keras.layers.Activation('relu')(x)

    x = tf.keras.layers.Conv2D(32, (3, 3), padding='same')(x)
    x = tf.keras.layers.BatchNormalization(axis=3)(x)
    x = tf.keras.layers.Activation('relu')(x)

    filter_size = 32
    x = identity_block(x, filter_size)

    x = tf.keras.layers.Conv2D(64, (3, 3), padding='same')(x)
    x = tf.keras.layers.BatchNormalization(axis=3)(x)
    x = tf.keras.layers.Activation('relu')(x)

    filter_size = 64
    x = identity_block(x, filter_size)
    x = convolutional_block(x, filter_size)

    filter_size=128
```

```
    x = tf.keras.layers.Conv2D(128, (3, 3), padding='same')(x)
    x = tf.keras.layers.BatchNormalization(axis=3)(x)
    x = tf.keras.layers.Activation('relu')(x)
    x = identity_block(x, filter_size)
    x = convolutional_block(x, filter_size)

    filter_size=256
    x = tf.keras.layers.Conv2D(256, (3, 3), padding='same')(x)
    x = tf.keras.layers.BatchNormalization(axis=3)(x)
    x = tf.keras.layers.Activation('relu')(x)
    x = identity_block(x, filter_size)
    x = convolutional_block(x, filter_size)

    x = tf.keras.layers.AveragePooling2D((2,2), padding = 'same')(x)
    x = tf.keras.layers.Flatten()(x)
    x = tf.keras.layers.Dense(512, activation = 'relu')(x)
    x = tf.keras.layers.Dense(classes, activation = 'softmax')(x)
    model = tf.keras.models.Model(inputs = x_input, outputs = x, name
= "NewModel")
    return model

from keras.utils import to_categorical, plot_model
model = NewModel()
plot_model(model, to_file='new-net.png')

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


r = model.fit(x_train, y_train, validation_data=(x_test, y_test),
epochs=50)

batch_size = 32
data_generator = tf.keras.preprocessing.image.ImageDataGenerator(
    width_shift_range=0.1, height_shift_range=0.1,
horizontal_flip=True)


train_generator = data_generator.flow(x_train, y_train, batch_size)
steps_per_epoch = x_train.shape[0] // batch_size

r = model.fit(train_generator, validation_data=(x_test, y_test),
              steps_per_epoch=steps_per_epoch, epochs=50)

plt.plot(r.history['accuracy'], label='acc', color='red')
plt.plot(r.history['val_accuracy'], label='val_acc', color='green')
plt.legend()

model.save('newmodel.h5')
```

Appendix 2 – Model Design

Model: "NewModel"
Layer (type) Output Shape Param # Connected to
input_1 (InputLayer) [(None, 32, 32, 3)] 0
zero_padding2d (ZeroPadding2D) (None, 38, 38, 3) 0 ['input_1[0][0]']
conv2d (Conv2D) (None, 19, 19, 1) 28 ['zero_padding2d[0][0]']
conv2d_1 (Conv2D) (None, 19, 19, 64) 640 ['conv2d[0][0]']
batch_normalization (BatchNormalization (None, 19, 19, 64) 256 ['conv2d_1[0][0]']
activation (Activation) (None, 19, 19, 64) 0 ['batch_normalization[0][0]']
conv2d_2 (Conv2D) (None, 19, 19, 64) 36928 ['activation[0][0]']
batch_normalization_1 (BatchNormalization (None, 19, 19, 64) 256 ['conv2d_2[0][0]']
activation_1 (Activation) (None, 19, 19, 64) 0 ['batch_normalization_1[0][0]']
conv2d_3 (Conv2D) (None, 19, 19, 64) 36928 ['activation_1[0][0]']
batch_normalization_2 (BatchNormalization (None, 19, 19, 64) 256 ['conv2d_3[0][0]']
activation_2 (Activation) (None, 19, 19, 64) 0 ['batch_normalization_2[0][0]']
conv2d_4 (Conv2D) (None, 19, 19, 64) 36928 ['activation_2[0][0]']
batch_normalization_3 (BatchNormalization (None, 19, 19, 64) 256 ['conv2d_4[0][0]']
add (Add) (None, 19, 19, 64) 0 ['batch_normalization_3[0][0]',
'activation_1[0][0]']
activation_3 (Activation) (None, 19, 19, 64) 0 ['add[0][0]']
conv2d_5 (Conv2D) (None, 19, 19, 64) 36928 ['activation_3[0][0]']
batch_normalization_4 (BatchNormalization (None, 19, 19, 64) 256 ['conv2d_5[0][0]']
activation_4 (Activation) (None, 19, 19, 64) 0 ['batch_normalization_4[0][0]']
conv2d_6 (Conv2D) (None, 19, 19, 64) 36928 ['activation_4[0][0]']
batch_normalization_5 (BatchNormalization (None, 19, 19, 64) 256 ['conv2d_6[0][0]']
add_1 (Add) (None, 19, 19, 64) 0 ['batch_normalization_5[0][0]',
'activation_3[0][0]']
activation_5 (Activation) (None, 19, 19, 64) 0 ['add_1[0][0]']
conv2d_7 (Conv2D) (None, 19, 19, 64) 36928 ['activation_5[0][0]']
batch_normalization_6 (BatchNormalization (None, 19, 19, 64) 256 ['conv2d_7[0][0]']
activation_6 (Activation) (None, 19, 19, 64) 0 ['batch_normalization_6[0][0]']
conv2d_8 (Conv2D) (None, 19, 19, 64) 36928 ['activation_6[0][0]']
batch_normalization_7 (BatchNormalization (None, 19, 19, 64) 256 ['conv2d_8[0][0]']
add_2 (Add) (None, 19, 19, 64) 0 ['batch_normalization_7[0][0]',
'activation_5[0][0]']

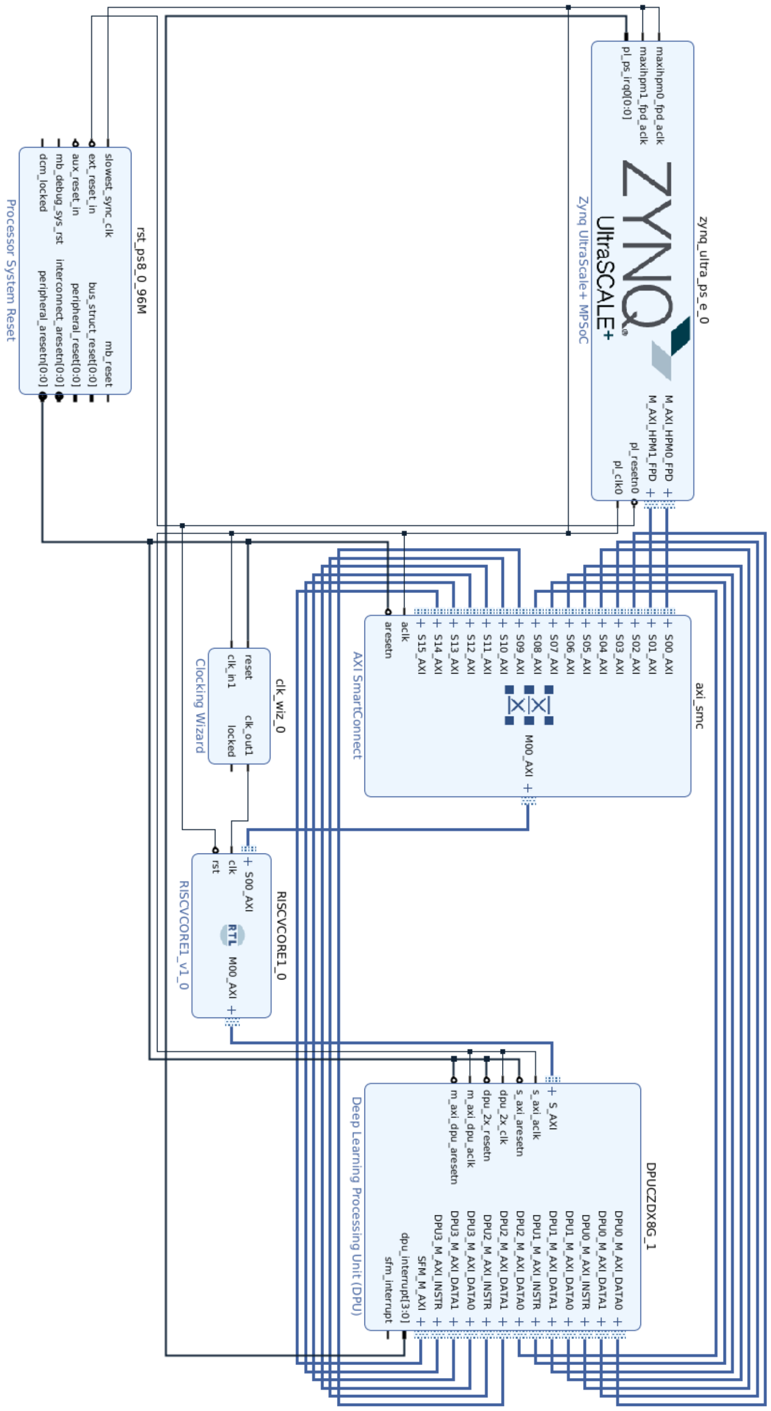
activation_7 (Activation)	(None, 19, 19, 64)	0	['add_2[0][0]']
conv2d_9 (Conv2D)	(None, 10, 10, 128)	73856	['activation_7[0][0]']
batch_normalization_8 (BatchNormalization	(None, 10, 10, 128)	512	['conv2d_9[0][0]']
activation_8 (Activation)	(None, 10, 10, 128)	0	['batch_normalization_8[0][0]']
conv2d_10 (Conv2D)	(None, 10, 10, 128)	147584	['activation_8[0][0]']
batch_normalization_9 (BatchNormalization	(None, 10, 10, 128)	512	['conv2d_10[0][0]']
conv2d_11 (Conv2D)	(None, 10, 10, 128)	8320	['activation_7[0][0]']
add_3 (Add)	(None, 10, 10, 128)	0	['batch_normalization_9[0][0]',
			'conv2d_11[0][0]']
activation_9 (Activation)	(None, 10, 10, 128)	0	['add_3[0][0]']
conv2d_12 (Conv2D)	(None, 10, 10, 128)	147584	['activation_9[0][0]']
batch_normalization_10 (BatchNormalization	(None, 10, 10, 128)	512	['conv2d_12[0][0]']
activation_10 (Activation)	(None, 10, 10, 128)	0	['batch_normalization_10[0][0]']
conv2d_13 (Conv2D)	(None, 10, 10, 128)	147584	['activation_10[0][0]']
batch_normalization_11 (BatchNormalization	(None, 10, 10, 128)	512	['conv2d_13[0][0]']
add_4 (Add)	(None, 10, 10, 128)	0	['batch_normalization_11[0][0]',
			'activation_9[0][0]']
activation_11 (Activation)	(None, 10, 10, 128)	0	['add_4[0][0]']
conv2d_14 (Conv2D)	(None, 10, 10, 128)	147584	['activation_11[0][0]']
batch_normalization_12 (BatchNormalization	(None, 10, 10, 128)	512	['conv2d_14[0][0]']
activation_12 (Activation)	(None, 10, 10, 128)	0	['batch_normalization_12[0][0]']
conv2d_15 (Conv2D)	(None, 10, 10, 128)	147584	['activation_12[0][0]']
batch_normalization_13 (BatchNormalization	(None, 10, 10, 128)	512	['conv2d_15[0][0]']
add_5 (Add)	(None, 10, 10, 128)	0	['batch_normalization_13[0][0]',
			'activation_11[0][0]']
activation_13 (Activation)	(None, 10, 10, 128)	0	['add_5[0][0]']
conv2d_16 (Conv2D)	(None, 10, 10, 128)	147584	['activation_13[0][0]']
batch_normalization_14 (BatchNormalization	(None, 10, 10, 128)	512	['conv2d_16[0][0]']
activation_14 (Activation)	(None, 10, 10, 128)	0	['batch_normalization_14[0][0]']
conv2d_17 (Conv2D)	(None, 10, 10, 128)	147584	['activation_14[0][0]']
batch_normalization_15 (BatchNormalization	(None, 10, 10, 128)	512	['conv2d_17[0][0]']
add_6 (Add)	(None, 10, 10, 128)	0	['batch_normalization_15[0][0]',
			'activation_13[0][0]']
activation_15 (Activation)	(None, 10, 10, 128)	0	['add_6[0][0]']
conv2d_18 (Conv2D)	(None, 5, 5, 256)	295168	['activation_15[0][0]']
batch_normalization_16 (BatchNormalization	(None, 5, 5, 256)	1024	['conv2d_18[0][0]']

activation_16 (Activation)	(None, 5, 5, 256)	0	['batch_normalization_16[0][0]']
conv2d_19 (Conv2D)	(None, 5, 5, 256)	590080	['activation_16[0][0]']
batch_normalization_17 (BatchNormalization	(None, 5, 5, 256)	1024	['conv2d_19[0][0]']
conv2d_20 (Conv2D)	(None, 5, 5, 256)	33024	['activation_15[0][0]']
add_7 (Add)	(None, 5, 5, 256)	0	['batch_normalization_17[0][0]',
			'conv2d_20[0][0]']
activation_17 (Activation)	(None, 5, 5, 256)	0	['add_7[0][0]']
conv2d_21 (Conv2D)	(None, 5, 5, 256)	590080	['activation_17[0][0]']
batch_normalization_18 (BatchNormalization	(None, 5, 5, 256)	1024	['conv2d_21[0][0]']
activation_18 (Activation)	(None, 5, 5, 256)	0	['batch_normalization_18[0][0]']
conv2d_22 (Conv2D)	(None, 5, 5, 256)	590080	['activation_18[0][0]']
batch_normalization_19 (BatchNormalization	(None, 5, 5, 256)	1024	['conv2d_22[0][0]']
add_8 (Add)	(None, 5, 5, 256)	0	['batch_normalization_19[0][0]',
			'activation_17[0][0]']
activation_19 (Activation)	(None, 5, 5, 256)	0	['add_8[0][0]']
conv2d_23 (Conv2D)	(None, 5, 5, 256)	590080	['activation_19[0][0]']
batch_normalization_20 (BatchNormalization	(None, 5, 5, 256)	1024	['conv2d_23[0][0]']
activation_20 (Activation)	(None, 5, 5, 256)	0	['batch_normalization_20[0][0]']
conv2d_24 (Conv2D)	(None, 5, 5, 256)	590080	['activation_20[0][0]']
batch_normalization_21 (BatchNormalization	(None, 5, 5, 256)	1024	['conv2d_24[0][0]']
add_9 (Add)	(None, 5, 5, 256)	0	['batch_normalization_21[0][0]', 'activation_19[0][0]']
activation_21 (Activation)	(None, 5, 5, 256)	0	['add_9[0][0]']
conv2d_25 (Conv2D)	(None, 5, 5, 256)	590080	['activation_21[0][0]']
batch_normalization_22 (BatchNormalization	(None, 5, 5, 256)	1024	['conv2d_25[0][0]']
activation_22 (Activation)	(None, 5, 5, 256)	0	['batch_normalization_22[0][0]']
conv2d_26 (Conv2D)	(None, 5, 5, 256)	590080	['activation_22[0][0]']
batch_normalization_23 (BatchNormalization	(None, 5, 5, 256)	1024	['conv2d_26[0][0]']
add_10 (Add)	(None, 5, 5, 256)	0	['batch_normalization_23[0][0]',
			'activation_21[0][0]']
activation_23 (Activation)	(None, 5, 5, 256)	0	['add_10[0][0]']
conv2d_27 (Conv2D)	(None, 5, 5, 256)	590080	['activation_23[0][0]']
batch_normalization_24 (BatchNormalization	(None, 5, 5, 256)	1024	['conv2d_27[0][0]']
activation_24 (Activation)	(None, 5, 5, 256)	0	['batch_normalization_24[0][0]']
conv2d_28 (Conv2D)	(None, 5, 5, 256)	590080	['activation_24[0][0]']
batch_normalization_25 (BatchNormalization	(None, 5, 5, 256)	1024	['conv2d_28[0][0]']

add_11 (Add)	(None, 5, 5, 256)	0	['batch_normalization_25[0][0]',
			'activation_23[0][0]']
activation_25 (Activation)	(None, 5, 5, 256)	0	['add_11[0][0]']
conv2d_29 (Conv2D)	(None, 5, 5, 256)	590080	['activation_25[0][0]']
batch_normalization_26 (BatchNormalization	(None, 5, 5, 256)	1024	['conv2d_29[0][0]']
activation_26 (Activation)	(None, 5, 5, 256)	0	['batch_normalization_26[0][0]']
conv2d_30 (Conv2D)	(None, 5, 5, 256)	590080	['activation_26[0][0]']
batch_normalization_27 (BatchNormalization	(None, 5, 5, 256)	1024	['conv2d_30[0][0]']
add_12 (Add)	(None, 5, 5, 256)	0	['batch_normalization_27[0][0]',
			'activation_25[0][0]']
activation_27 (Activation)	(None, 5, 5, 256)	0	['add_12[0][0]']
conv2d_31 (Conv2D)	(None, 3, 3, 512)	1180160	['activation_27[0][0]']
batch_normalization_28 (BatchNormalization	(None, 3, 3, 512)	2048	['conv2d_31[0][0]']
activation_28 (Activation)	(None, 3, 3, 512)	0	['batch_normalization_28[0][0]']
conv2d_32 (Conv2D)	(None, 3, 3, 512)	2359808	['activation_28[0][0]']
batch_normalization_29 (BatchNormalization	(None, 3, 3, 512)	2048	['conv2d_32[0][0]']
conv2d_33 (Conv2D)	(None, 3, 3, 512)	131584	['activation_27[0][0]']
add_13 (Add)	(None, 3, 3, 512)	0	['batch_normalization_29[0][0]', 'conv2d_33[0][0]']
activation_29 (Activation)	(None, 3, 3, 512)	0	['add_13[0][0]']
conv2d_34 (Conv2D)	(None, 3, 3, 512)	2359808	['activation_29[0][0]']
batch_normalization_30 (BatchNormalization	(None, 3, 3, 512)	2048	['conv2d_34[0][0]']
activation_30 (Activation)	(None, 3, 3, 512)	0	['batch_normalization_30[0][0]']
conv2d_35 (Conv2D)	(None, 3, 3, 512)	2359808	['activation_30[0][0]']
batch_normalization_31 (BatchNormalization	(None, 3, 3, 512)	2048	['conv2d_35[0][0]']
add_14 (Add)	(None, 3, 3, 512)	0	['batch_normalization_31[0][0]', 'activation_29[0][0]']
activation_31 (Activation)	(None, 3, 3, 512)	0	['add_14[0][0]']
conv2d_36 (Conv2D)	(None, 3, 3, 512)	2359808	['activation_31[0][0]']
batch_normalization_32 (BatchNormalization	(None, 3, 3, 512)	2048	['conv2d_36[0][0]']
activation_32 (Activation)	(None, 3, 3, 512)	0	['batch_normalization_32[0][0]']
conv2d_37 (Conv2D)	(None, 3, 3, 512)	2359808	['activation_32[0][0]']
batch_normalization_33 (BatchNormalization	(None, 3, 3, 512)	2048	['conv2d_37[0][0]']
add_15 (Add)	(None, 3, 3, 512)	0	['batch_normalization_33[0][0]', 'activation_31[0][0]']
activation_33 (Activation)	(None, 3, 3, 512)	0	['add_15[0][0]']
average_pooling2d (AveragePooling2D	(None, 2, 2, 512)	0	['activation_33[0][0]']
flatten (Flatten)	(None, 2048)	0	['average_pooling2d[0][0]']

dense (Dense)	(None, 512)	1049088	['flatten[0][0]']
dense_1 (Dense)	(None, 10)	5130	['dense[0][0]']

Appendix 3 – RISC-V with Xilinx Zynq FPGA Implementation Diagram


zynq_ultra_ps_e_0
maxihpm0_fpd_aclk
maxihpm1_fpd_aclk
pl_ps_irq0[0:0]
ZYNQ UltraSCALE+
Zynq UltraScale+ MPSoC
M_AXI_HPM0_FPD
M_AXI_HPM1_FPD
pl_resetn0
pl_clk0
rst_ps8_0_96M
slowest_sync_clk
ext_reset_in
aux_reset_in
mb_debug_sys_rst
dcm_locked
mb_reset
bus_struct_reset[0:0]
peripheral_reset[0:0]
interconnect_aresetn[0:0]
peripheral_aresetn[0:0]
Processor System Reset
axi_smc
S00_AXI
S01_AXI
S02_AXI
S03_AXI
S04_AXI
S05_AXI
S06_AXI
S07_AXI
S08_AXI
S09_AXI
S10_AXI
S11_AXI
S12_AXI
S13_AXI
S14_AXI
S15_AXI
aclk
aresetn
M00_AXI
AXI SmartConnect
clk_wiz_0
reset
clk_in1
clk_out1
locked
Clocking Wizard
RISCVCORE1_0
S00_AXI
clk
rst
RTL
M00_AXI
RISCVCORE1_v1_0
DPUCZDX8G_1
S_AXI
s_axi_aclk
s_axi_aresetn
dpu_2x_clk
dpu_2x_resetn
m_axi_dpu_aclk
m_axi_dpu_aresetn
DPU0_M_AXI_DATA0
DPU0_M_AXI_DATA1
DPU0_M_AXI_INSTR
DPU1_M_AXI_DATA0
DPU1_M_AXI_DATA1
DPU1_M_AXI_INSTR
DPU2_M_AXI_DATA0
DPU2_M_AXI_DATA1
DPU2_M_AXI_INSTR
DPU3_M_AXI_DATA0
DPU3_M_AXI_DATA1
DPU3_M_AXI_INSTR
SFM_M_AXI
dpu_interrupt[3:0]
sfm_interrupt
Deep Learning Processing Unit (DPU)

Appendix 4 – Thesis Work Code Screenshots

```
anil@ubuntu:~$ sudo apt install autoconf automake autotools-dev libmpc-dev libmpfr-dev libgmp-dev
hutils bc zlib1g-dev libexpat-dev libusb-1.0-0-dev gawk build-essential bison flex texinfo gperf libtool patc
anil@ubuntu:~$ sudo apt install device-tree-compiler pkg-config libncurses5-dev libncursesw5-dev
anil@ubuntu:~$ git clone --recursive https://github.com/riscv/riscv-tools.git
anil@ubuntu:~$ git submodule update -init -recursive
anil@ubuntu:~$ export RISCV=~/riscv
anil@ubuntu:~/Desktop/project$ cd riscv-gnu-toolchain/
anil@ubuntu:~/Desktop/project/riscv-gnu-toolchain$ ./configure --prefix=$RISCV
```

Figure 18 RISC-V ISA Installation Codes

```
anil@ubuntu:~$ which riscv64-unknown-elf-gcc
/opt/riscv/bin/riscv64-unknown-elf-gcc
anil@ubuntu:~$ riscv64-unknown-elf-gcc --version
riscv64-unknown-elf-gcc (g2ee5e430018) 12.2.0
Copyright (C) 2022 Free Software Foundation, Inc.
This is free software; see the source for copying conditions.  There is NO
warranty; not even for MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.
```

Figure 19 – RISC-V ELF GCC Toolchain

```
anil@ubuntu:~$ which riscv64-unknown-linux-gnu-gcc
/opt/riscv/bin/riscv64-unknown-linux-gnu-gcc
anil@ubuntu:~$ riscv64-unknown-linux-gnu-gcc --version
riscv64-unknown-linux-gnu-gcc (g2ee5e430018) 12.2.0
Copyright (C) 2022 Free Software Foundation, Inc.
This is free software; see the source for copying conditions.  There is NO
warranty; not even for MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.
```

Figure 20 – RISC-V Linux GCC Toolchain

```
anil@ubuntu:~/Desktop/project/riscv-gnu-toolchain$ cd ..
anil@ubuntu:~/Desktop/project$ git clone https://github.com/ucb-bar/chipyard.git
anil@ubuntu:~/Desktop/project/chipyard$ ./build-setup.sh riscv-tools
anil@ubuntu:~/Desktop/project/chipyard$ source ./env.sh
anil@ubuntu:~/Desktop/project/chipyard/sims/verilator$ ./simulator-chipyard-RocketConfig $RISCV/riscv64-unknown elf/share/riscv-tests/isa/rv64ui-p-simple
anil@ubuntu:~/Desktop/project/chipyard/sims/verilator$ make run-binary BINARY=$RISCV/riscv64-unknown-elf/share/riscv-tests/isa/rv64ui-p-simple
```

Figure 21 Chipyard Installation Codes

```
root@ubuntu:~/anil# ls
image_convolution.c
root@ubuntu:~/anil# riscv64-unknown-elf-gcc -fno-common -fno-builtin-printf
-specs=htif_nano.specs -c image_convolution.c
root@ubuntu:~/anil# ls
image_convolution.c  image_convolution.o
root@ubuntu:~/anil#
```

Figure 5 – Image Convolution with Chipyard RISC-V

```
root@ubuntu:~/anil# riscv64-unknown-elf-gcc -static -specs=htif_nano.specs image_convolution.o
 -o conv.riscv
root@ubuntu:~/anil# ls
conv.riscv  image_convolution.c  image_convolution.o
```

Figure 6 – Creating Executables with RISC-V

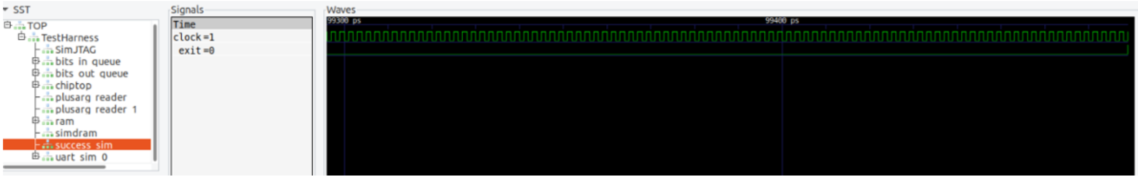


Figure 7 – GTKWave Simulation Output

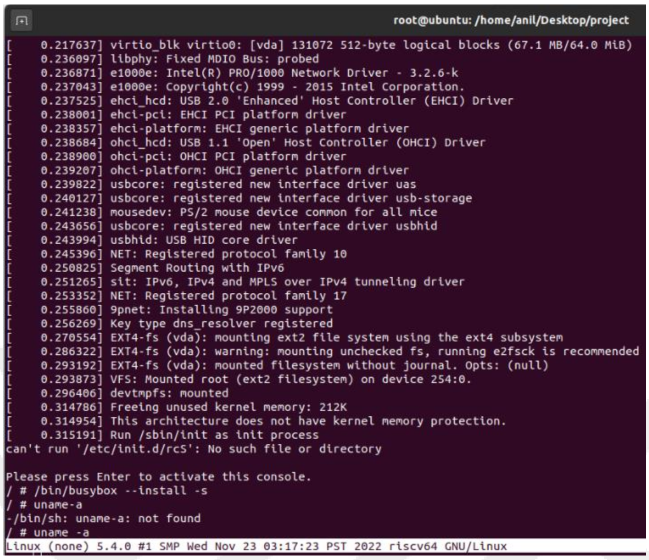


Figure 8– Linux OS on RISC-V

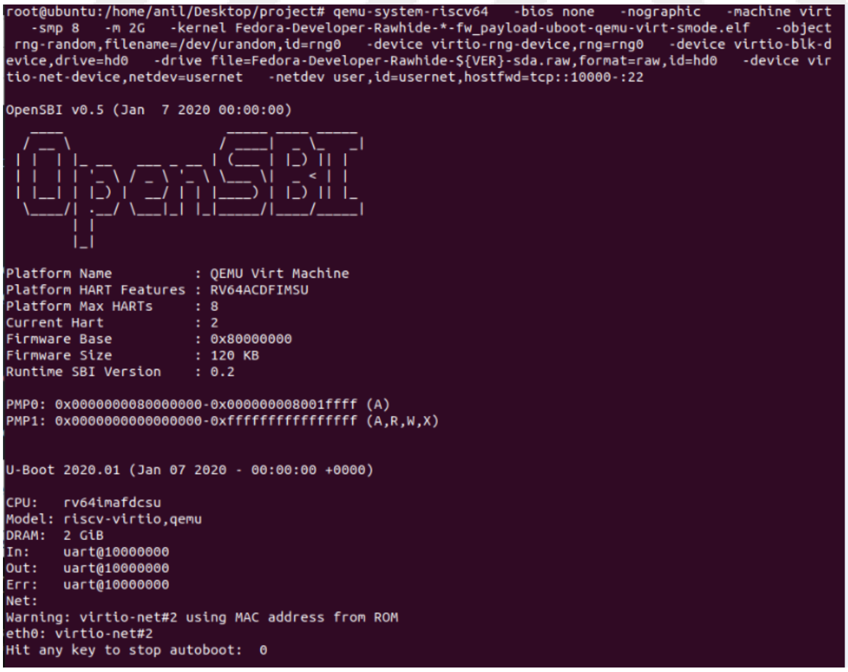


Figure 9 – Fedora OS Specifications on RISC-V

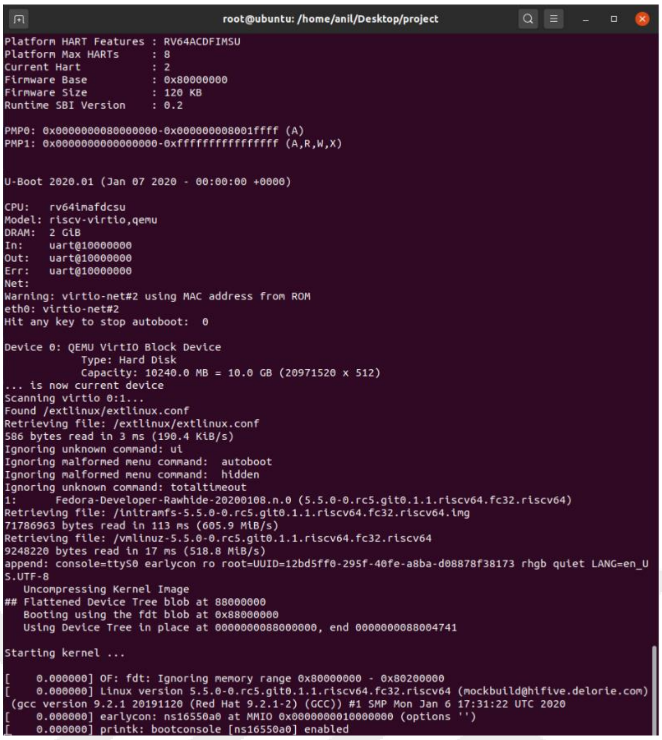


Figure 10 – Fedora OS Specifications on RISC-V

Figure 11– Installing Python Libraries on RISC-V

```
[  FAILED  ] Test_Caffe.read_gtsrb
[  FAILED  ] Test_Caffe.read_googlenet
[  FAILED  ] Test_Caffe.multiple_inputs
[  FAILED  ] Test_Caffe.shared_weights
[  FAILED  ] Reproducibility_FCN.Accuracy
[  FAILED  ] Reproducibility_SSD.Accuracy
[  FAILED  ] Reproducibility_AlexNet_fp16.Accuracy
[  FAILED  ] Reproducibility_GoogLeNet_fp16.Accuracy
[  FAILED  ] Test_Darknet.read_tiny_yolo_voc
[  FAILED  ] Test_Darknet.read_yolo_voc
[  FAILED  ] Test_Darknet.read_yolo_voc_stream
[  FAILED  ] Layer_LSTM_Test_Accuracy_with_.CaffeRecurrent
[  FAILED  ] Layer_LSTM_Test_Accuracy_with_.HiddenParams
[  FAILED  ] Layer_GRU_Test_Accuracy_with_.Pytorch
[  FAILED  ] Layer_RNN_Test_Accuracy_with_.CaffeRecurrent
[  FAILED  ] readNet.Regression
[  FAILED  ] readNet.do_not_call_setInput
[  FAILED  ] NMS.Accuracy
[  FAILED  ] BatchedNMS.Accuracy
[  FAILED  ] SoftNMS.Accuracy
[  FAILED  ] Test_TensorFlow.two_inputs
[  FAILED  ] Torch_Importer.simple_read
[  FAILED  ] DNNTestNetwork.MobileNet_SSD_Caffe/0, where GetParam() = OCV/CPU
[  FAILED  ] DNNTestNetwork.MobileNet_SSD_Caffe_Different_Width_Height/0, where GetParam() = OCV/CPU
[  FAILED  ] DNNTestNetwork.MobileNet_SSD_v1_TensorFlow/0, where GetParam() = OCV/CPU
[  FAILED  ] DNNTestNetwork.MobileNet_SSD_v1_TensorFlow_Different_Width_Height/0, where GetParam() = OCV/CPU
[  FAILED  ] DNNTestNetwork.MobileNet_SSD_v2_TensorFlow/0, where GetParam() = OCV/CPU
[  FAILED  ] DNNTestNetwork.SSD_VGG16/0, where GetParam() = OCV/CPU
[  FAILED  ] DNNTestNetwork.opencv_face_detector/0, where GetParam() = OCV/CPU
[  FAILED  ] DNNTestNetwork.Inception_v2_SSD_TensorFlow/0, where GetParam() = OCV/CPU
[  FAILED  ] DNNTestNetwork.FastNeuralStyle_eccv16/0, where GetParam() = OCV/CPU
[  FAILED  ] Test_Caffe_nets.Axpy/0, where GetParam() = OCV/CPU
[  FAILED  ] Test_Caffe_nets.Colorization/0, where GetParam() = OCV/CPU
[  FAILED  ] Test_Caffe_nets.FasterRCNN_vgg16/0, where GetParam() = OCV/CPU
[  FAILED  ] Test_Caffe_nets.FasterRCNN_zf/0, where GetParam() = OCV/CPU
[  FAILED  ] Test_Caffe_nets.RFCN/0, where GetParam() = OCV/CPU
[  FAILED  ] Reproducibility_AlexNet.Accuracy/0, where GetParam() = (false, CPU)
[  FAILED  ] Reproducibility_AlexNet.Accuracy/1, where GetParam() = (true, CPU)
[  FAILED  ] Reproducibility_ResNet50.Accuracy/0, where GetParam() = CPU
[  FAILED  ] Reproducibility_SqueezeNet_v1_1.Accuracy/0, where GetParam() = CPU
[  FAILED  ] Test_Caffe/opencv_face_detector.Accuracy/0, where GetParam() = ("dnn/opencv_face_detector.caffemodel", CPU)
[  FAILED  ] Test_Caffe/opencv_face_detector.Accuracy/1, where GetParam() = ("dnn/opencv_face_detector_fp16.caffemodel", CPU)
[  FAILED  ] Test_Caffe/opencv_face_detector.issue_15106/0, where GetParam() = ("dnn/opencv_face_detector.caffemodel", CPU)
[  FAILED  ] Test_Caffe/opencv_face_detector.issue_15106/1, where GetParam() = ("dnn/opencv_face_detector_fp16.caffemodel", CPU)
[  FAILED  ] Test_Darknet_nets.YoloVoc/0, where GetParam() = OCV/CPU
[  FAILED  ] Test_Darknet_nets.TinyYoloVoc/0, where GetParam() = OCV/CPU
[  FAILED  ] Test_Darknet_nets.YOLOv3/0, where GetParam() = OCV/CPU
[  FAILED  ] Test_Darknet_nets.YOLOv4/0, where GetParam() = OCV/CPU
[  FAILED  ] Test_Darknet_nets.YOLOv4_tiny/0, where GetParam() = OCV/CPU
[  FAILED  ] Test_Darknet_nets.YOLOv4x_mish/0, where GetParam() = OCV/CPU
[  FAILED  ] Test_Darknet_layers.shortcut/0, where GetParam() = OCV/CPU
[  FAILED  ] Test_Darknet_layers.shortcut_leaky/0, where GetParam() = OCV/CPU
[  FAILED  ] Test_Darknet_layers.shortcut_unequal/0, where GetParam() = OCV/CPU
```

Figure 12 – DNNTestNetwork Errors

```
--
--   OpenCV modules:
--     To be built:                 calib3d core dnn features2d flann gapi
highgui imgcodecs imgproc ml objdetect photo stitching ts video videoio
```

Figure 13 – Installed OpenCV Libraries

```
root@ubuntu: /home/anil/opencv/build
--       requested:                  DETECT
--
--   C/C++:
--     Built as dynamic libs?:       YES
--     C++ standard:                 11
--     C++ Compiler:                 /opt/RISCV/bin/riscv64-unknown-linux-gnu-g++  (ver 12
.2.0)
--     C++ flags (Release):          -march=rv64gc   -fsigned-char -W -Wall -Wreturn-type
-Wnon-virtual-dtor -Waddress -Wsequence-point -Wformat -Wformat-security -Wmissing-declar
ations -Wundef -Winit-self -Wpointer-arith -Wshadow -Wsign-promo -Wuninitialized -Wsugges
t-override -Wno-delete-non-virtual-dtor -Wno-comment -Wimplicit-fallthrough=3 -Wno-strict
-overflow -fdiagnostics-show-option -pthread -fomit-frame-pointer -ffunction-sections -fd
ata-sections  -fvisibility=hidden -fvisibility-inlines-hidden -O3 -DNDEBUG  -DNDEBUG
--     C++ flags (Debug):            -march=rv64gc   -fsigned-char -W -Wall -Wreturn-type
-Wnon-virtual-dtor -Waddress -Wsequence-point -Wformat -Wformat-security -Wmissing-declar
ations -Wundef -Winit-self -Wpointer-arith -Wshadow -Wsign-promo -Wuninitialized -Wsugges
t-override -Wno-delete-non-virtual-dtor -Wno-comment -Wimplicit-fallthrough=3 -Wno-strict
-overflow -fdiagnostics-show-option -pthread -fomit-frame-pointer -ffunction-sections -fd
ata-sections  -fvisibility=hidden -fvisibility-inlines-hidden -g  -O0 -DDEBUG -D_DEBUG
--     C Compiler:                   /opt/RISCV/bin/riscv64-unknown-linux-gnu-gcc
--     C flags (Release):            -march=rv64gc   -fsigned-char -W -Wall -Wreturn-type
-Waddress -Wsequence-point -Wformat -Wformat-security -Wmissing-declarations -Wmissing-pr
ototypes -Wstrict-prototypes -Wundef -Winit-self -Wpointer-arith -Wshadow -Wuninitialized
 -Wno-comment -Wimplicit-fallthrough=3 -Wno-strict-overflow -fdiagnostics-show-option -pt
hread -fomit-frame-pointer -ffunction-sections -fdata-sections  -fvisibility=hidden -O3 -
DNDEBUG  -DNDEBUG
```

Figure 14 – Installed OpenCV Modules

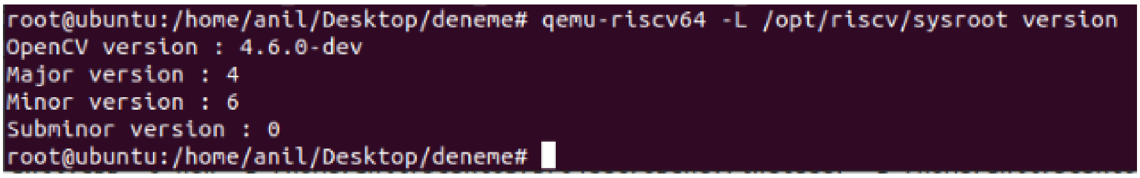


Figure 15 – Installed OpenCV Version

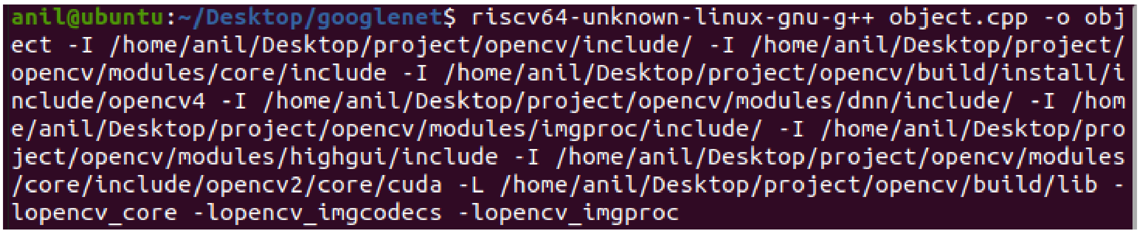


Figure 16 – RISC-V OpenCV Object Detection Code

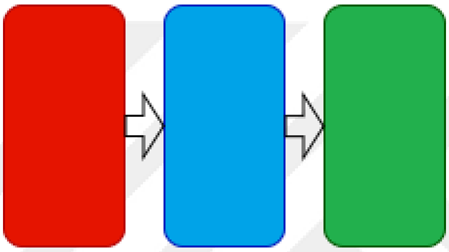


Figure 17 – Designed FPGA and RISC-V System Model Flow

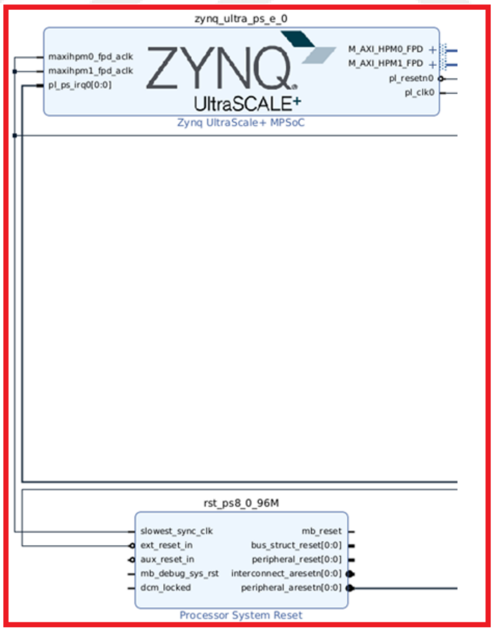


Figure 18 – Designed System Diagram in Vivado Part-1

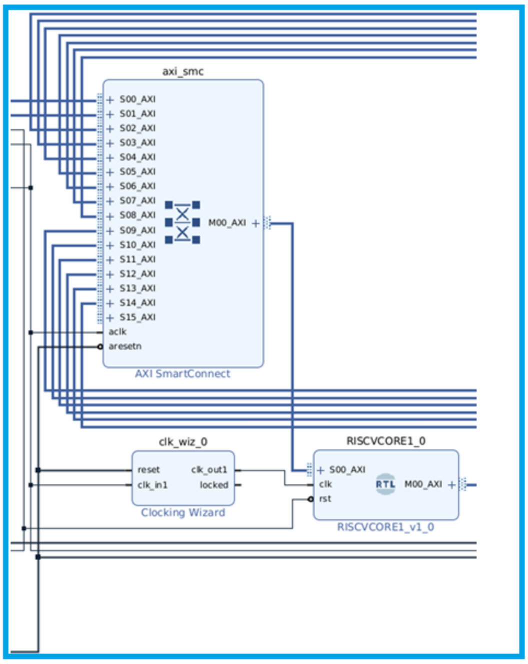


Figure 19 – Designed System Diagram in Vivado Part-2

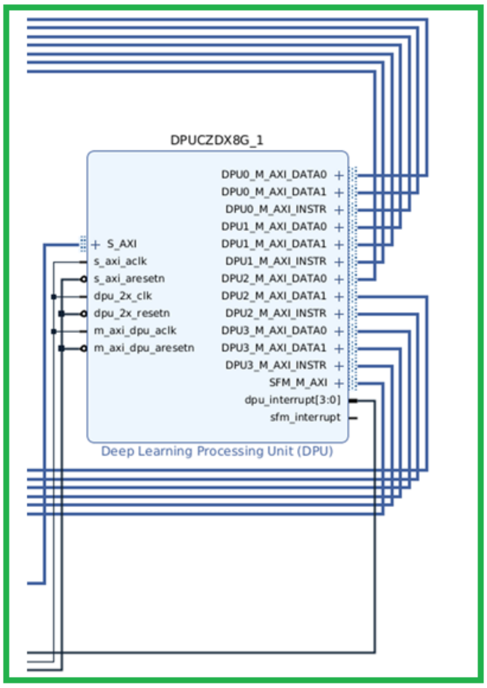


Figure 20 – Designed System Diagram in Vivado Part-3

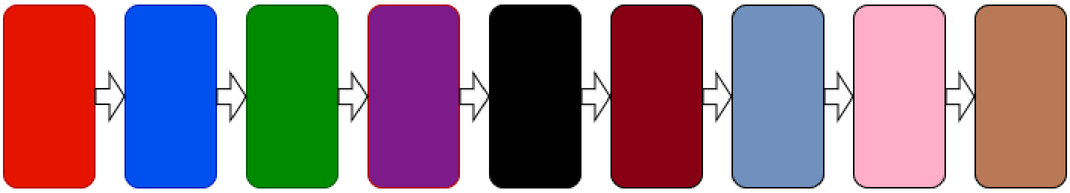


Figure 21 – Inspired Model Flow

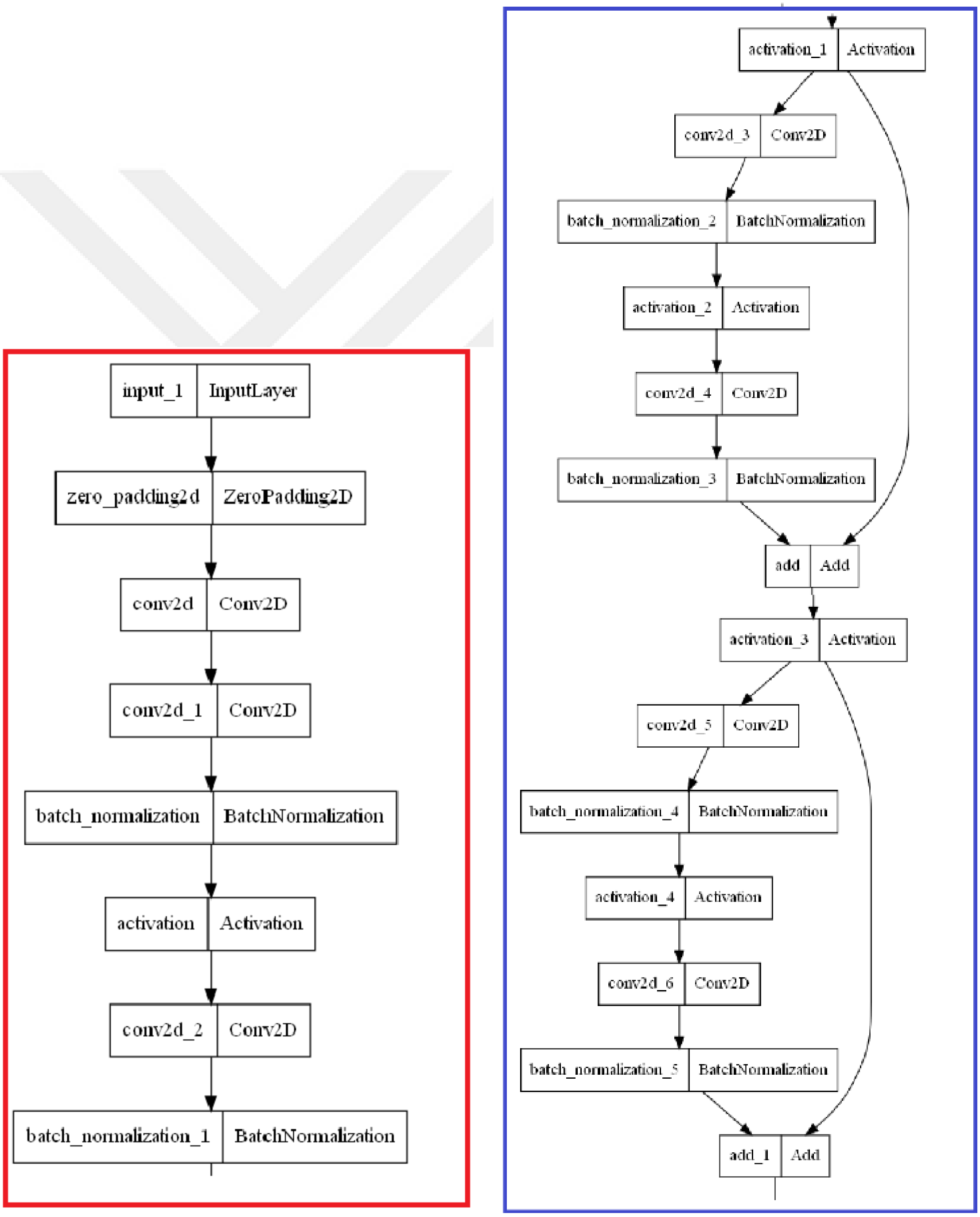


Figure 22 – Inspired Model Section 1-2

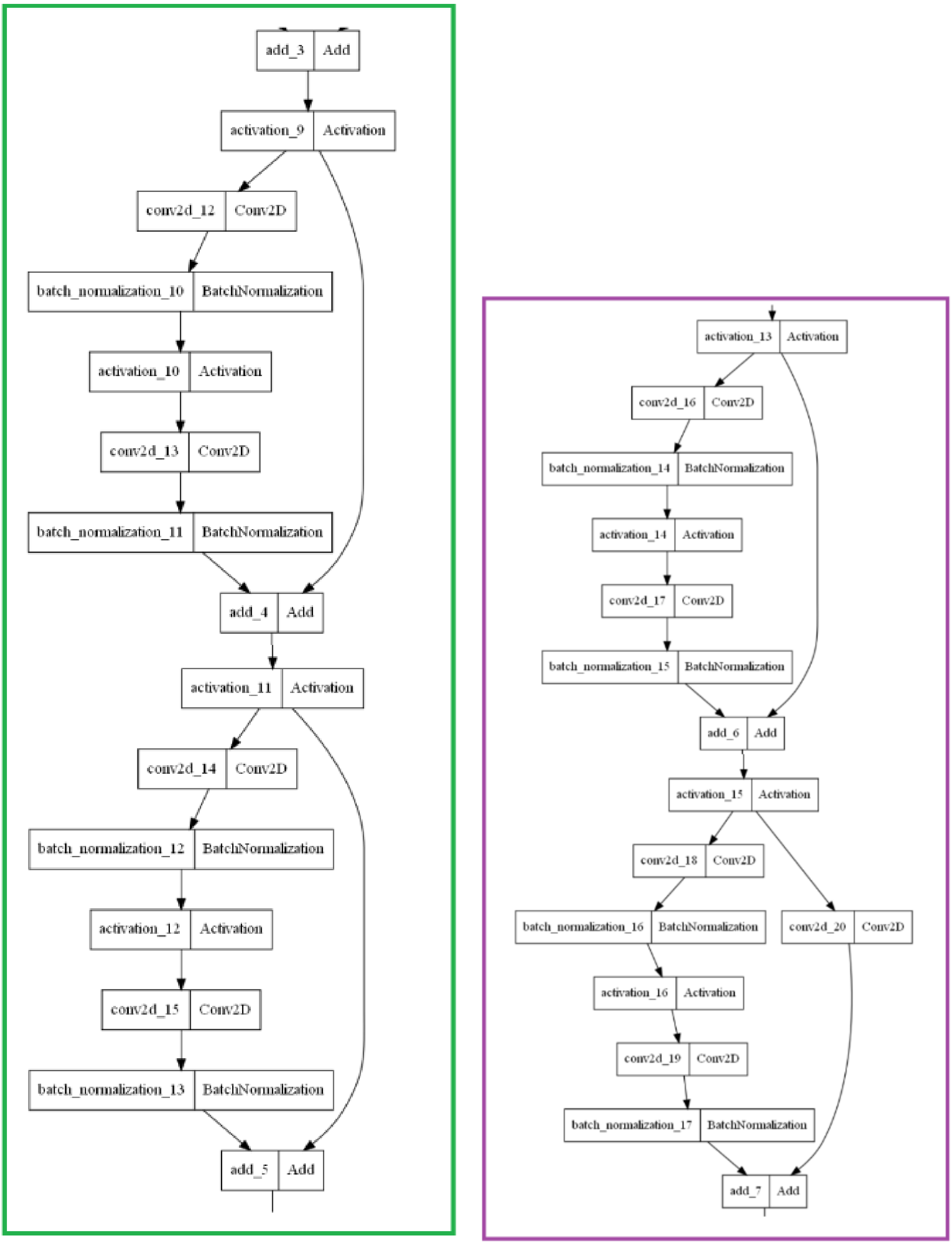


Figure 23 – Inspired Model Section 3-4

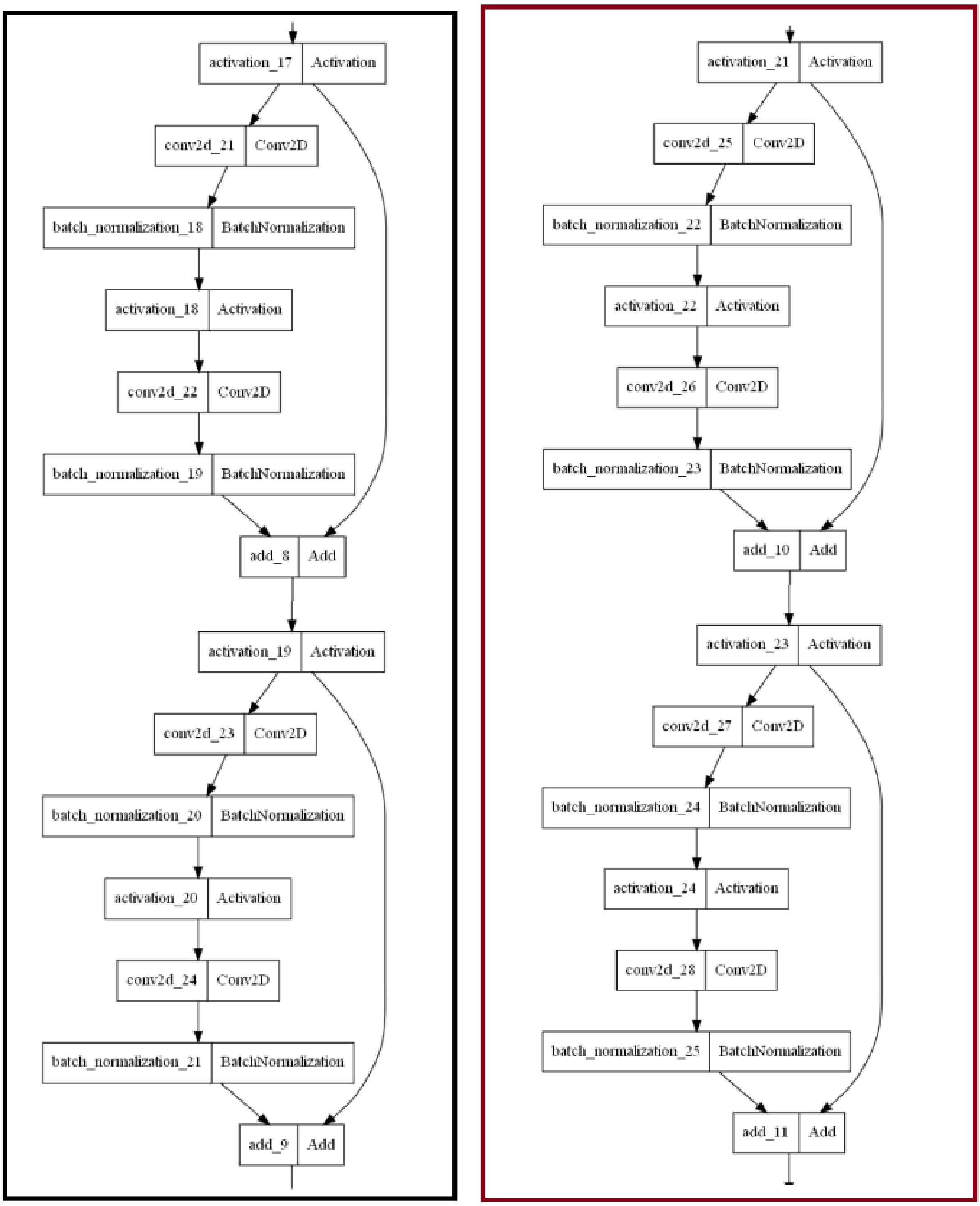


Figure 24 – Inspired Model Section 5-6

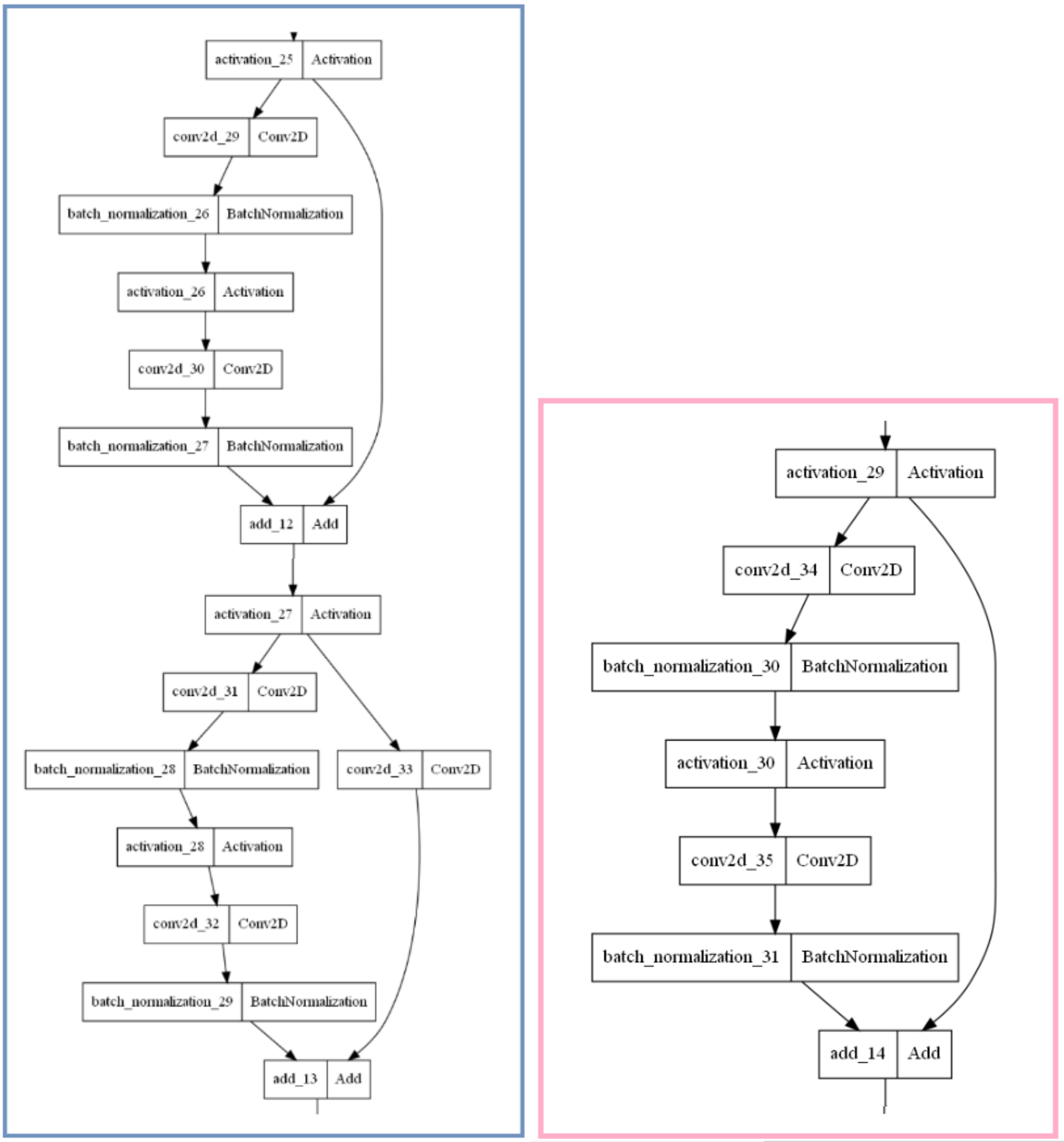


Figure 25 – Inspired Model Section 7-8

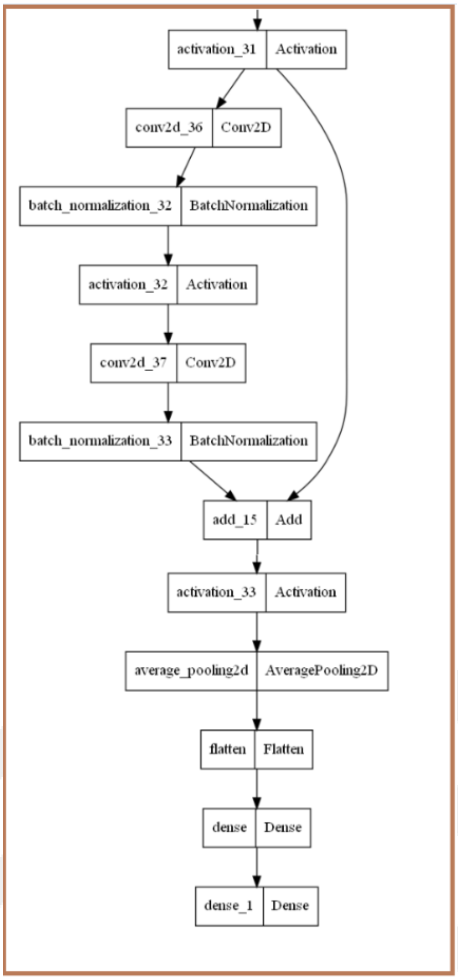


Figure 26 – Inspired Model Section 9