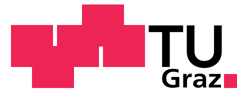

Mario Freisinger, BSc

Hardware/Software Co-Design for Edge AI Acceleration on RISC-V-based Systems

MASTER'S THESIS

to achieve the university degree of

Diplom-Ingenieur

Master's degree programme: Information and Computer Engineering

submitted to

Graz University of Technology

Supervisor
Ass.Prof. Dipl.-Ing. Dr.techn. Christian Steger

Advisor
Dipl.-Ing. Christian Seifert, BSc

Institute of Technical Informatics

Graz, March 2026

Affidavit

I declare that I have authored this thesis independently, that I have not used other than the declared sources/resources, and that I have explicitly indicated all material which has been quoted either literally or by content from the sources used. The text document uploaded to TUGRAZonline is identical to the present master‘s thesis.

Date

Signature

Abstract

The deployment of neural networks for inference on edge devices poses significant challenges, particularly in terms of memory capacity and power consumption. Hardware and software co-design methodologies provide a systematic framework for developing hardware architectures that address these constraints while ensuring adequate performance for efficient execution of the forward pass. To establish a comprehensive understanding of current methodologies, several existing embedded systems and **R**educed **I**nstruction **S**et **C**omputer – V (RISC-V)-based platforms for edge artificial intelligence inference are reviewed.

Based on these insights, a hardware accelerator and a custom instruction are developed to facilitate the classification of the number of fingers present in an image. As a representative use case, a camera connected to the **F**ield-**P**rogrammable **G**ate **A**rray (FPGA) is integrated with a classification task executed on the RISC-V core cv32e40x to validate the proposed hardware accelerator and instruction-set extension. The implemented system incorporates an OV7670 camera and performs image classification directly on the FPGA, thereby demonstrating the feasibility of on-device inference.

The cv32e40x soft core deployed on the Nexys 4 DDR FPGA serves as the evaluation platform for assessing the effectiveness of the modified microarchitecture. Hardware and software co-design techniques are systematically discussed and applied to derive an appropriate solution. To facilitate the rapid assessment of alternative hardware accelerator and custom instruction designs, the entire system is co-simulated utilizing a RISC-V SystemC core.

Kurzfassung

Für die Implementierung eines neuronalen Netzes auf einem Edge-Gerät ergeben sich verschiedene Einschränkungen, wie etwa begrenzte Speicher- und Energieressourcen. Hardware/Software-Co-Design-Methoden können eingesetzt werden, um unterschiedliche Hardwarearchitekturen zu entwickeln, die diese Einschränkungen erfüllen und gleichzeitig eine ausreichend schnelle Berechnung des Forward-Pfads ermöglichen.

Um aktuell verwendete Techniken zur Erreichung dieses Ziels zu verstehen, werden mehrere bestehende Embedded-Systeme sowie RISC-V-basierte Systeme für Edge-AI-Inferenz untersucht. Auf Grundlage dieser Erkenntnisse werden ein Hardwarebeschleuniger und eine benutzerdefinierte Instruktion zur Klassifikation der Fingeranzahl in einem Bild entworfen. Als grundlegender Anwendungsfall wird ein Klassifikationsszenario mit einer an das FPGA angeschlossenen Kamera verwendet, das auf einem RISC-V-core namens cv32e40x ausgeführt wird, um den entworfenen Hardwarebeschleuniger zu validieren. Das entwickelte System verwendet eine OV7670-Kamera und ist in der Lage, die von ihr aufgenommenen Bilder, direkt auf dem FPGA zu klassifizieren.

Der auf dem FPGA Nexys 4 DDR implementierte cv32e40x-Softcore wird genutzt, um die Zweckmäßigkeit der veränderten Mikroarchitektur zu evaluieren. Zur Lösungsfindung werden Hardware/Software-Co-Design-Methoden diskutiert und angewendet. Das gesamte System wird mithilfe eines RISC-V-SystemC-Cores co-simuliert, um unterschiedliche Hardwarebeschleuniger- oder Custom-Instruction-Designs auf schnellere Weise testen zu können.

Contents

1 Introduction **1**

2 State of the Art **4**

2.1 Neural Network Optimization Techniques for Embedded Systems . . 5

2.1.1 Quantization of Neural Networks 6

2.1.2 Accuracy of Quantized Neural Networks 8

2.1.3 Basics of Quantization 9

2.1.4 Energy Consumption, Memory Requirement, and Execution Speed of Quantized vs. Non-Quantized Neural Networks . . . 10

2.1.5 Post-Training Quantization and Quantization Aware Training 13

2.1.6 Comparison of **Q**uantization-**A**ware **T**raining (QAT) and **P**ost-**T**raining **Q**uantization (PTQ) 14

2.2 HW/SW Co-design for Edge AI Acceleration 15

2.3 Edge AI Acceleration on Embedded Systems 16

2.3.1 Acceleration via Systolic Matrix Multiplication 17

2.3.2 Edge Tensor Processing Unit (TPU) 21

2.3.3 Smaller Hardware Accelerators for Different Layer Structures . 21

2.4 HW/SW Co-design for Edge AI Acceleration on RISC-V-based Systems 24

2.4.1 Acceleration of RISC-V-based Edge AI Systems 26

2.4.2 Edge AI Acceleration via Custom Instructions for RISC-V-based Systems 29

3 Design **34**

3.1 Selecting a Suitable Deep Learning Compiler Stack 34

3.1.1 Chosen Deep Learning Compiler Stack 35

3.2 Selecting the Neural Network Layer Structure for Finger Count Classification 36

3.3 Profiling and Runtime Analysis 37

3.3.1 Basic Hotspot Identification 39

4 Implementation **42**

4.1 Structure of the Neural Network Used for Finger Count Classification 43

4.2 Training of the Neural Network and Quantization 44

4.3 Extension of **T**ensorflow **L**ite for **M**icrocontrollers (TFLM) to Operate on cv32e40x Softcore 45
4.4 Integration of the Core on the System-on-Chip and Simulation 47
4.4.1 Basic Memory Layout 48
4.4.2 Co-simulation for Enhanced Verification of the Hardware Accelerator 49
4.5 Hardware Acceleration of the Forward Pass 53
4.5.1 Implementation of the Custom Instruction 56
4.6 Validation of the Synthesized Custom **S**ystem-on-**C**hip (SOC) with the cv32e40x as Core 59

5 Results and Evaluation 63
5.1 Results Obtained Through Co-Simulation 63
5.2 Post-Synthesis Results 65
5.3 Analysis of Area Utilization of the Different Architectures 66

6 Conclusion 68
6.1 Future Work 69

Acronyms 71

Bibliography 73

List of Figures

1.1 Image classification system using the OV7670 camera, SCCB configuration,cv32e40x with custom instruction and the hardware accelerator with **U**niversal **A**synchronous **R**eceiver/**T**ransmitter (UART) output bridged via **U**niversal **S**erial **B**us (USB). 3

2.1 VRAM and memory comparison for parameters development of large language models. [51, p. 1] 6
2.2 Memory requirements and classification error of neural network models with varying complexity from ImageNet dataset classification [49, p. 2] 7
2.3 Energy use of different tasks on a 45nm 0.9V system from 2014 [21, p. 12]. 10
2.4 Energy consumption with different quantization levels and different bit accuracy models [34, p. 5] 11
2.5 Comparing quantized and non-quantized neural networks in terms of time, storage, memory, and top-error rate. [50, p. 1] 12
2.6 Important limits on edge devices for running neural networks. [47, p. 1] 15
2.7 How well different methods improve neural network performance on small devices. [47, p. 4] 16
2.8 Different designs to speed up neural networks on small devices. [17, p. 5] 17
2.9 Explanation of a helpful hardware design to speed up the convolutional layer. [16, p. 31] 18
2.10 Schematic overview of systolic array matrix multiplication with stationary weights used to accelerate convolutions. [1, p. 3] 20
2.11 Overview of different layer structures grouped by families of layer structures. [6, p. 3] 22
2.12 Acceleration for layer types of family 1 and family 2 which are mostly **C**onvolutional **N**eural **N**etwork (CNN) layers. [6, p. 8] 23
2.13 Acceleration for a layers in family 3, which are mostly LSTM or fully connected layers. [6, p. 8] 24
2.14 Overview of a design space exploration in the context of edge AI systems [8, p. 1] 26
2.15 High-level overview of embedded Systems for neural network inference. [17, p. 5] 27

2.16 Diagram of a system called TinyVers, which uses a RISC-V-based SoC. [17, p. 6] . . . 28
2.17 Overview of the system used to manage multiple **M**atrix **V**ector **U**nit (MVU)s and the diagram of MVU. [17, p. 11] . . . 29
2.18 Zero-riscy core with an extended pipeline to enhance CNN performance. [56, p. 3] . . . 31
2.19 A block diagram of the dot product unit used to improve the zero-riscy core. [56, p. 3] . . . 31
2.20 Custom instruction doing a multiply-shift-accumulate operation as one instruction by extending the RISC-V vector instruction subset. [14, p. 3] 32
2.21 Comparison of Conv2d using a 7x7 kernel, where LP-conv2d and ULP-conv2d use a special instruction called vmacsr. [14, p. 4] . . . 33

3.1 Diagram showing the neural network's design. Made with Netron [36]. 37
3.2 Histogram of program counter values with corresponding line-numbers of TFLM files . . . 40

4.1 Design process for creating custom instructions and hardware accelerators to make finger count classification faster . . . 43
4.2 Overview of tools used to train, adjust, and compile the model for a 32-bit RISC-V core. . . . 44
4.3 Overview of tool-flow for simulation and co-simulation of the whole finger count classification System . . . 47
4.4 Overall tool flow used for adding, simulating, and checking the hardware accelerator and custom instruction for each abstraction layer. The cycle-accurate simulation time was measured on an i9-9900K. . . . 51
4.5 Convolution operation with input offset applied before element-wise multiplication. . . . 53
4.6 Hardware accelerator for 3×3 convolution. Each PE performs one multiplication using memory-mapped registers for the input and kernel values. The results are added in an accumulator. . . . 55
4.7 The add_mul_acc instruction can add an input offset and then multiply the result by a filter value. It uses an accumulation register to store results until they are reset to zero with the add_mul_acc_reset instruction. 58
4.8 CV32E40X pipeline with a custom add_mul_acc instruction integrated via the eXtension Interface (XIF). . . . 58
4.9 Image classification system using the OV7670 camera, SCCB configuration,cv32e40x with custom instruction and the hardware accelerator with UART output bridged via USB. . . . 60
4.10 Picture of the finger count classification system. . . . 61

4.11 Picture of hand taken by the camera and transmitted via UART. This image is used for classification on the FPGA board. 61
4.12 Pmod connections called JA and JB with OV7670 camera signals. . . 62

5.1 Data read and write counts and instructions executed for default ISA, hardware accelerator, and custom instruction are obtained by Co-Simulation. 64
5.2 Measured times for the forward pass with the default architecture, hardware accelerator on, and custom instruction on. 66
5.3 Chip size comparison obtained by Yosys for a 130nm process (SKY130) for the differnt architectures designed. 67

List of Tables

2.1 Edge AI Device Comparison 5
2.2 Overview of various edge AI accelerators, including their power consumption and costs. [31, p. 1] 5
2.3 16-bit or 8-bit floating point and fixed point resource requirements for multiplication[49, p. 3] 7
2.4 Different bit widths for quantizing CONVNET and their test accuracy. [45, p. 5] 8
2.5 Comparison of different quantization methods like PTQ and QAT. [41, p. 4]. 14

4.1 Memory Map of the FPGA Peripherals and Custom Instruction Registers 49

5.1 Performance Comparison of Default ISA, Hardware Accelerator, and Custom Instruction via co-simulation 65

1 Introduction

The application of supervised machine learning on edge devices, which may be constrained by power and memory limitations, presents a significant challenge. The development of an edge device capable of rapidly classifying data while minimizing the energy consumption per classification requires the resolution of numerous requirements. A vast design space exists from which solutions must be selected, utilizing a wide array of existing tools to implement neural networks on edge devices. Various architectures, such as Google's edge **T**ensor **P**rocessing **U**nit (TPU) [6], or systems that address certain bottlenecks of the edge TPU and propose novel architectural designs [42] for edge-based machine learning, may prove beneficial. These aspects are elaborated upon in the state-of-the-art chapter and are used to design an RISC-V-based system featuring a specialized hardware accelerator and a custom instruction for the efficient execution of the forward pass on the edge. RISC-V-based systems are considered due to their open architecture, which can be modified and utilized for production without incurring additional licensing fees [2]. Several proposals suggest specific accelerator architectures [52] [53] and custom instructions [32] tailored for RISC-V-based architectures, which are examined in this thesis.

Fundamental architectures for accelerating the forward pass on edge devices and specific designs for RISC-V-based systems are described to propose a novel approach for accelerating the forward pass. Use cases such as gesture recognition, event-based sensing in drones, and autonomous driving assistance can benefit from an efficient edge machine learning system. These use cases, along with current architectural methods to enhance system efficiency for their respective applications, are discussed in the state-of-the-art section.

The primary objective is to implement an image classification system with a camera connected to an FPGA board to ascertain the number of visible fingers in a captured image. This served as an initial use case to deploy a neural network on the RISC-V-based system, necessitating several custom modifications to existing neural network creation and training tools, such as TFLM [46]. The layers of the neural network were selected, and training was conducted using PyTorch [37]. The model was then converted to operate without an operating system on the cv32e40x with a custom SOC, utilizing the TFLM. framework to facilitate this process.

The fundamental implementation of a neural network for classifying the finger count on a custom SOC presented a significant challenge. The methodology employed to

achieve this is detailed in the Implementation chapter.

The ultimate objective is to identify a hardware accelerator and custom instruction to enhance the speed and energy efficiency of the forward pass. A co-simulation of a RISC-V core via a **T**ransaction-**L**evel **M**odel (TLM) written in SystemC [33] was utilized to expedite the verification results. This approach allows for the acquisition of metrics on the impact of the designed accelerator on the forward pass without the necessity of synthesizing and loading the bitstream onto the FPGA. The cycle-accurate simulation of the entire forward pass with the default **I**nstruction **S**et **A**rchitecture (ISA) executes approximately 137 million instructions and requires several minutes to complete. Consequently, the co-simulation proves beneficial in the design of the hardware accelerator and custom instruction, as do the papers discussed in the state-of-the-art chapter.

The primary goal of this thesis is to address the significant challenges of implementing neural networks for inference on edge devices, specifically focusing on memory capacity and power consumption constraints. This thesis aims to develop and validate a hardware/software co-design methodology that creates efficient hardware architectures capable of enabling neural network inference on resource-limited edge devices.
The specific objectives of this thesis are as follows:

- The practical implementation of feasible on-device inference is demonstrated on a RISC-V-based platform utilizing the cv32e40x as core.
- Utilizing co-simulation techniques for rapid evaluation of design alternatives.
- Creating a custom hardware accelerator for neural network inference tasks.
- To develop and integrate a custom instruction set extension aimed at accelerating the image classification process.

Ultimately, all custom hardware and the RISC-V-based system with the cv32e40x as the core were synthesized and validated on the FPGA board known as Nexys 4 DDR. The employed SOC is augmented with a frame buffer and memory-mapped registers for the hardware accelerator, and the core is further enhanced with a custom instruction. This system was tested using real images captured by the camera, ov7670. The classification times required with the default ISA, the hardware accelerator, and the custom instruction are compared. This comparison was conducted using co-simulation results and clock-cycle measurements of the actual synthesized system, which classifies images captured by the camera and reports the clock-cycle count alongside the classification result via UART to the terminal.

The synthesized system with the cv32e40x, extended with a custom instruction and hardware accelerator, is depicted in Figure 1.1. The camera communicates with the core via the frame buffer and modules for **S**erial **C**amera **C**ontrol **M**odule (SCCM) communication, utilizing an 8-bit interface to transfer the recorded images, as illustrated in the block diagram.

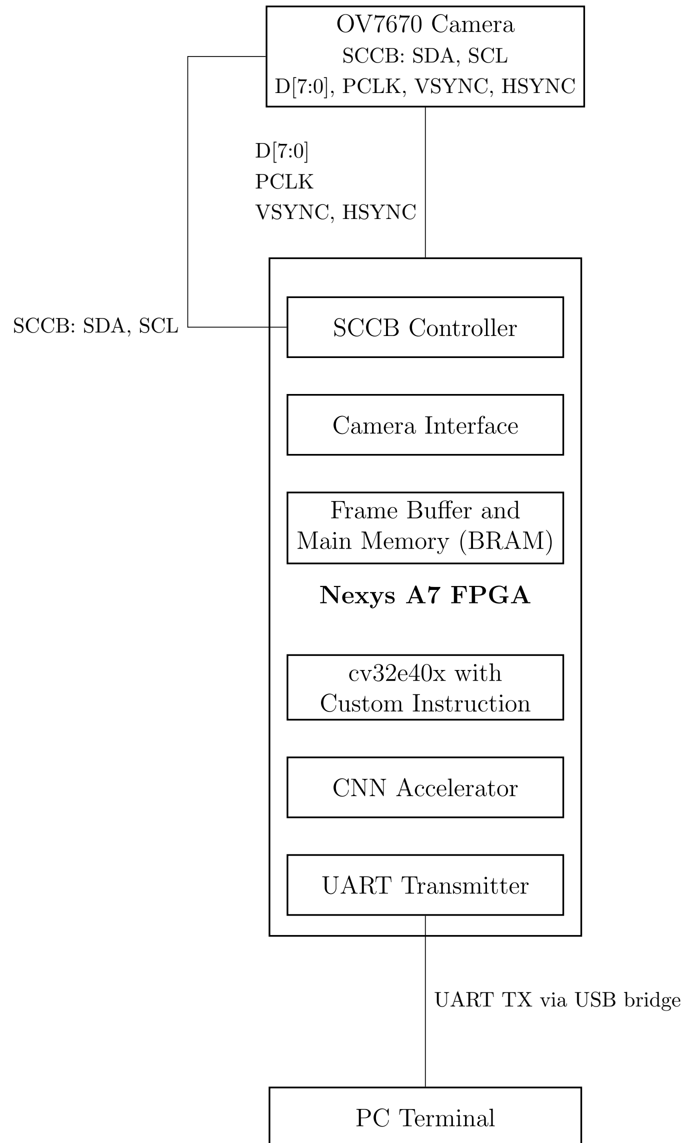

Figure 1.1: Image classification system using the OV7670 camera, SCCB configuration,cv32e40x with custom instruction and the hardware accelerator with UART output bridged via USB.

The forthcoming chapter seeks to deliver a thorough overview and insights into current solutions for this task by evaluating cutting-edge projects. It focuses on the key results and architectural frameworks of these projects.

2 State of the Art

The challenge of efficiently executing classification tasks on energy-constrained edge devices is significant, as merely increasing parallel computing power may not suffice. Edge systems often face severe memory limitations, and even the bandwidth for memory access may be restricted, complicating the design of neural network architectures and custom hardware solutions. These constraints necessitate innovative architectural solutions to achieve efficient classification while maintaining high accuracy across various applications.

The applications of artificial intelligence on edge devices include gesture recognition and event-based sensing in drones and low-power devices. A notable advantage of local classification execution on edge devices is the elimination of data transmission to external servers, thereby avoiding reliance on centralized classification services. Additionally, reducing the inference latency on local devices is a critical consideration [35, p. 2].

Another use case involves the use of autonomous systems in vehicles for driver assistance. Edge AI systems can significantly reduce decision-making latency from 95ms-180ms to approximately 8.5ms, thereby expediting critical decision-making processes. In complex traffic scenarios, where potential hazards may arise, such a latency reduction is crucial [35, p. 6].

A hardware/software co-design approach to accelerate the forward pass of neural networks on edge devices may effectively address the stringent power and memory constraints of these systems. Table 2.1 provides an overview of existing edge AI accelerators, detailing their power usage, memory size and cost. Google's Edge TPU is also examined in greater detail, as the architectures of these existing systems can inform the design of an accelerator for the targeted RISC-V-based system.

The initial step in executing the forward pass on an edge device involves optimizing the neural network for a specific system. Given that the system may only support integer calculations, it is necessary to quantize the neural network and make specific adjustments to accommodate the limited available memory. The subsequent state-of-the-art section delineates the available strategies and their operational mechanisms. This approach facilitates the selection of an appropriate optimization technique for a neural network that must operate on a memory-constrained RISC-V-based system, which exclusively supports integer operations.

Table 2.1: Edge AI Device Comparison

Device	Power (W)	Memory	Cost	Accelerated Workloads
Intel NCS2 (VPU)	1–2	512 MB	$99	Vision, imaging
Google EdgeTPU	0.5–2	8 MB	$75	TensorFlow Lite models
Nvidia Nano	5–10	4 GB	$99	GPU workloads; AI
Nvidia TX2	7.5	8 GB	$399	GPU workloads; AI

Table 2.2: Overview of various edge AI accelerators, including their power consumption and costs. [31, p. 1]

2.1 Neural Network Optimization Techniques for Embedded Systems

For neural networks deployed on edge devices, it is imperative to optimize them for minimal memory and power consumption. This can be achieved through the use of quantized neural networks, wherein the weights are represented as integers or even binary numbers. When binary numbers are employed, a **Q**uantized **N**eural **N**etwork (QNN) is referred to as a binary neural network. The utilization of quantized neural networks obviates the need for a floating-point unit in the evaluation process. The absence of a floating-point unit is advantageous in terms of the general resource requirements for an FPGA implementation of QNNs. Although the results may be less precise, the benefits conferred to edge devices often justify the trade-off. [44, p. 2]

To expedite the forward pass of a neural network on an RISC-V core, various hardware-software co-design methods can be employed to enhance the prediction capabilities of the network. Currently, HW/SW co-design methods are utilized to accelerate neural networks while simultaneously reducing power consumption. This is particularly crucial for edge devices, where resources such as power and memory are severely limited. Edge devices offer advantages such as enhanced privacy and reduced bandwidth requirements for neural network operations. [7, p. 2] These benefits are particularly significant in scenarios where network connectivity is unavailable, yet the classifier system must remain operational.

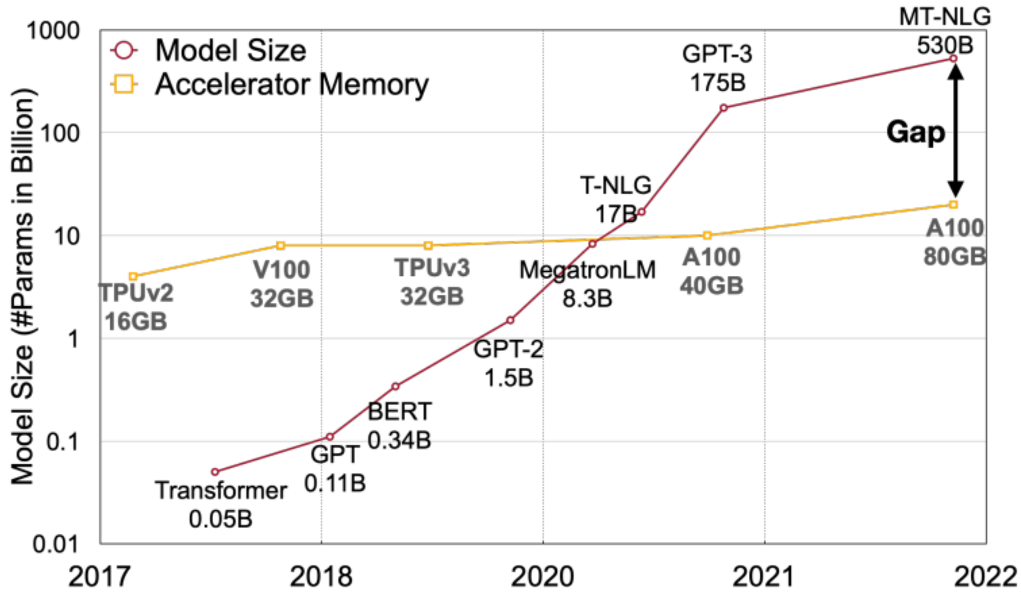

Figure 2.1: VRAM and memory comparison for parameters development of large language models. [51, p. 1]

The current utilization of language models and their parameter development surpasses the advancement of GPU memory, as illustrated in Figure 2.1. Figure 2.1 demonstrates that the GPT-3 model comprises 175 billion parameters [9, p. 1]. With 16-bit floating-point parameters, a minimum of 350GB of memory is necessary. For inference, five A100 GPUs are required to execute the model's forward pass, with each A100 GPU possessing 80GB of GPU memory. This setup also introduces additional communication overhead between each dedicated GPU. Quantization can reduce the required memory by a factor of two and double the throughput of the model. This poses no issue for convolutional neural networks or small transformer networks. However, challenges arise when attempting to quantize large language models with parameter counts in the billions. In such cases, outliers in the activation functions lead to significant quantization errors and accuracy degradation [51, p. 1].

2.1.1 Quantization of Neural Networks

The storage requirements of a neural network can rapidly become substantial, potentially leading to issues when targeting an FPGA. Figure 2.2 illustrates various convolutional neural network layouts and their memory requirements. The layout of the selected CNN significantly influences the memory required. There are two primary strategies to reduce the memory requirements for a CNN: either by reducing the bit

size of the floating-point numbers representing the weights and biases or by employing fixed-point quantization with bit sizes of, for example, 16, 4, or 8 bits [49, p. 2].

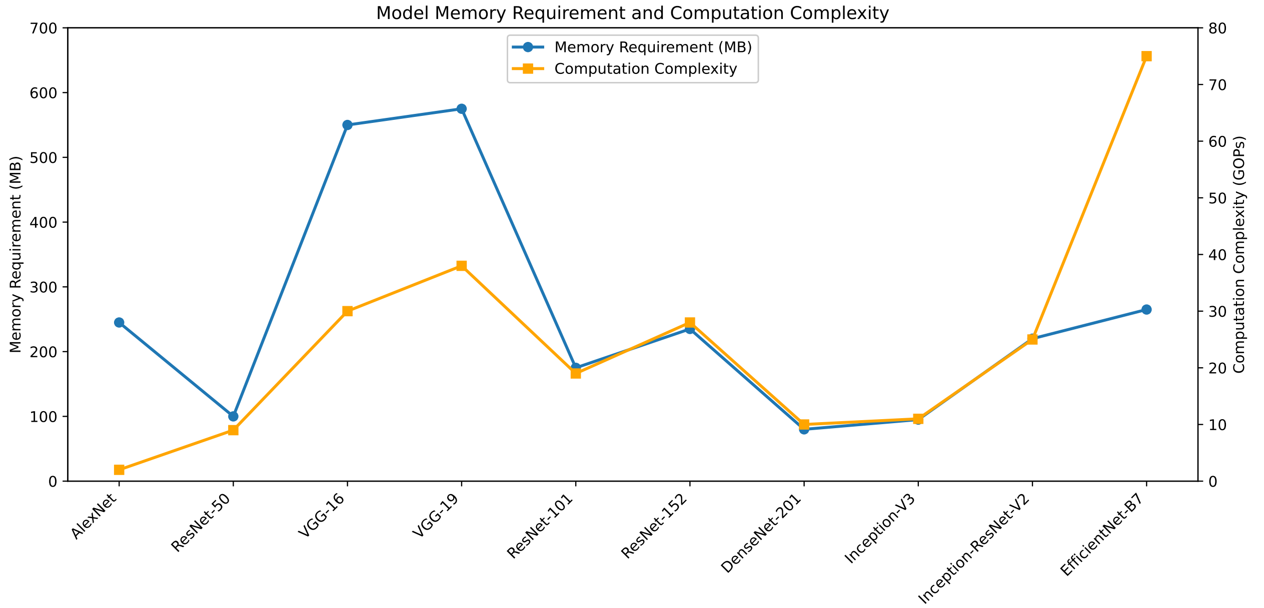

Figure 2.2: Memory requirements and classification error of neural network models with varying complexity from ImageNet dataset classification [49, p. 2]

Table 2.3: 16-bit or 8-bit floating point and fixed point resource requirements for multiplication[49, p. 3]

Data Representation	DSP	LUT	FF
One 16-bit floating multiplication	1	85	167
One 16-bit fixed multiplication	1	0	0
Two 8-bit fixed multiplications	1	2	0
Four 8-bit floating multiplications	1	20	27

The implementation of fixed-point arithmetic can significantly decrease the number of required lookup tables and logic units. Table 2.3 presents the distinct hardware requirements for 16-bit floating-point multiplications, 16-bit fixed-point multiplications, and 8-bit fixed multiplications utilized for weights and biases [49, p. 3].

This suggests that quantized neural networks can reduce the necessary storage and hardware resources. The primary question that remains unresolved is whether the

accuracy of neural networks can be maintained sufficiently to ensure that classification continues to yield precise results despite reduced resource requirements.

2.1.2 Accuracy of Quantized Neural Networks

To minimize the size and enable the execution of a neural network on an RISC-V core without a floating-point unit, the neural model must be quantized. During the training of a neural network, floating-point numbers are essential for weights and biases to perform backpropagation and calculate the minor adjustments needed for weight modification. This training method, which employs floating-point operations followed by quantization, yields favorable results for large models and is referred to as PTQ. However, smaller models tend to experience more pronounced accuracy declines when post-quantization is used compared to larger models. For smaller models, it is beneficial to simulate quantization effects during the forward pass while maintaining the backward pass as if no quantization were applied [23, p. 5].

Weights bits	**Epochs**	**Loss**	**Train acc. (%)**	**Test acc. (%)**
4	20	0.6546	77	72.34
4	30	0.1016	96	74.56
4	40	0.0546	98	76.13
8	20	0.6493	77	72.56
8	30	0.1131	96	74.57
8	40	0.0543	98	76.78
16	20	0.6868	75	70.56
16	30	0.1178	95	73.48
16	40	0.0646	96	74.13

Table 2.4: Different bit widths for quantizing CONVNET and their test accuracy. [45, p. 5]

Table 2.4 presents data on the impact of training loss across varying epochs on test accuracy while altering the quantization bit width among four, eight, and sixteen. The results indicate that model quantization at 4, 8, and 16 bits does not degrade the test accuracy to the extent that it would be inferior to random guessing. Notably, the test accuracy for an 8-bit width exceeded that of a 16-bit width. This suggests that, for a specific number of training epochs, quantization can yield relatively high accuracies while simultaneously reducing memory requirements and simplifying the computations involved in the forward pass [45, p. 5].

The fundamental implementation details of quantization are elaborated in the subsequent section, as this technique is crucial for executing the forward pass on an RISC-V core without a floating-point unit and for accommodating the neural network within the constrained memory of the edge device.

2.1.3 Basics of Quantization

Quantization of a neural network involves converting all floating point inputs, weights, and biases into integer values. The basic method for converting a floating-point value to a quantized one is demonstrated by the equations for quantization 2.1, 2.2, and 2.3. Equation 2.1 illustrates the calculation of the quantized value from a floating-point value. The determined scale and zero point must also be stored to quantize the input to be classified by the neural network in the forward pass.

The zero point, calculable via Equation 2.3, exists because, for instance, an unsigned integer ranging from 0 to 255 is not centered around zero. The zero point is calculated such that 0.0 can be mapped to an integer value. Conversely, the scale is employed to ensure a sufficiently large change in the integer values for each change in the floating-point values. This ensures that small differences in floating-point numbers, which are significant for the model, can still be represented within the quantized value range [24, pp. 4–6].

Equation 2.1 explains the parameters used, such as the scale factor and zero-point.

$$(x_q = \text{round}\left(\frac{x}{s}\right) + z) \tag{2.1}$$

The equation is defined using the following variables:

- x represents the original floating-point value,
- s denotes the scale factor,
- z is the zero-point (offset).

To determine the scale and zero-point, it is necessary to consider the maximum and minimum values for both floating-point and integer values:

- f_{min}: minimum floating-point value,
- f_{max}: maximum floating-point value,
- $i_{\text{min}}, i_{\text{max}}$: minimum and maximum integer values.

The scale factor is calculated as follows:

$$s = \frac{f_{\text{max}} - f_{\text{min}}}{i_{\text{max}} - i_{\text{min}}} \tag{2.2}$$

The zero point is determined by the equation:

$$z = \text{round}\left(i_{\text{min}} - \frac{f_{\text{min}}}{s}\right) \tag{2.3}$$

The zero point and scale can be calculated per layer or for a specific number of different layers [24, pp. 4–6].

After outlining the fundamental calculation strategy for quantization, the subsequent section analyzes the effects of quantization on memory usage and power consumption.

2.1.4 Energy Consumption, Memory Requirement, and Execution Speed of Quantized vs. Non-Quantized Neural Networks

Quantization is a consideration of significant merit because integer operations require substantially less energy than floating-point computations. Specifically, floating-point operations consume approximately 20 times more energy than integer operations do. For instance, a 32-bit multiplication requires approximately 3.7 pJ, whereas an 8-bit integer multiplication requires approximately 0.2 pJ [21, p. 12]. Figure 2.3 illustrates the energy requirements of quantized and non-quantized values. Provided that the accuracy and speed of the forward pass remain within an acceptable range for the intended application, quantization may prove beneficial in reducing the energy consumption and hardware costs.

Integer	
Add	
8 bit	0.03pJ
32 bit	0.1pJ
Mult	
8 bit	0.2pJ
32 bit	3.1pJ

FP	
FAdd	
16 bit	0.4pJ
32 bit	0.9pJ
FMult	
16 bit	1.1pJ
32 bit	3.7pJ

Memory	
Cache	(64bit)
8KB	10pJ
32KB	20pJ
1MB	100pJ
DRAM	1.3-2.6nJ

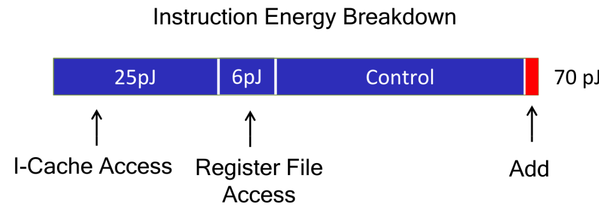

Figure 2.3: Energy use of different tasks on a 45nm 0.9V system from 2014 [21, p. 12].

The potential power savings achievable with reduced quantization bit-widths are shown in Figure 2.4. The figure demonstrates that employing a lower quantization bit-width, such as 4 bits instead of 8 bits, can result in significant energy savings, provided that the forward path maintains sufficient accuracy for the application scenario.

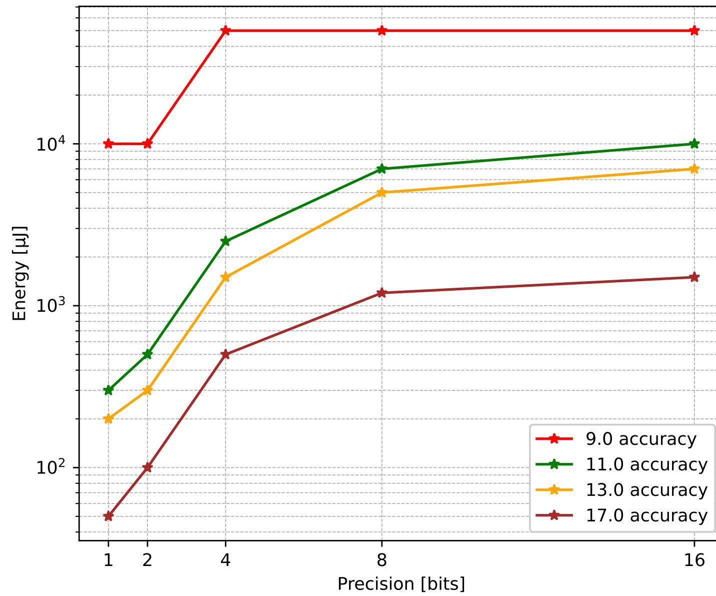

Figure 2.4: Energy consumption with different quantization levels and different bit accuracy models [34, p. 5]

An additional benefit of quantization is the reduction in operational complexity, which allows simpler hardware that can be clocked at higher frequencies. This results in decreased execution times for the forward pass in a quantized form. As illustrated in Figure 2.5, both the time consumption and memory requirements are significantly reduced. These factors present a substantial advantage, as does the reduced area required for individual computation hardware engines.

The primary reason for the decreased execution time is the processing of fewer bits, as the weights, biases, and input values are quantized to a lower bit width. This also enhances memory efficiency, as less data need to be transferred from the **R**andom **A**ccess **M**emory (RAM) to the cache and ultimately to the registers. Furthermore, the same cache can accommodate more model parameters, thereby reducing the risk of memory congestion. [29, p. 1]

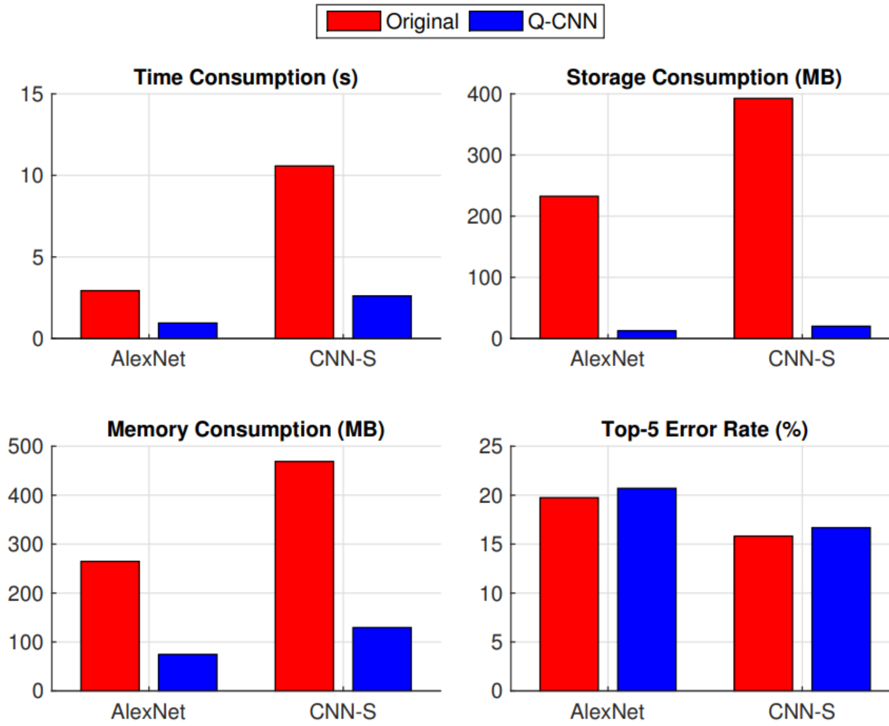

Figure 2.5: Comparing quantized and non-quantized neural networks in terms of time, storage, memory, and top-error rate. [50, p. 1]

Addressing the challenges associated with large transformer models deployed on datacenter servers, which demand substantial memory and energy and exhibit latency, remains a critical concern. Quantization initially appears to be a promising approach for mitigating these issues. However, while quantization can address these requirements, it introduces noise into the network and diminishes the performance of the neural network. This is particularly problematic for large transformer networks, such as those utilized in systems such as Chat-GPT, where quantization results in significant performance degradation [5, p. 2]. Current research is focused on resolving this issue; however, an efficient solution has not yet been identified.

The methods discussed thus far predominantly employ post-training quantization, as training with integer numbers presents considerable challenges to researchers. Some approaches aim to account for the eventual quantization of the neural network and enhance the accuracy through quantization-aware training. Initially, PTQ) is described, allowing a choice between PTQ and QAT.

2.1.5 Post-Training Quantization and Quantization Aware Training

The learning process, specifically the adjustment of neural network model weights and biases via back-propagation, is typically conducted on a device distinct from the edge device where classification is performed. Consequently, quantization can be applied after training the model has been trained using floating-point numbers. The model's parameters are then fixed, and quantization is implemented. This procedure has both advantages and disadvantages.

The primary benefit of quantizing neural networks for execution on an edge device is the reduction in memory requirements. A basic estimation suggests approximately a fourfold reduction for 8-bit quantization, as 8-bit integer values replace 32-bit floating-point values [25, p. 9].

However, this reduction in size is accompanied by a decrease in accuracy. This is due to outlier weight values, which necessitate that all other weights become less precise after quantization, as the outliers must also fit within the reduced quantization range. In addition, all channels within the same layer of a neural network must be quantized to the same resolution. This requirement results in a larger quantization error for channels that occupy a smaller range of values than for channels with a higher range of values. Both channels must use the same resolution, that is, the same scale and offset for converting floating-point values to integer form. [25, p. 5]

By quantizing not only the weights but also all parameters of the neural network, the complexity of multiplications is reduced, as multiplying smaller range values, such as 8-bit integers, is simpler than multiplying wider ranges. There is also the option to restrict weights to values that are powers of two, allowing multiplications to be converted into bit shifts. In custom hardware, this leads to performance improvements; however, on existing hardware not specifically optimized for multiple shifts in a clock cycle, this yields no significant improvement [25, p. 1].

To mitigate the error introduced by post-training quantization, quantization effects can be simulated during the forward pass, whereas back-propagation during training remains unchanged from the non-quantized state [25, p. 5]. This technique falls under the category of quantization-aware training, which is further elaborated in the following paragraph.

To mitigate the adverse effects of quantization on network accuracy, QAT can be employed as an alternative to PTQ. This approach involves the introduction of quantization errors during the training phase of the neural network. By doing so, the parameters can be adjusted in real-time, thereby accounting for these errors. Various strategies exist to implement QAT, many of which require a substantial labeled dataset to effectively introduce quantization errors, ultimately enhancing accuracy. To address the challenge of large data requirements, **S**elf-**T**rained-**Q**uantization **A**ware **T**raining (SD-QAT) can be utilized to improve accuracy while simultaneously reducing the

volume of labeled data needed for the modified training process.[48, p. 1]

2.1.6 Comparison of QAT and PTQ

Previously described methods for quantizing a neural network, such as QAT and PTQ, result in varying impacts on accuracy and latency. Figure 2.5 provides an overview of the effects on the accuracy and latency of both the MobileNet and ResNetv2 models. As illustrated in Figure 2.5, the use of PTQ generally leads to a reduction in accuracy, primarily because of the quantization error and the difficulty of accommodating sufficient differences within the considerably reduced range of each neural network parameter. Conversely, QAT has the potential to enhance accuracy, approaching the original unquantized accuracy; however, achieving this requires a substantial amount of labeled data, as discussed in Section 2.1.5. When feasible, QAT may serve as a viable alternative to PTQ. The principal advantage of PTQ lies in the fact that it does not necessitate retraining the model. The model trained with full precision values can be converted into a quantized version, albeit with a slight reduction in accuracy, as demonstrated in Figure 2.5 [41, p. 4].

The reduction in latency is also evident in Figure 2.5. It is observable that quantization reduces latency in both QAT and PTQ scenarios. This phenomenon can be attributed to the fact that computations involving quantized integer values with limited ranges, such as 8 bits, result in reduced latency compared to full precision models [19, p. 4].

Table 2.5: Comparison of different quantization methods like PTQ and QAT. [41, p. 4].

Model	Accuracy			Latency (ms)			Size (MB)	
	Orig	PTQ	QAT	Orig	PTQ	QAT	Orig	Opt
Mobilenet-v1-1-224	0.709	0.657	0.700	124	112	64	16.9	4.3
Mobilenet-v2-1-224	0.719	0.637	0.709	89	98	54	14	3.6
Inception v3	0.780	0.772	0.775	1130	845	543	95.7	23.9
Resnet v2 101	0.770	0.768	N/A	3973	2868	N/A	178.3	44.9

This chapter provides an overview of neural network optimization techniques, which are essential for developing a RISC-V-based system designed to classify the number of fingers in an image. The subsequent section discusses the use of hardware/software co-design for edge AI acceleration, detailing the acceleration techniques employed in the creation of this system.

2.2 HW/SW Co-design for Edge AI Acceleration

Hardware/software co-design, which involves the concurrent development of hardware and software to achieve specific design objectives, extends beyond merely adapting software to the existing hardware. It encompasses the consideration of hardware architecture and the identification of neural network components that may benefit from hardware acceleration. This approach, which allows for modifications to the hardware architecture to reduce energy consumption or enhance neural network inference, has been employed in the development of numerous neural network accelerators, such as NVIDIA Jetson, Google Edge TPU, and Intel Movidius. In these instances, hardware/-software co-design was utilized to achieve an optimal partitioning of hardware and software while acknowledging the manufacturing costs associated with the specialized hardware required for the newly designed architectures [47, p. 1].

The specific domain of edge devices necessitates distinct design objectives compared to non-edge devices, where power constraints may be less stringent. In the context of designing hardware architectures for edge devices intended for neural network inference, energy consumption and latency are paramount considerations. In contrast, constraints such as computational capability and thermal requirements are comparatively less critical, as illustrated in Figure 2.6.

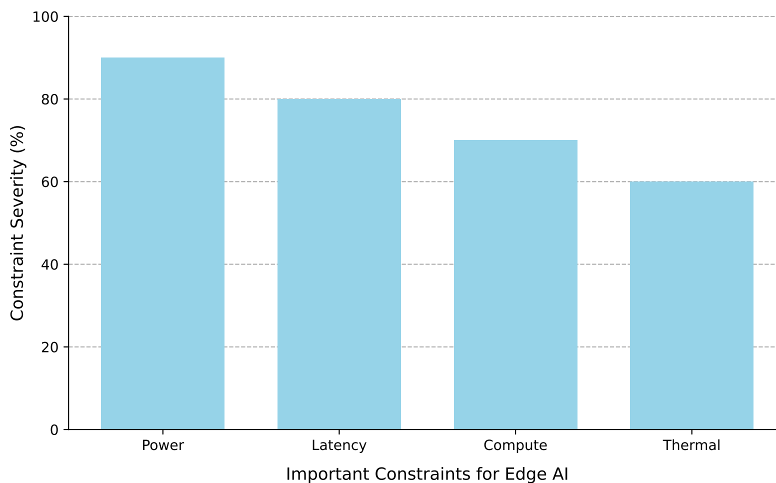

Figure 2.6: Important limits on edge devices for running neural networks. [47, p. 1]

A hardware/software co-design approach, in which the hardware architecture can be modified concurrently with software development, can enhance the energy consumption efficiency. This is achieved, for instance, by requiring additional resources for the integrated design of hardware and software. Figure 2.15 illustrates the various

optimization techniques for neural networks intended for execution on an edge device. The implementation of quantization, pruning, and co-designing specialized hardware and software solutions can improve energy efficiency. The additional cost associated with developing a specialized architecture may be justified if the energy availability on an edge device is severely constrained, potentially necessitating a co-designed solution to achieve the desired runtime.

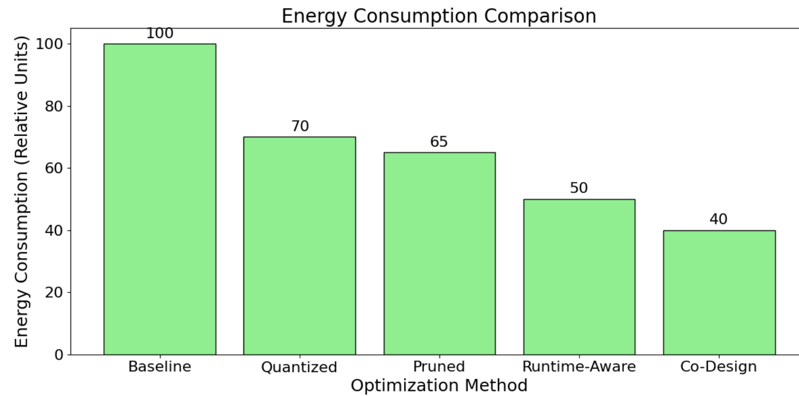

Figure 2.7: How well different methods improve neural network performance on small devices. [47, p. 4]

In the subsequent chapter, the emphasis is placed on the acceleration of edge AI on embedded systems, providing a comprehensive overview of the techniques employed to expedite the forward pass on these systems. Following this, attention is directed towards hardware/software co-designed systems specifically tailored for RISC-V cores. This approach ensures that the generally applied methods on edge devices may also be applicable to RISC-V or, at the very least, facilitate an understanding of the techniques utilized.

2.3 Edge AI Acceleration on Embedded Systems

The implementation of machine learning on embedded devices imposes significant computational demands because standard architectures are not optimized for efficiently processing data structures such as vectors and matrices. Consequently, offloading these workloads and designing hardware accelerators or specialized architectures to enhance the performance of the forward pass has become crucial for achieving both speed and energy efficiency. [17, p. 4]

Strategies for accelerating edge AI on embedded systems can be intricate, with numerous architectural designs emphasizing memory layout. Enhancing embedded systems for classification tasks can be advantageous, as exemplified by the use case

of drones. Various approaches to Edge AI acceleration exist, as illustrated in Figure 2.8. The specific functions of the forward pass can be accelerated using dedicated hardware, or a co-design approach may be employed.

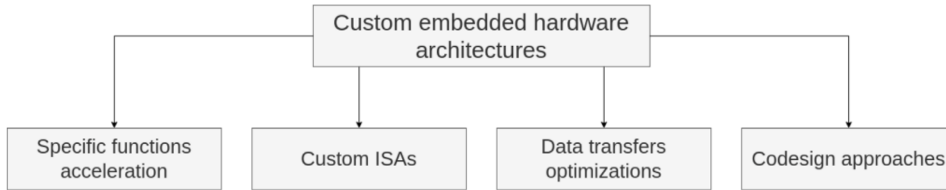

Figure 2.8: Different designs to speed up neural networks on small devices. [17, p. 5]

Various methods exist to expedite the forward pass, such as implementing custom ISA modifications that enhance the efficiency of these architectures. Co-design strategies are also employed, both with and without the assistance of machine learning, to allocate available FPGA resources more effectively. The design space exploration for the co-design strategy utilizing machine learning is divided into two components. The initial component employs a genetic algorithm to optimize the utilized hardware area. Subsequently, the second component optimizes latency and power consumption. This approach allows for the automatic adjustment of resource allocation to identify an optimal solution for the employed FPGA. However, this method is applicable only to specific **P**rocessing **E**lement (PE) structures and logic supported by the framework [17, p. 16].

To provide a foundational understanding of the structures used in embedded systems for efficiently computing the forward pass, the following chapters elucidate systolic array multiplication and the commonly employed PE structures to achieve this.

2.3.1 Acceleration via Systolic Matrix Multiplication

The convolutions required during the forward pass necessitate numerous instructions, primarily comprising multiplications and load/store operations. This results in significant energy consumption and binaries that require many cycles for classification. Fundamentally, convolution involves numerous multiply and accumulate operations [52, p. 1].

The operations necessary for convolution can be accelerated using either loosely or tightly coupled strategies. Initially, the loosely coupled acceleration for the convolutional layer is described.

For a basic overview of how a convolutional layer is accelerated using specialized hardware, refer to Figure 2.9. The buffers for the input and weight values store the currently required values, which are written into the buffers by the controller. Each

PE utilizes the buffer values to compute the convolution. Weights are transferred into the PE hardware from the top, whereas input values are introduced from the left [27, p. 4].

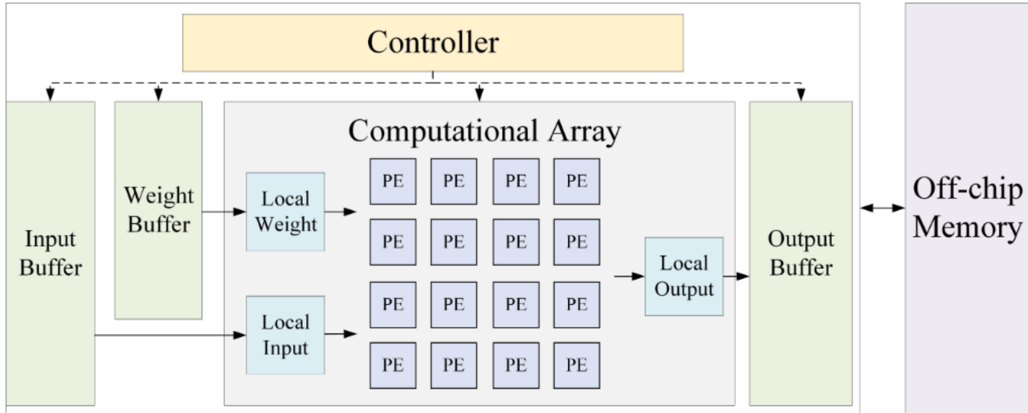

Figure 2.9: Explanation of a helpful hardware design to speed up the convolutional layer. [16, p. 31]

Various methodologies exist for implementing systolic array multiplication, which are characterized by distinct stationary and non-stationary components. The selection of elements transmitted between processing elements can be tailored for specific objectives. Options include weight and input stationary data-flows, where either the weights or inputs are initially stored in each PE, and the partial sums propagate through the processing elements. The weight stationary data-flow is prevalent in numerous neural network accelerators, such as the Google TPU, due to its scalability and applicability to both convolution and matrix multiplications [1, p. 2].

Additionally, two alternative data-flow implementations for systolic array matrix multiplication are available: output and row stationary. The output stationary data flow maintains the accumulation results within the processing elements, whereas the weights and inputs traverse the PEs. In contrast, the row stationary data flow assigns each PE its own memory for storing weights and inputs. In this configuration, the inputs are transmitted diagonally, the weights horizontally, and the accumulated summations vertically through the processing units [1, p. 2].

Weight-stationary approaches effectively mitigate memory bottlenecks, which is the primary reason for their frequent utilization. This mitigation occurs because the weights are initially loaded into each corresponding processing element, allowing only the input data to circulate through the calculation units. This necessitates the use of **F**irst-**I**n-**F**irst-**O**ut (FIFO) queues for both the input and output to ensure that the correct values are available at the appropriate time for each processing element. Figure 2.10 illustrates the data flow of the inputs and weights when the weights remain

stationary, represented by the W[j,i] values for each processing element. The inputs are clocked from left to right, as depicted by the blue lines for the input, whereas the weights are initially clocked into the structure from top to bottom, ensuring that only the input changes and is supplied by the input FIFO queues. The partial sum from one processing element is vertically connected to the processing element below, where it is added as a partial sum. Consequently, the total partial sum is calculated in the bottom row. To generate the correct input for the stationary weights, FIFO queues are essential to provide the appropriate input data at the correct cycle [1, p. 2].

The partial depiction b of Figure 2.10 reveals the internal structure of each PE. These elements utilize the sums (psum input) calculated by the PE above as input, to which they add their sum of inputs multiplied by the weight, subsequently providing their partial sum to the PE below. The output FIFO serves to synchronize the calculated partial sums, as the second PE of a row receives the correct input data one clock cycle later than the first one in that row. The output FIFO registers can be employed to add the correct results, with the second column providing the correct result one clock cycle after the first column and the third column lagging two cycles behind the first. The correct results must be aggregated to obtain the final calculation result [1, p. 2]. In summary, the data flow can be described as weights being stationary and after they were clocked in vertically, while the input data move horizontally through the structure and the partial sums move vertically down to the next PE.

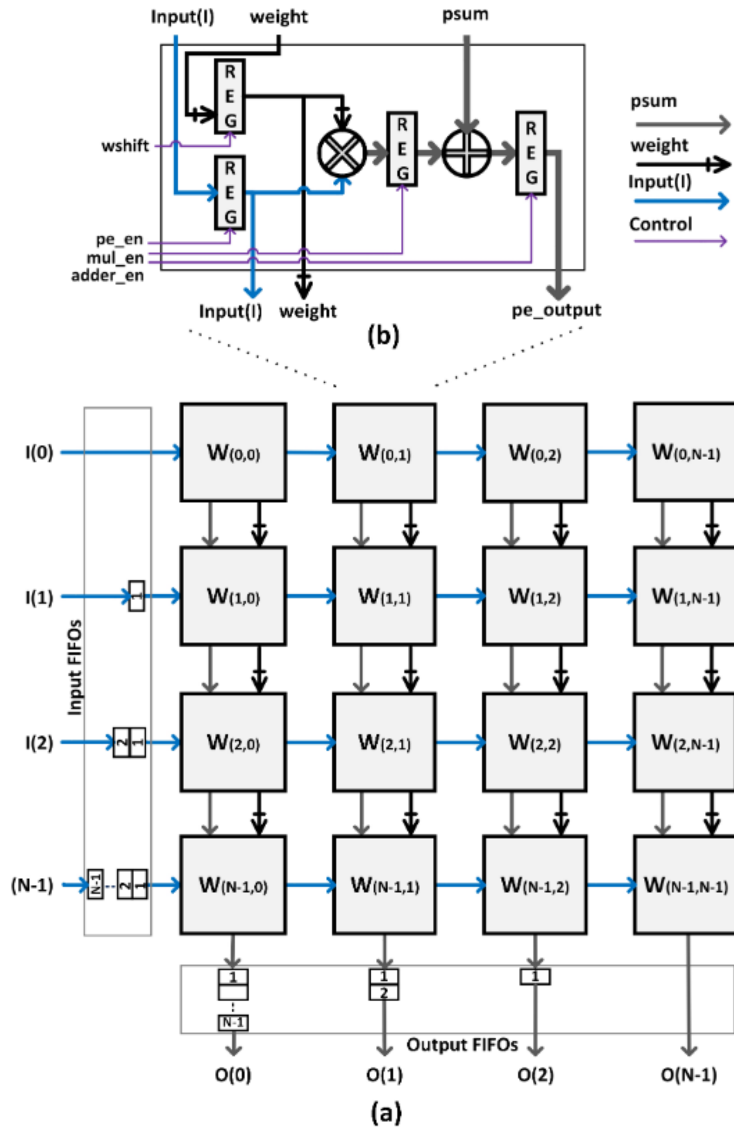

Figure 2.10: Schematic overview of systolic array matrix multiplication with stationary weights used to accelerate convolutions. [1, p. 3]

The weight stationary approach is also employed within the TPU developed by Google to enhance the forward pass of a neural network [1, p. 2]. The utility of this approach is elaborated upon in the subsequent section, which aims to elucidate how the described systolic matrix multiplication can be utilized to expedite the forward pass of a neural network executed on an embedded system.

2.3.2 Edge Tensor Processing Unit (TPU)

Enhancing performance in terms of energy efficiency and speed of the forward pass can be achieved through domain-specific hardware, particularly when alternative methods for improvement are exhausted. In data centers, a specialized TPU, developed by Google, is employed to accomplish this objective. It primarily utilizes 8-bit **M**ultiply and **A**ccumulate (MAC) units to accelerate inference [28, p. 1]. While this is applied to full precision networks, an edge AI version with quantized weights exists, and various strategies to enhance its performance are discussed.

Diverse hardware designs exist for edge AI acceleration. A significant challenge is the existence of various layer structures for different types, such as CNN, **L**ong **S**hort-**T**erm **M**emory (LSTM), **R**ecurrent **C**onvolutional **N**eural **N**etwork (RCNN), and transducer neural networks, each with distinct hotspots that may necessitate specialized design strategies.

The fundamental TPU design proposed by Google employs a two-dimensional array of processing elements with dimensions of 64 by 64. The fundamentals of how these PEs and the weight stationary data flow operate are detailed in Section 2.3.1. This approach is also utilized in the TPU proposed by Google. Additionally, the accelerator comprises two **S**tatic **R**andom **A**ccess **M**emory (SRAM) memories for the input and parameter values used within the PEs [6, p. 3].

The employed on-chip buffer appears excessively large, and the PEs are not fully utilized. This inefficiency arises from the varying layer architectures present even within CNNs [6, p. 3]. The traffic from off-chip memory to the buffer within the accelerator is substantial, necessitating the exploration of alternative strategies for its enhancement. The subsequent paragraphs provide an overview of critical considerations when attempting to accelerate the forward pass of a neural network on an edge device.

The primary observation is that the original design of the accelerator's performance is contingent on the structure of each layer, resulting in certain layers benefiting more than others from the accelerator. The trade-off inherent in employing a single accelerator design for every layer of the neural network is that, owing to the differing layer structures, the PEs and data flow are variably utilized. Additionally, the efficiency of on-chip and off-chip memories is a significant factor. This leads to the conclusion that specialized accelerators tailored to different layer structures may be advantageous, as suggested by [6, p. 3].

2.3.3 Smaller Hardware Accelerators for Different Layer Structures

To address the issue introduced by utilizing a single accelerator design for all layer structures, smaller accelerators that are specialized to some extent for different layers can be employed. This can be achieved by deploying multiple small hardware acceler-

ators tailored to the requirements of various layer structures of the neural network. A scheduler capable of allocating specific neural network layer structures to available acceleration structures must be developed. A positive aspect of this approach is that not every layer requires specialized accelerator types, as layers can be categorized into layer families. Another crucial consideration is whether the designed accelerators can be adapted to future layer structures, which is addressed by the design described in the following paragraphs [6, p. 3]. This is a noteworthy aspect, as it allows for the comparison and analysis of the effects of specialized accelerators versus a single large accelerator used for all layer structures.

The categorization of different layer architectures into families is illustrated in Figure 2.11. The right section of Figure 2.11 demonstrates that categories can be established based on the FLOP/B ratio and number of MAC operations. For instance, Family 1 is characterized by a very high FLOP/B ratio coupled with a significant utilization of multiply and accumulate operations. This family predominantly consists of CNN layers. Conversely, the left section of Figure 2.11 illustrates a scenario with a low parameter footprint, yet a high operations-per-byte ratio [6, p. 6].

In contrast, Family 3, as depicted in Figure 2.11, exhibits a medium number of MAC operations and a very low FLOP/B ratio, while necessitating a substantial amount of parameters (high memory load). The layers within Family 3 are structural components required for LSTM, fully-connected, and transducer layers. These layers, for example, derive limited benefit from the edge TPU designed by Google, as they involve minimal MAC operations and require numerous parameters from off-chip memory. Consequently, this family does not significantly benefit from a monolithic accelerator [6, p. 7].

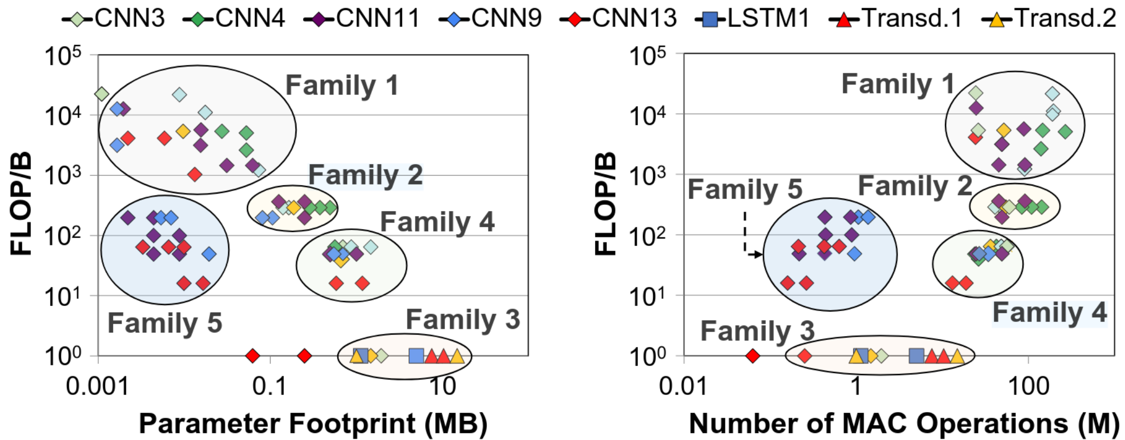

Figure 2.11: Overview of different layer structures grouped by families of layer structures. [6, p. 3]

Families 1 and 2, which are compute-intensive, such as CNNs, utilize a moderate number of parameters while performing extensive computations, resulting in a high FLOP/B ratio. These architectures can be enhanced through a design known as Pascal, as illustrated in Figure 2.12. This design minimizes the on-chip memory requirements by incorporating reusable output registers, which are highlighted with red borders in figure 2.12.

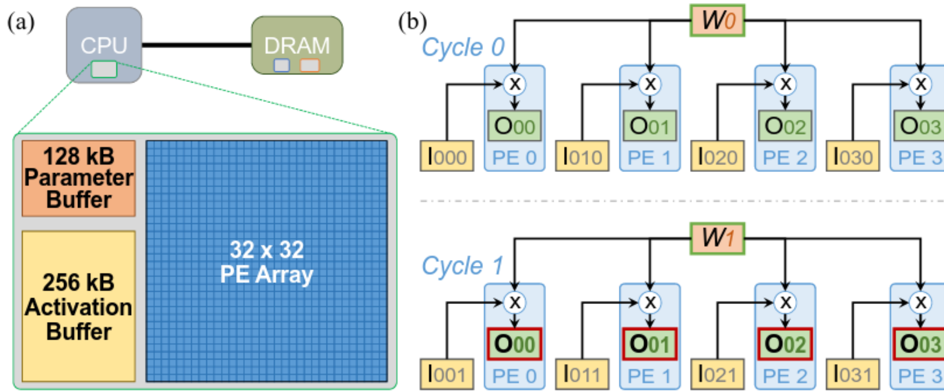

Figure 2.12: Acceleration for layer types of family 1 and family 2 which are mostly CNN layers. [6, p. 8]

For layers categorized under family 3, which predominantly consist of LSTM or fully connected layers, an alternative design approach may be advantageous. These layers are characterized by a high parameter demand, minimal requirement for PEs, and low MAC intensity. The design illustrated in figure 2.13 can be employed to develop an accelerator that enhances the performance of family 3 layers. The number of PEs is reduced compared to that in Figure 2.12 because of the limited necessity for MAC operations, with the majority of operations being **M**atrix-**V**ector **M**ultiplication (MVM) operations. Given the substantial parameter requirements of the **D**ynamic **R**andom **A**ccess **M**emory (DRAM), the design is strategically positioned in proximity to the DRAM area to facilitate an increased memory bandwidth. This configuration can be implemented on a 3D-stacked memory chip. [6, p. 7]

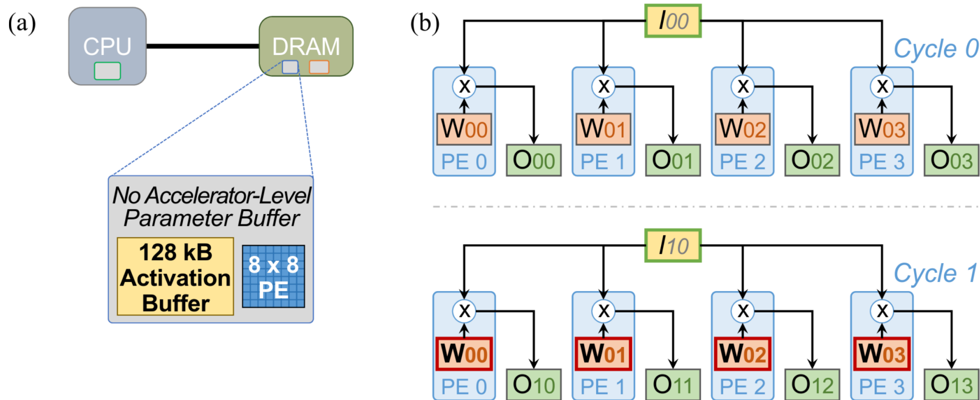

Figure 2.13: Acceleration for a layers in family 3, which are mostly LSTM or fully connected layers. [6, p. 8]

By employing distinct accelerators for different layers and utilizing a scheduler to determine the optimal accelerator for each layer structure, the throughput and energy efficiency can be enhanced by a factor of three compared to the monolithic TPU design. Additionally, latency is reduced by approximately 1.96 times when specialized accelerators are employed for each layer [6, p. 12].

This information is particularly valuable for accelerating neural networks operating on a RISC-V core, as it facilitates the decision-making process regarding whether different layers necessitate distinct accelerators or if a single accelerator suffices.

Having outlined the fundamental strategy of accelerator design for embedded systems in this chapter, the subsequent chapter explores hardware/software co-design methods and acceleration techniques specifically tailored for RISC-V-based systems.

2.4 HW/SW Co-design for Edge AI Acceleration on RISC-V-based Systems

A neural network operating on a **C**entral **P**rocessing **U**nit (CPU) without substantial parallel hardware support may result in a significantly slow inference process, rendering it unsuitable for real-time applications. It is almost inevitable to employ parallel hardware kernels to expedite a neural network that surpasses a certain size threshold. The foundation for hardware accelerators capable of performing matrix multiplications in parallel was established by the demand for graphic accelerators in video games. These accelerators are designed to manage numerous tasks concurrently. Notably, the transition from shader cores, which are specifically engineered for shader calculations, to tensor cores, which are tailored for matrix multiplication and accumulation (MAC),

is crucial for the training and inference of neural networks [55, p. 2].

The established principle of enhancing neural networks with tensors or Google's TPU serves as a benchmark for the current state-of-the-art in neural network acceleration through additional hardware.

To identify the optimal solution for partitioning specialized edge AI applications between hardware and software, hardware/software co-design techniques are employed, as the basic application of the MAC strategy may not suffice for certain edge AI systems. The decision between specialized hardware acceleration and more general solutions that can be repurposed for various machine learning architectures can be challenging in specific cases. Therefore, this section delineates the state-of-the-art hardware/software co-design processes for edge AI systems. In addition, existing methods for accelerating neural networks are examined in this chapter. For this purpose, hardware/software co-design practices and methods are also elucidated.

Figure 2.14 illustrates the design space exploration process for the neural networks and the corresponding hardware architecture on which they are to be executed. The process of partitioning an edge AI system into hardware and software components is inherently complex because of the multitude of parameters that require tuning and selection. Initially, the architecture of the neural networks is designed, involving the selection and training of the layer structure of the neural network. Rapid evaluations are conducted between intermediate steps to account for architectural constraints or design decisions of the hardware architecture, which are considered during neural network architecture design. Subsequently, the hardware architecture is selected. A decision is made regarding the use of a hardware accelerator or an ISA extension, essentially reflecting the choice between a tightly coupled and loosely coupled system. Additionally, a decision must be made regarding the memory system to be employed. For instance, on an FPGA, it is necessary to determine whether the weights should be stored in **B**lock **R**andom **M**emory **A**ccess (BRAM) or DRAM, given the size limitations. [8, p. 13]

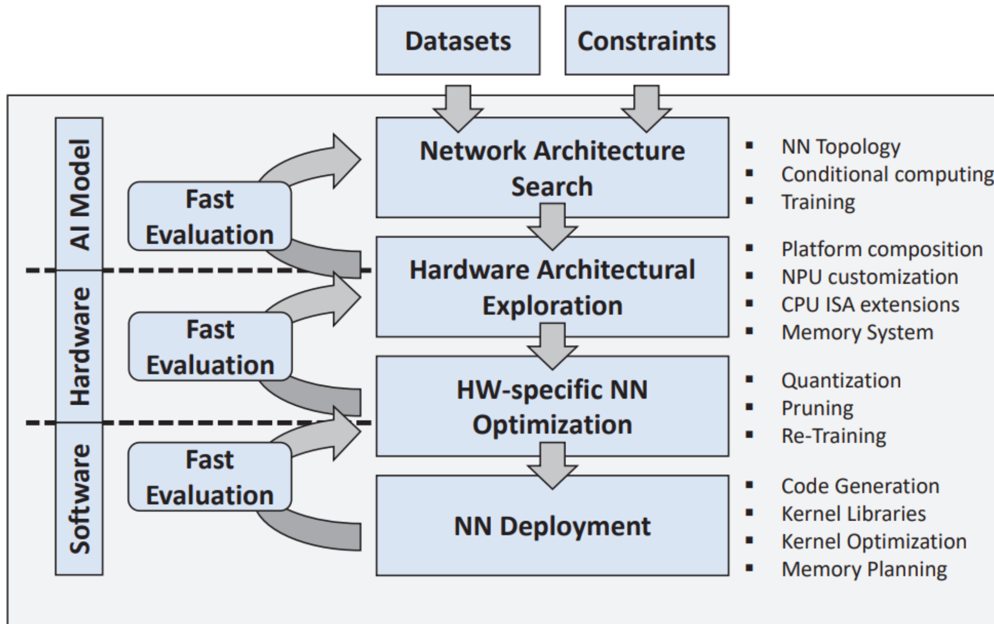

Figure 2.14: Overview of a design space exploration in the context of edge AI systems [8, p. 1]

The process of optimizing neural networks for specific hardware subsequently commences. This phase includes quantization, retraining and pruning. Pruning involves the simplification of the network by eliminating superfluous components, such as redundant weights. This approach not only enhances efficiency but also mitigates the risk of overfitting.[11, p. 1]

Various accelerator strategies are employed in edge AI systems, some of which are tightly coupled, whereas others are loosely coupled. As previously mentioned, the memory and energy requirements of neural networks can be substantial. Addressing these demands while maintaining sufficient classification accuracy is challenging.

The primary advantage of the RISC-V ISA lies in its open-source nature, which permits the extension of the architecture with custom instructions to enhance the forward pass of a neural network without incurring licensing costs. Numerous micro-architectures are compliant with the RISC-V ISA, some of which offer tightly coupled interfaces that facilitate the process of extending the micro-architecture. [13, p. 1]

2.4.1 Acceleration of RISC-V-based Edge AI Systems

The fundamental architecture of various embedded systems is shown in Figure 2.15. Multiple solution pathways are available, including off-chip hardware (depicted in blue), on-chip memory solutions (in red), and dedicated hardware accelerators (in

green). These distinct approaches facilitate the structuring and acceleration of the system at different points. Enhancements in memory latency or computational power can be achieved by modifying these components, such as by incorporating specialized hardware, altering the memory hierarchy, or implementing different bus systems [17, p. 16].

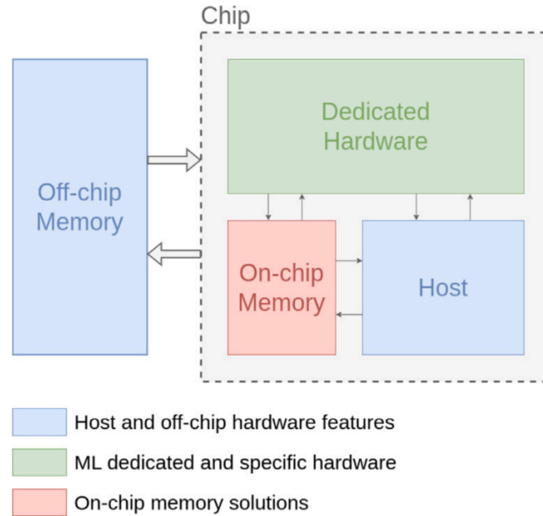

Figure 2.15: High-level overview of embedded Systems for neural network inference. [17, p. 5]

An illustration of the integration of a RISC-V core with a hardware accelerator (green) and memory system (red) is presented in Figure 2.16. The fundamental system comprises an 8×8 array of processingglspes, capable of performing MAC operations at rates of 1, 2, or 4 per cycle. The system supports integer parameter widths of eight, four, and two bits. This constitutes the primary architecture of the hardware accelerator, as shown in green in Figure 2.16. To optimize the utilization of the PEs, the memory layout and data input mechanisms are configured such that the on-chip memory system, which includes the L1 memory for network parameters, is positioned in proximity to the 2D PE array and the data mover, as depicted in green in Figure 2.16.

The PULPissimo-based RISC-V core and SOC can efficiently access the L2 memory in a single cycle, facilitated by a logarithmic **T**ightly **C**oupled **D**ata **M**emory (TCDM) connection to the L2 memory. This configuration ensures that the hardware accelerator receives the weights and activations at a sufficient speed to compute the results efficiently without being constrained by memory communication bottlenecks. The configuration of the hardware accelerator and various control registers for the data mover are established via **A**dvanced **P**eripheral **B**us (APB) communication. [26, p. 3]

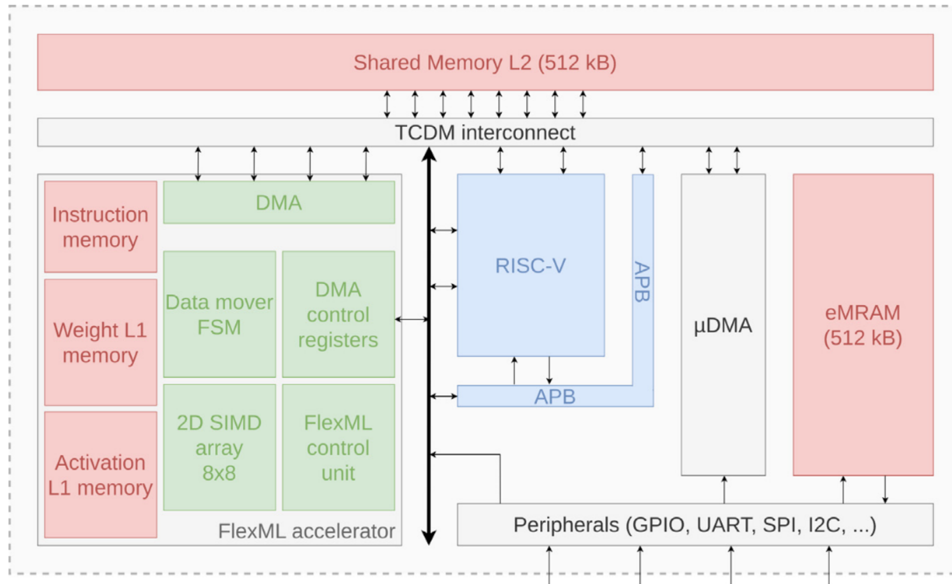

Figure 2.16: Diagram of a system called TinyVers, which uses a RISC-V-based SoC. [17, p. 6]

The data loading process can be executed in either low-power or normal mode. In low-power mode, only a portion of the shared L2 memory is utilized, specifically 64 kB instead of the full 512 kB. The chip is equipped with six switchable power domains that employ bidirectional level shifters in conjunction with isolation cells. This configuration allows for the optimization and minimization of power consumption for the RISC-V-based SOC. [26, p. 3]

The second system discussed employs a more sophisticated RISC-V SOC, incorporating a 64-bit RV64GC RISC-V core with a 6-stage pipeline and virtual memory management. It features 1MB of L2 memory and fast TCDM memory. The primary acceleration technique involves multiple MVUs managed by the RISC-V processor. These MVUs can process data bit by bit, enabling support for various quantization bit widths. The RISC-V processor coordinates the memory transfers and adjusts the precision bit width. Even the ISA is customized to enhance the efficiency of workload distribution management. The fundamental architecture is depicted in figure 2.17, illustrating the basic components of the MVU, such as the weight RAM, scaler, and quantizers. The memory within each MVU is optimized for latency, and a total of 8 MVUs are employed to accelerate neural network inference. [26, p. 10]

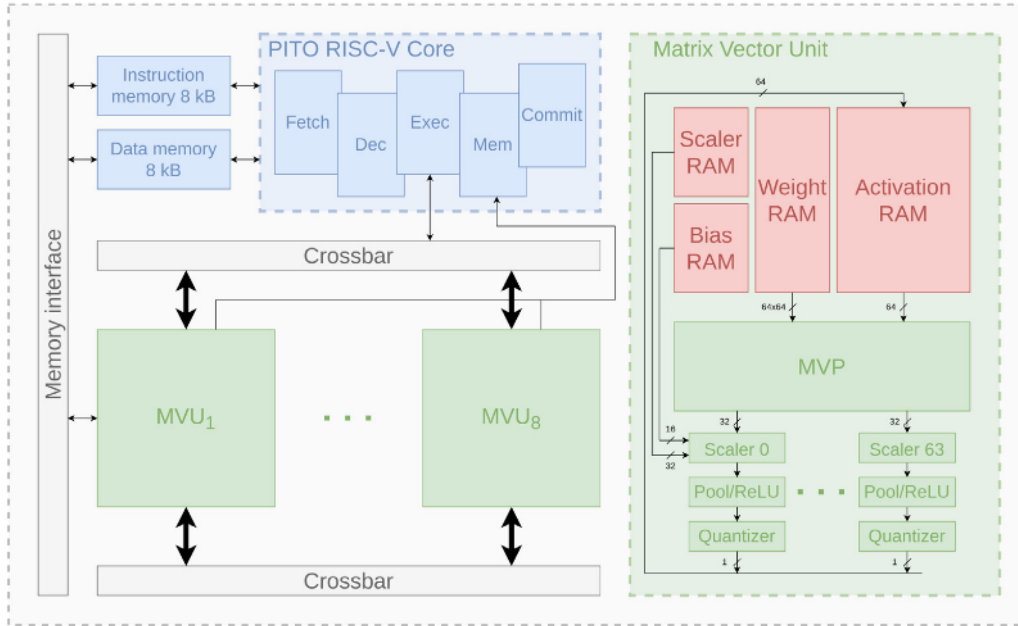

Figure 2.17: Overview of the system used to manage multiple MVUs and the diagram of MVU. [17, p. 11]

This section provides a brief overview of the hardware accelerators utilized in conjunction with RISC-V cores to enhance the forward pass. The possibility of extending the ISA of a RISC-V core is available, offering certain advantages and disadvantages. Examples of customized RISC-V instruction sets are discussed in the subsequent section.

2.4.2 Edge AI Acceleration via Custom Instructions for RISC-V-based Systems

Tightly coupled accelerators are integrated into the CPU pipeline, potentially causing pipeline stalls and affecting the main CPU pipeline. This category includes custom instructions that may enhance the forward path of a RISC-V processor. The primary disadvantage of a tightly coupled interface is its potential to influence the pipeline because additional hardware may stall the pipeline if it requires more than one clock cycle. Consequently, the added interface to the hardware acceleration module can impact the operation of the CPU and other instructions executed on it. Another significant disadvantage is the increased complexity of incorporating an instruction that affects the pipeline within the basic architecture. [3, p. 2]

This complexity arises because the core must be modified and extended, which can

also affect the maximum clock frequency of the core if the added instructions have a longer combinational delay than the longest existing one in the unextended CPU.

Overall, a tightly coupled accelerator can benefit from low-latency access to memory via the core's load/store unit and single-cycle access to the CPU registers, which may be advantageous for small calculations. [3, p. 2]

To provide an overview of existing RISC-V-based systems that utilize custom instructions, the following paragraphs describe several implementations of distinct custom instructions designed to enhance the forward pass of neural networks.

The zero-risky RISC-V core is employed, with the addition of two custom instructions to expedite the execution of CNN layers. The modified and added functional blocks are shown in Figure 2.18. Specifically, a dot product unit, referred to as DOTP, is incorporated, capable of processing two vectors, each comprising nine elements. To facilitate data transfer into the dot product unit, a custom instruction named VLOAD is introduced. This instruction is utilized to load vector elements via the load/store unit. For a vector containing nine elements, which is the maximum supported, nine VLOAD instructions are required. Subsequently, the VMAC instruction executes convolution with the stored vectors. Given that two vectors with nine elements are supported, a 3×3 convolution kernel can be computed by storing the elements within the DOTP unit and employing the VMAC instruction to initiate the calculation. A 5 $\times$ 5 kernel convolution can be achieved using three VMAC instructions. These three VMAC instructions can compute the dot product of 27 elements, as nine elements of a 3×3 kernel can be calculated in one cycle using the VMAC instruction. The block diagram of the dot product unit is depicted in Figure 2.19. To accommodate this, modifications were made to the controller, decoder, and LSU. To support custom instructions, the assembler of the GNU binutils was extended by the developers of this instruction to enable support via inline assembly in C. Otherwise, the custom instruction call was embedded in C functions through inline assembly to access them via C functions. [56, p. 3]

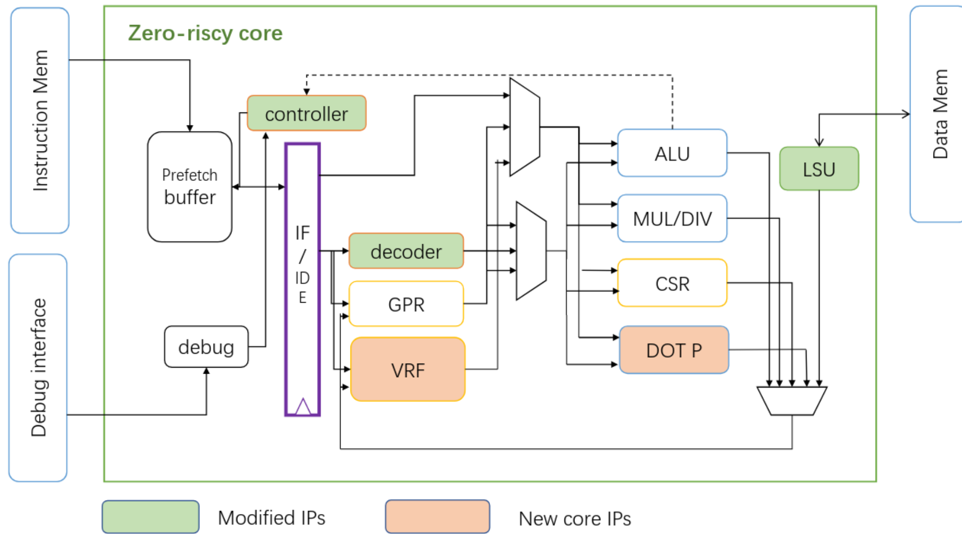

Figure 2.18: Zero-riscy core with an extended pipeline to enhance CNN performance. [56, p. 3]

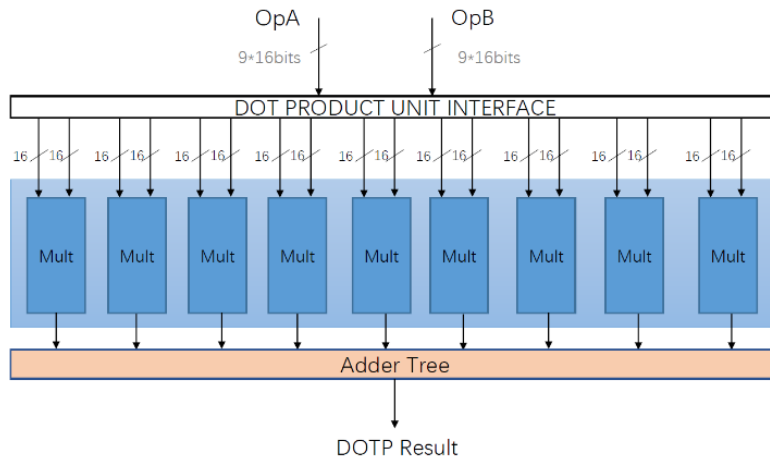

Figure 2.19: A block diagram of the dot product unit used to improve the zero-riscy core. [56, p. 3]

In convolutional neural networks (CNNs), most computations are executed within the conv2D layer. On edge devices, weights and activations are frequently quantized, making vector operations that facilitate multiple additions within a single instruction of quantized values particularly relevant. For the sub-byte operations necessary for convolutions, the Ara processor, which complies with the RISC-V RVV1.0 standard, can be enhanced to support multiply shift-accumulate operations. This enhancement benefits neural networks with low quantization widths, such as 16, 8, or 4 bits, while

adhering to the RISC-V standard, which mandates the separation of load/store operations from register calculations. To minimize the area utilized, no floating-point unit is employed; instead, the vector RISC-V extension is used alongside a proposed custom instruction to expedite convolutions. Figure 2.20 illustrates the fundamental concept of the proposed custom instruction, termed the vmacsr. This custom instruction facilitates the multiplication of multiple weights for a single register with activations in one operation, concurrently performing the right shift and accumulation. Consequently, the proposed custom instruction obviates the need for a right shift, accumulation, and an intermediate register. The requirement for a right shift prior to the accumulation of sub-multiplied sub-register values stems from the ULPPACK algorithm, which can be employed to accelerate convolution computations in software. [14, p. 3]

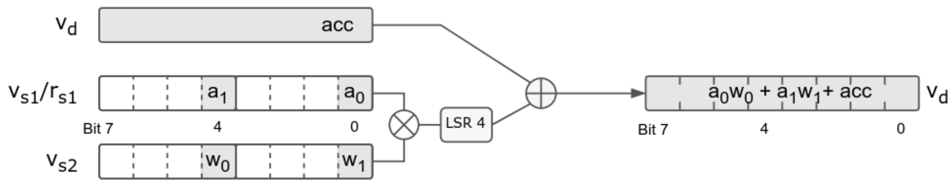

Figure 2.20: Custom instruction doing a multiply-shift-accumulate operation as one instruction by extending the RISC-V vector instruction subset. [14, p. 3]

The ULPPACK algorithm can return the accumulated result of multiple packed low-precision values from a single register, utilizing the four most significant bits as the output. Prior to accumulation, a logical right shift must be performed, as shown in Figure 2.20. This approach has the potential to enhance the efficiency of convolution calculations, as demonstrated in Figure 2.21. Additionally, the area utilized is reduced in comparison to the standard core, primarily due to the removal of the floating point unit. Only a minimal increase in area is observed, attributed to the modified vector calculation unit necessitated by the custom instruction. [14, p. 3]

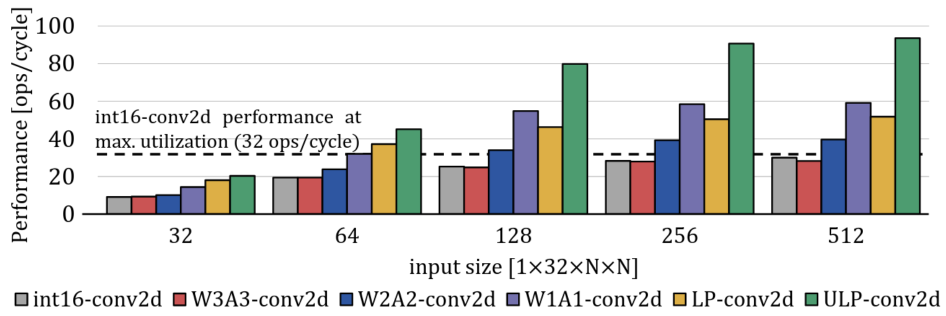

Figure 2.21: Comparison of Conv2d using a 7x7 kernel, where LP-conv2d and ULP-conv2d use a special instruction called vmacsr. [14, p. 4]

3 Design

The initial phase in the design of the neural network and the RISC-V-based SOC, intended for classifying the number of fingers in images captured by a camera connected to the FPGA board, involves the design of the neural network. Notably, constraints such as limited memory availability and exclusive support for integer operations on the cv32e40x core must be considered. The preliminary step involved selecting a neural network compiler and ensuring that the basic version operated on the FPGA. Upon successful implementation and execution of the quantized neural network on the cv32e40x, the SOC was tailored and designed to accommodate a hardware accelerator and custom instruction.

The subsequent sections elucidate various design decisions, including the chosen deep learning compiler stack and the layer structure employed to maintain a compact neural network, ensuring that it fits within the memory constraints of the Nexys 4 DDR FPGA.

This chapter also details the profiling process for identifying potential hotspots for hardware accelerators or custom instructions. Using an instruction set simulator, an initial analysis of the binary was conducted through simulation, yielding metrics to pinpoint these hotspots. For the identified sections, a conceptual design for the hardware accelerator is proposed.

The selection of the deep neural network compiler and description of the layer structure are presented in the following sections.

3.1 Selecting a Suitable Deep Learning Compiler Stack

To compile the trained neural network model, a deep learning compiler stack that supports RISC-V, quantization, and preferably operates on a bare-metal system, is required. Several options are available, necessitating a comparative analysis to select the most appropriate option. Options that fulfill the requisite functionalities include

- **T**ensor **V**irtual **M**achine (TVM) [10]
- **T**ensorflow **L**ite for **M**icrocontrollers (TFLM) [46]

The subsequent sections provide a detailed examination of these options, culminating

in the selection of one for compiling the trained model to ensure its operation on a RISC-V core without floating-point operations.

TVM is a deep neural network compiler stack that accommodates various backends, including **V**ersatile **T**ensor **A**ccelerator (VTA), **L**ow **L**evel **V**irtual **M**achine (LLVM), microcontrollers, and GPUs, among others. It supports inputs from multiple machine learning libraries, such as Pytorch, TensorFlow, ONNX, and Keras. This compatibility allows TVM to be utilized with a wide range of machine learning libraries and backends for system targets [4]. This versatility, along with the capability to employ backends specifically targeting RISC-V CPUs or FPGA logic, presents a significant advantage. It facilitates the implementation of neural networks on a RISC-V core and enables the demonstration of hardware/software co-design strategies to enhance neural network acceleration.

The TFLM framework accepts a TensorFlow Lite model as input and integrates the specified model into a binary executable suitable for various microcontrollers. Depending on the chosen toolchain, makefiles utilize the appropriate compiler and compiler flags to generate a binary that operates on the selected instruction set architecture. The input data can be loaded into a designated memory location, enabling the neural network to perform inferences using the data generated by the microcontroller. Additionally, the TFLM framework can convert input data into a C header file and store it within the microcontroller memory, which may be advantageous for testing purposes. A notable advantage over TVM is the provision of several examples that offer substantial insight into its application for different use cases. Given that many of these frameworks are currently undergoing significant development, examples are crucial for understanding how to effectively utilize the framework. The open-source project is accessible at [46].

3.1.1 Chosen Deep Learning Compiler Stack

An optimal framework for generating a neural network to be executed on a RISC-V core equipped with either a suitable hardware accelerator or custom instruction was identified. TFLM emerged as the most suitable option. Primarily, the availability of functional examples that require only minor adjustments to accommodate various use cases is advantageous. Although TVM is a viable alternative, it was observed that the quantization process for the neural network, designed to detect finger count in hand images, did not yield satisfactory results. Some floating-point operations remained within the network, which could not be executed on the RISC-V core because it only supports the **B**asic **I**nteger, **M**ultiplication and **C**ompression operations (IMC) instruction subset. These considerations render TFLM the most appropriate choice for the task of classifying finger count in hand images, where all computations can be performed on the CPU.

3.2 Selecting the Neural Network Layer Structure for Finger Count Classification

To maintain simplicity while effectively determining the finger count in hand images, a neural network structure comprising two convolutional layers, followed by pooling and linear rectifier layers, was employed. Although basic, this structure is likely to produce satisfactory results for finger-count detection.

For a visual representation, Figure 3.1 illustrates each layer as a block diagram. The diagram depicts the fundamental layer structure utilized, which includes two convolutional layers with a stride of two and a kernel size of three. Following each convolutional layer, a rectifier linear unit introduces nonlinearity, enabling the network to learn nonlinear mappings in addition to linear mappings.

The filter values of the various output channels are trained to extract distinct features from the input data. An appropriate number of output channels must be selected to accurately detect the finger count in different scenarios. Increasing the number of output channels allows for the extraction of more features, albeit at the cost of additional computational power. The filter values are determined through training, wherein the process of back-propagation identifies which features are essential to extract. [57, p. 5]

A stride of two is employed, resulting in a reduction in the resolution of the extracted output feature maps and a decrease in the computational demand of the convolutions. With a stride of two, the pixel position in the x-direction increases by two, instead of one. Additionally, for each row, one row is skipped, leading to a reduction factor of 4, as each axis is reduced by a factor of 2. [57, p. 2]

Pooling layers are used to further reduce the size of the fully connected layer required at the end of the neural network. Each max-pooling layer selects only the maximum value within the region specified by the kernel size. For instance, a 2x2 kernel slides over the image with a stride of 2, selecting only the maximum values within each current kernel position to decrease the output size. [57, p. 2]

Finally, the fully connected or dense layer utilizes the feature maps extracted from the convolutional layers, which have been reduced in size by the max-pooling layers, with rectifier linear units interspersed. The first dense layer processes the resulting values from the preceding operations and outputs 64 values. These 64 values are subsequently used to derive 5 output scores via the second dense layer. Ultimately, the neural network generates a score for each possible finger count, facilitating classification with minimal memory requirements. If quantized, it can operate on an RISC-V core without a floating-point unit.

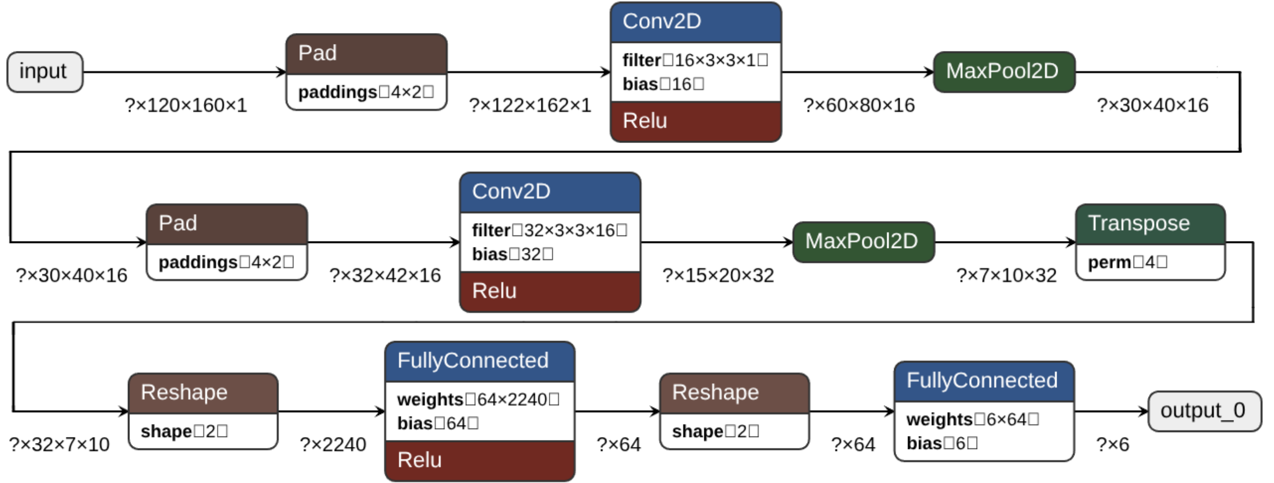

Figure 3.1: Diagram showing the neural network's design. Made with Netron [36].

3.3 Profiling and Runtime Analysis

Various methodologies exist for analyzing the binary produced for a RISC-V IMC processor. The initial consideration is whether the binary functions correctly without floating-point operations and yields accurate classification results. This can be assessed using an **I**nstruction **S**et **S**imulator (ISS) with a modified UART print function, which enables the observation of the program's debug output in the console where the ISS operates. This step is crucial when a binary is cross-compiled for a different target to verify its execution. Metrics, such as the count of executed instructions, can be obtained from Spike. A log file detailing which instructions were executed, their frequency, and their sequence can be generated. Several ISS frameworks are available for RISC-V processors, including Spike, QEMU, ETISS, and a SystemC-TLM simulator, which can be utilized for this purpose.

Spike is particularly suitable for initial analysis, as it allows for the verification of the binary's functionality and provides an overview of the required instructions, facilitating runtime estimation with a known **C**ycles per **I**nstruction (CPI). If the CPI is unknown, it is possible to examine the micro-architecture to determine the cycles required for each instruction and estimate the CPI. In addition to the total count, the Spike simulator facilitates instruction profiling, providing a granular breakdown of the execution frequency for each individual instruction.

The CPI can be calculated by understanding the cycles associated with specific instructions and the frequency of their execution [20].

To obtain initial feedback on whether the binary operates on a RISC-V core without

a floating-point unit and with a custom memory layout, an ISS for a RISC-V core is employed. Spike was selected for this purpose because of its efficient method of printing debug output via the primary kernel [40], which emulates an **O**perating **S**ystem (OS) and redirects all characters typically output via UART to the console. Consequently, the output, usually directed through the UART, can be proxied to the console on the host computer.

The statically linked binary, compiled using the framework provided by TFLM [46], can be simulated. This framework employs the RISC-V GCC compiler from GNU [39]. Makefiles are utilized to compile the binary for simulation with Spike, and the relevant Makefile is located within the tensorflow/openocd directory.

To simulate the compiled binary, the command listed in 3.1 can be employed to run the simulation. This facilitates the initial estimation of the runtime for the forward pass of the neural network. This estimation is instrumental in predicting the time required to classify a single image.

Listing 3.1: Spike command with including the **P**rimary **K**ernel (PK) and producing an commit log with -g of the finger count binary

```
spike --isa=RV32IMC -g ../pk -s finger_count
```

The output generated by spike is illustrated in Listing 3.2. Utilizing spike facilitates an early estimation of the runtime. For the purpose of estimation, a CPI of 1.5 is currently employed to account for the numerous multiplications and load/store operations executed during the forward pass. On the targeted microarchitecture, these load/store accesses may require two clock cycles per instruction if the memory access is not word-aligned. [12, p. 18] The TFLM framework employs int8_t values, resulting in memory access that is not aligned with the 32-bit word size.

```
0 fingers score: -108
1 fingers score: -103
2 fingers score: -93
3 fingers score: -32
4 fingers score: 32
5 fingers score: 74

Classified as 5

1547100 ticks
154164235 cycles
154164237 instructions
```

Listing 3.2: Output of spike ISS to estimate runtime of forward pass via instruction count and CPI

With an instruction count of 154 million, clock frequency of 50 MHz, and CPI of 1.5, the runtime estimation is determined as described in Equation 3.1.

$$\text{CPU Time} = \frac{\text{Instruction Count} \cdot \text{CPI}}{\text{Clock Frequency}} = \frac{154{,}164{,}237 \cdot 1.5}{50 \cdot 10^6 \frac{1}{s}} \approx 4.625 \text{ seconds} \tag{3.1}$$

The ability to modify parameters, such as the number of output channels in the convolutional layers, is advantageous because it can significantly impact the runtime and memory requirements. Utilizing the outlined procedure allows for the evaluation of various layer configurations and provides feedback on the runtime required for the forward pass, which is instrumental in identifying the optimal layer structure. Additionally, early feedback is provided regarding the compatibility of the binary with the RISC-V core. For instance, a quantized binary generated with TVM would be incompatible with a RISC-V core lacking a floating point unit. Consequently, TFLM was selected, to produce a functional binary for the RISC-V core.

The basic binary for the forward pass of the neural network has now been verified to operate on an ISS. The SOC for the cv32e40x must be designed to accommodate UART printing, as well as the integration of a custom hardware accelerator and custom instruction to enhance the speed of classification beyond what is achievable without custom hardware.

3.3.1 Basic Hotspot Identification

To establish a foundation for enhancing the forward pass of the neural network employed for classification, an initial overview of computational hotspots can be generated through a histogram of program counter values. This approach provides a preliminary understanding of where significant computational resources are expended, thereby offering insights into areas that may benefit from acceleration using custom hardware. The histogram of program counter values can be constructed using Spike, and the addresses can be translated into line numbers corresponding to the C++ source code files, as illustrated in Listing 3.3. Utilizing Spike, a file containing addresses and a count of the frequency with which instructions at these addresses are executed can be produced, as demonstrated by the command in Listing 3.4.

Listing 3.3: Changing addresses to line numbers using addr2line from the RISC-V GNU compiler Toolchain [39].

```
riscv64-unknown-elf-addr2line -e finger_count 0x1aa2c
```

Listing 3.4: Spike command to find out how many times each instruction was run.

```
spike --instructions=500000 --isa=RV32IMC -g ../pk -s bin
```

The final histogram, as depicted in Figure 3.2, provides a comprehensive overview of hot spots and their locations within the source files. The Python script employed to generate the mapping to the line numbers of the source code utilizes addr2line, as described in Listing 3.3, to automatically translate program counter values into line numbers.

This methodology enables the identification of frequently executed spots within the extensive source code, which comprises numerous files, thereby presenting opportunities to optimize the binary at these specified locations. To enhance visualization, eight 32-bit addresses are aggregated into a single bin, facilitating a clearer overview and allowing for the display of more sections of program counter values that are frequently executed.

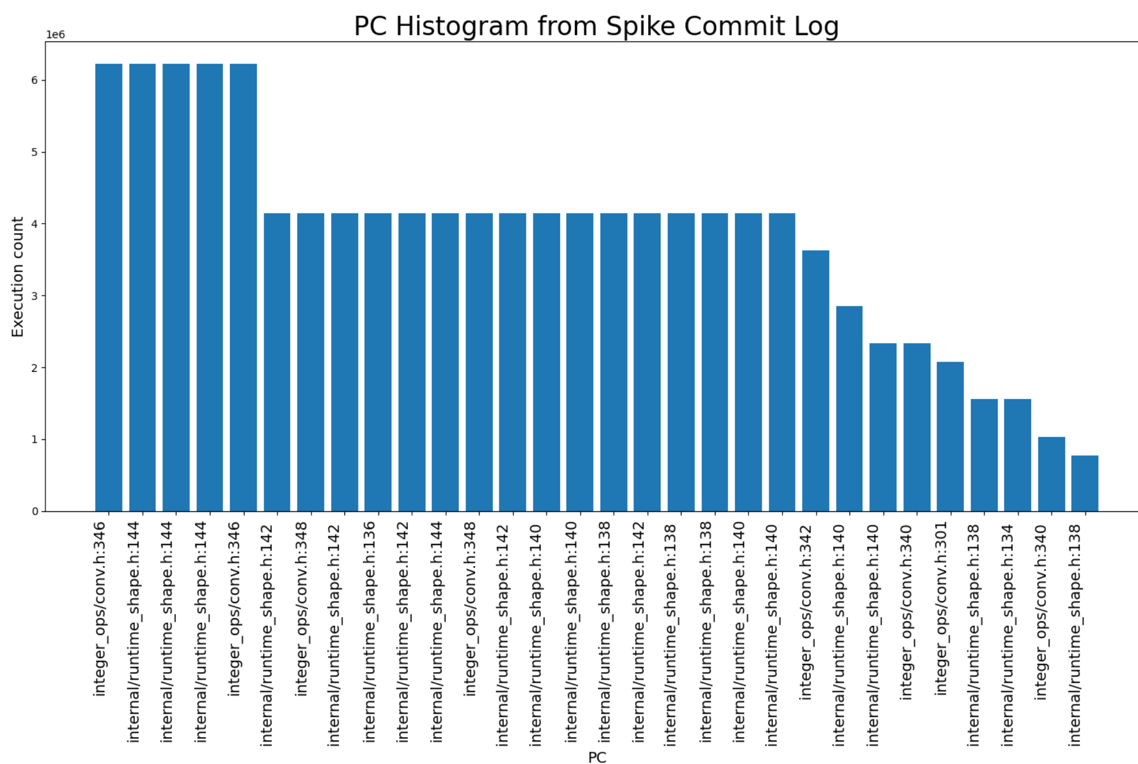

Figure 3.2: Histogram of program counter values with corresponding line-numbers of TFLM files

To identify a suitable starting point, the histogram of program counter values across all source files may prove beneficial. This approach allows the identification of sections within the source code that are executed more frequently than others, potentially

suggesting opportunities for the development of a hardware accelerator.

The profiling conducted thus far has resulted in Figure 3.2, which can be utilized to pinpoint code sections that may derive the greatest benefit from a hardware accelerator. The section identified using the program counter histogram is presented in Listing 3.5. Memory-mapped registers for weights and filter values can be employed to develop a hardware accelerator for the inner loop of the convolution process. Concurrently operating processing elements can execute calculations for each accumulation in parallel. Line 9 of Listing 3.5 is executed by one processing element (PE) on all currently stored filter and input data values. The precise implementation of this approach may enhance the classification speed, and its implementation is detailed in Chapter 4.5.

Listing 3.5: The code section for convolution from TFLM could work better with a hardware accelerator. You can find the code in the git repository: [46]

```
for (int filter_x = 0; filter_x < filter_width; ++filter_x) {
  const int in_x = in_x_origin + dilation_width_factor
                                 * filter_x;
  for (int in_channel = 0; in_channel < filter_input_depth;
                                        ++in_channel) {

     int32_t input_val = input_data[Offset(input_shape,
       batch, in_y, in_x,
       in_channel + group * filter_input_depth)];

      int32_t filter_val = filter_data[Offset(
        filter_shape, out_channel,
        filter_y, filter_x, in_channel)];

     acc += filter_val * (input_val + input_offset);

   }
}
```

4 Implementation

The implementation process commenced with the establishment of the neural network architecture, followed by the conversion of the neural network model from PyTorch to a TensorFlow Lite model via an ONNX file. This conversion facilitates the creation of a binary executable on a RISC-V core via TFLM. The subsequent steps, which involve the development of the hardware accelerator and custom instructions, are delineated thereafter. Figure 4.1 provides a comprehensive overview of each intermediate step necessary for accelerating the neural network. This figure illustrates the integration process of the custom instruction and hardware accelerator within the cv32e40x SOC. A detailed explanation of all these steps is provided in this section. Additionally, this chapter includes a description of the generation of the binary executable for the RISC-V core, as well as the creation, co-simulation, verification, and validation of the custom instruction and hardware accelerator.

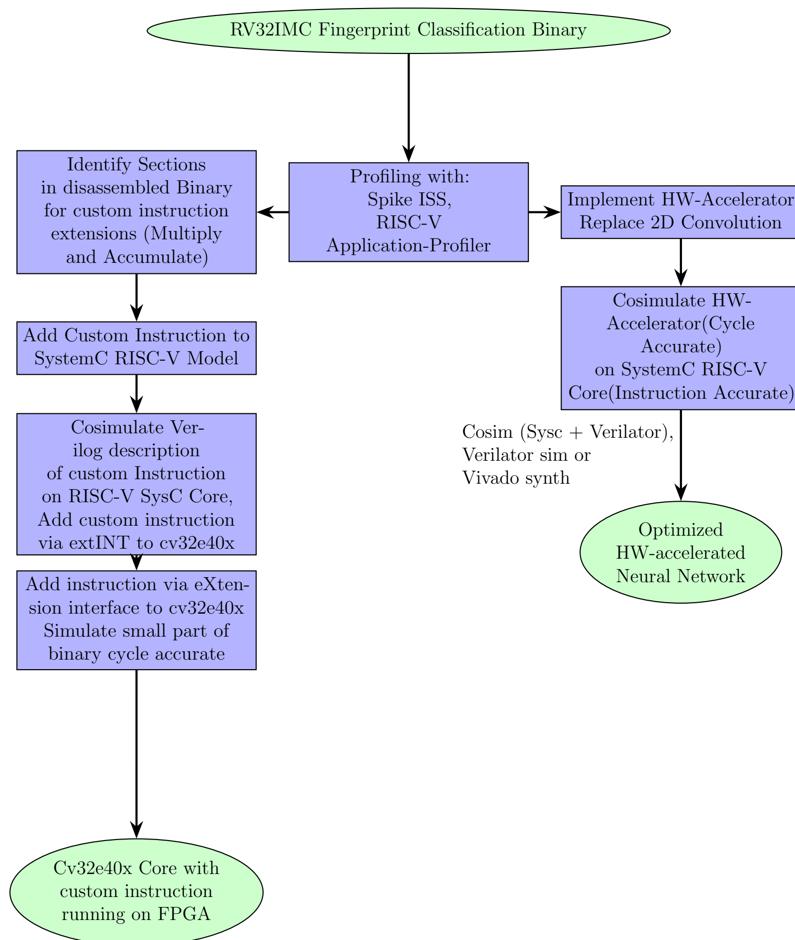

Figure 4.1: Design process for creating custom instructions and hardware accelerators to make finger count classification faster

4.1 Structure of the Neural Network Used for Finger Count Classification

Initially, the structure of the neural network was implemented using PyTorch, followed by conversion to a TensorFlow Lite model. The fundamental tool flow is illustrated in Figure 4.2. The model, including trained parameters such as weights and biases, was exported from PyTorch in an intermediate model format known as ONNX. Subsequently, the onnx2tf [22] converter was employed to transform the model into a TensorFlow format. Quantization was performed via the TensorFlow Lite framework, allowing the input range of the camera to be set, thereby obviating the need for the computation of the zero point and scale on the FPGA. Consequently, the recorded

camera pixel values can be directly input into the neural network by merely converting them from unsigned eight-bit values to signed values.

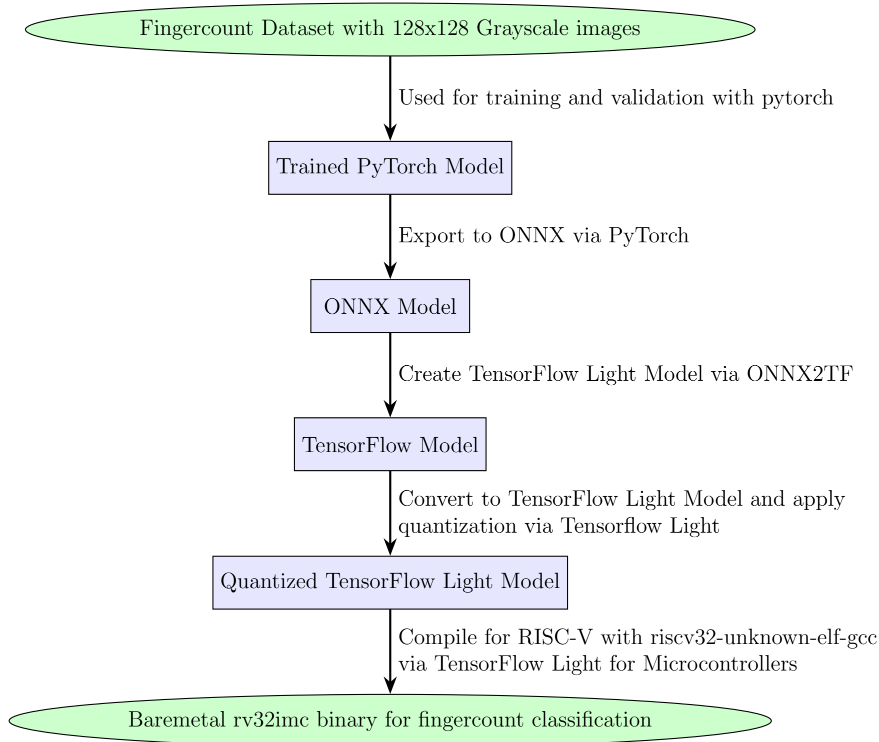

Figure 4.2: Overview of tools used to train, adjust, and compile the model for a 32-bit RISC-V core.

4.2 Training of the Neural Network and Quantization

To obtain a quantized neural network with a fixed forward path that operates on a RISC-V 32-bit core lacking a floating-point unit and utilizes only basic integer operations such as multiplication, division, and a compression instruction subset, it is necessary to first train the model and select and describe the basic architecture within a machine learning environment. PyTorch was chosen for training because of its versatility, as it supports various output formats of the trained model and is open source. Additionally, it features an efficient C++ core and CUDA support, facilitating the relatively fast training of the neural network [37].

The employed neural network is depicted in Figure 3.1, as presented in the design chapter, Section 3.2. The primary objective was to ensure minimal memory usage, ranging from 150 to 450 kB, while maintaining comprehensibility and achieving reasonably high accuracy. The initial design of the neural network extracted an

excessive number of features from 2D convolutional layers. During the optimization process, the neural network was reduced to 16 output kernels instead of 96, resulting in a reduction from 85 MB to a 500 KB binary file. The ELF binary file was generated using the TensorFlow Light Micro framework.

During the training process, the model with the lowest batch loss was selected. It is important to note that the loss may increase when training is performed on a new batch. Consequently, the weights corresponding to the lowest loss for each epoch were preserved, and ultimately, the model with the lowest loss was identified as the optimal model.

4.3 Extension of TFLM to Operate on cv32e40x Softcore

To modify the TensorFlow Lite Micro examples available in the Git repository [46], the input dimensions for the images were specified as 120 pixels in height and 160 pixels in width (160×120). Both the input size and the layer architecture are configurable. Each utilized layer must be declared in the main_function.cc file of the TFLM examples to ensure that the compilation process includes it and compiles the necessary files for the various layers employed. Listing 4.1 presents the required layers for the neural network, as shown in Figure 3.1. The sequence in which these layers are used is inconsequential; however, all the layers utilized in the neural network must be incorporated.

Listing 4.1: Added layers to TFLM to support imported tensorflow light model

```
AddConv2D(tflite::Register_CONV_2D_INT8());
AddMaxPool2D(tflite::Register_MAX_POOL_2D_INT8());
AddRelu();
AddTranspose();
AddReshape();
AddFullyConnected(tflite::Register_FULLY_CONNECTED_INT8());
AddPad();
```

In this context, the input for the neural network, specifically an image with a resolution of 160 × 120 pixels, must be processed. The images were loaded into the appropriate memory region designated as the input for the neural network. A custom frame buffer is integrated into the SOC to store the image captured by the camera, and the data are subsequently transferred from the frame buffer to the correct address region. The SOC, which facilitates interaction with the I/O devices for the cv32e40x core, requires modification. The memory region allocated for the frame buffer commences at address 0xFF000000 and accommodates 19,200 entries for each grayscale value. The SOC is configured to switch between the main memory and frame buffer

based on the address provided by the core. The frame buffer is a dual-port memory that allows the camera controller to write to it while the microcontroller simultaneously reads from it. Listing 4.2 presents the fundamental code that transfers data from the frame buffer into the necessary memory region for TFLM. This constitutes the input read function, which must be adapted to support the frame buffer, enabling real-time image reading and classification of each image captured by the camera. Listing 4.2 illustrates the modifications required for the image_provide.cc file of the TFLM examples.

Listing 4.2: The neural network gets its input from the frame-buffer and changes it to signed 8-bit values.

```
uint8_t pixel_val_uint8 = 0;
for (int i = 0; i < 120; i++){
  for (int j = 0; j < 160; j++){
    pixel_val_uint8 = framebuffer[(i*160)+j];
    image_data[(i * 160) + j] =
    (int8_t)(((int16_t)pixel_val_uint8) - 128);
  }
}
```

The primary requirements for classifying an image captured by a camera directly on the FPGA are thus met. The classification outcome of the image is displayed in a terminal via UART. To facilitate this, the MicroPrintf functionality of the TFLM framework must be modified to write the prints to the correct address corresponding to the designated address of the integrated UART module. The address chosen for the ready register was 0×80002014. This address is used for the memory-mapped signal to verify whether the UART module has completed the previous transmission and is prepared for the subsequent one. The actual 8-bit data intended for transmission via UART must be written to address 0x80002000. The fundamental communication logic for UART is located in tensorflow/lite/micro/micro_log.cc and has been adjusted to accommodate the custom UART module. For UART communication, a simple UART implementation of picorv32 [38] was employed and integrated into the SOC with the cv32e40x as the processor.

The essential requirements to support the classification of images with TFLM operating on the FPGA with images captured by the camera are met with the aforementioned modifications to the TFLM framework. The camera is directly connected to the FPGA's 12-pin PMOD ports. The memory layout and the SOC are described in the subsequent chapter.

4.4 Integration of the Core on the System-on-Chip and Simulation

Figure 4.3 provides an overview of the fundamental system and illustrates how the classification is obtained from the soft core operating on the FPGA. The binary utilized for classification, generated with TFLM, is executed on the cv32e40x with a memory layout detailed in Section 4.4.1. To evaluate the hardware accelerator and the system's basic functionality, a cycle-accurate simulation was conducted using Verilator. Given that the cycle-accurate simulation requires approximately 5–10 min for the entire forward pass, the hardware accelerator can be co-simulated, allowing only the accelerator to be simulated cycle-accurately while the core is simulated within a TLM. This approach enables the entire forward pass to be simulated in approximately 15 s, thereby providing quicker feedback on the performance of the hardware accelerator. Nonetheless, both simulation methods, cycle-accurate simulation and co-simulation, are valuable, offering either a precise simulation or a faster, albeit less accurate, alternative.

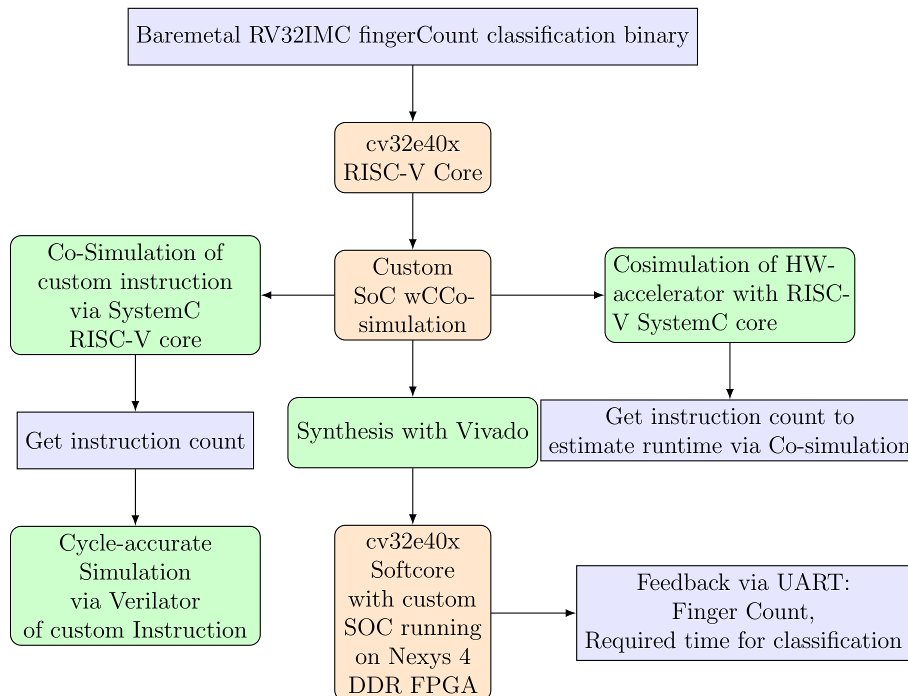

Figure 4.3: Overview of tool-flow for simulation and co-simulation of the whole finger count classification System

4.4.1 Basic Memory Layout

The memory map of the system designed to classify the finger count in hand images was constructed, as shown in Table 4.1. The registers for UART communication and the cycle counter register are available at the addresses specified in Table 4.1.

The frame buffer is allocated 19,200 bytes to store a grayscale image with a resolution of 160 × 120 pixels captured by the camera. The frame buffer is read-only for the processor, whereas the camera exclusively writes to it.

The memory-mapped registers for the hardware accelerator are located at the addresses shown in Table 4.1 and are utilized to input the 8-bit data and filter values into the hardware accelerator. The input and filter value registers are 32 bits in length and can hold four 8-bit input or filter values. The three input registers and three filter value registers accommodate the necessary values for a convolution operation with a kernel size of 3 × 3, which requires nine 8-bit values that can be stored in the registers. Each input register d_a to d_c contains three values and is multiplied by the corresponding filter value in the filter value registers f_a to f_c. The register d_b_2 is employed to add an additional 8-bit value, ensuring that memory accesses are consistently 32-bit aligned, i.e., aligned with the processor's word size. This alignment is achieved by reading one value separately, ensuring that every address used to access the main memory is divisible by four with no remainder. Consequently, the memory content read from the main memory can be directly written into the hardware accelerator's memory-mapped registers, with all memory accesses aligned to 32-bit (4-byte) boundaries.

The custom instruction essentially represents addition and multiplication within a single instruction and utilizes an input offset register separately. This register is infrequently modified and introduces minimal computational overhead. The input offset remains constant during the convolution process as the 3 × 3 convolution kernel traverses the image. Therefore, a memory-mapped register is particularly advantageous in this context.

Address	Name	Description
0x80002000	uart_addr	UART transmit/receive register
0x80002014	uart_ready	UART status register
0xF000002C	cycle_count	Cycle counter register
Camera Framebuffer		
0xFF000000	frame-buffer	Start of 19,200-byte camera frame-buffer
FRAME_BUF_SIZE	19200	Size of frame-buffer (8-bit entries)
Hardware Accelerator Registers		
0xF0000004	d_a	Integer input value A
0xF0000008	d_b	Integer input value B
0xF000000C	d_b_2	Integer input value B2
0xF0000010	d_c	Integer input C
0xF0000014	f_a	Filter value input A
0xF0000018	f_b	Filter value input B
0xF000001C	f_c	Filter value input C
0xF0000020	hw_input_offset	Input offset for hardware accelerator
0xF0000024	hw_result	Computation result register
0xF0000028	fin	Accelerator finished flag
Custom Instruction Registers		
0xF0000030	cust_instr_input_offset	Offset input for custom instruction

Table 4.1: Memory Map of the FPGA Peripherals and Custom Instruction Registers

4.4.2 Co-simulation for Enhanced Verification of the Hardware Accelerator

To obtain prompt and efficient feedback regarding the accuracy of the hardware accelerator's output, co-simulation utilizing a RISC-V SystemC core [33] was employed. This approach facilitates the exploration and testing of custom instructions and the hardware accelerator in a more expedited manner. The primary rationale for adopting a co-simulation strategy is the extensive duration required for a complete cycle-accurate simulation of the forward pass on the RISC-V core, which was conducted on an i9-9900k processor. Figure 4.4 illustrates the overarching methodology employed for the co-simulation and simulation for verification purposes, as well as the validation process. The procedure is stratified into layers, ranging from the co-simulation layer to the cycle-accurate simulation layer, culminating in the validation of the FPGA through images captured by a camera.

The RISC-V SystemC core was augmented to map the memory-mapped registers to the hardware accelerator. The hardware accelerator can be described using either SystemVerilog or VHDL. In instances where the accelerator was described in VHDL, the files were converted to SystemVerilog files using YOSYS [54]. This conversion is

necessary because Verilator [43] supports only SystemVerilog, and the utilized cores are also written in SystemVerilog.

Figure 4.4 illustrates the overall design flow and tools employed to expedite the forward pass. The left pathway of Figure 4.4 represents the co-simulation of the hardware accelerator, whereas the right pathway depicts the custom instruction. All nodes are organized into layers, as specified on the left side of figure. The hardware accelerator undergoes co-simulation, allowing the generation of a waveform of the hardware described in SystemVerilog or VHDL. This co-simulation process, which lasted approximately 20 s, was instrumental in both the development and verification of the hardware accelerator. The entire forward pass, which executes approximately 137 million instructions, can be simulated efficiently, enabling the verification of the accelerator by acquiring the classification scores. This approach offers a significant advantage over cycle-accurate simulations, which are time-consuming. Although a small test case can be used to verify the hardware accelerator, the complete result of the forward pass cannot be obtained using this method. Although cycle-accurate simulation would provide the required cycles for classification, these can also be estimated using the CPI value. The instructions utilized and other metrics obtained through co-simulation are detailed in Section 5.1 in Table 5.1.

The custom instruction is incorporated solely into the TLM layer and is not co-simulated in this study. This approach allows for an early assessment of the utility of custom instructions. The identified custom instruction was integrated into the core at the cycle-accurate layer, specifically at the register-transfer level, enabling the acquisition of precise cycle values for the forward pass. As depicted in Figure 4.4, at the cycle-accurate simulation layer, the eXtension interface of the cv32e40x is utilized to verify the custom instruction and to obtain cycle count values for runtime estimation.

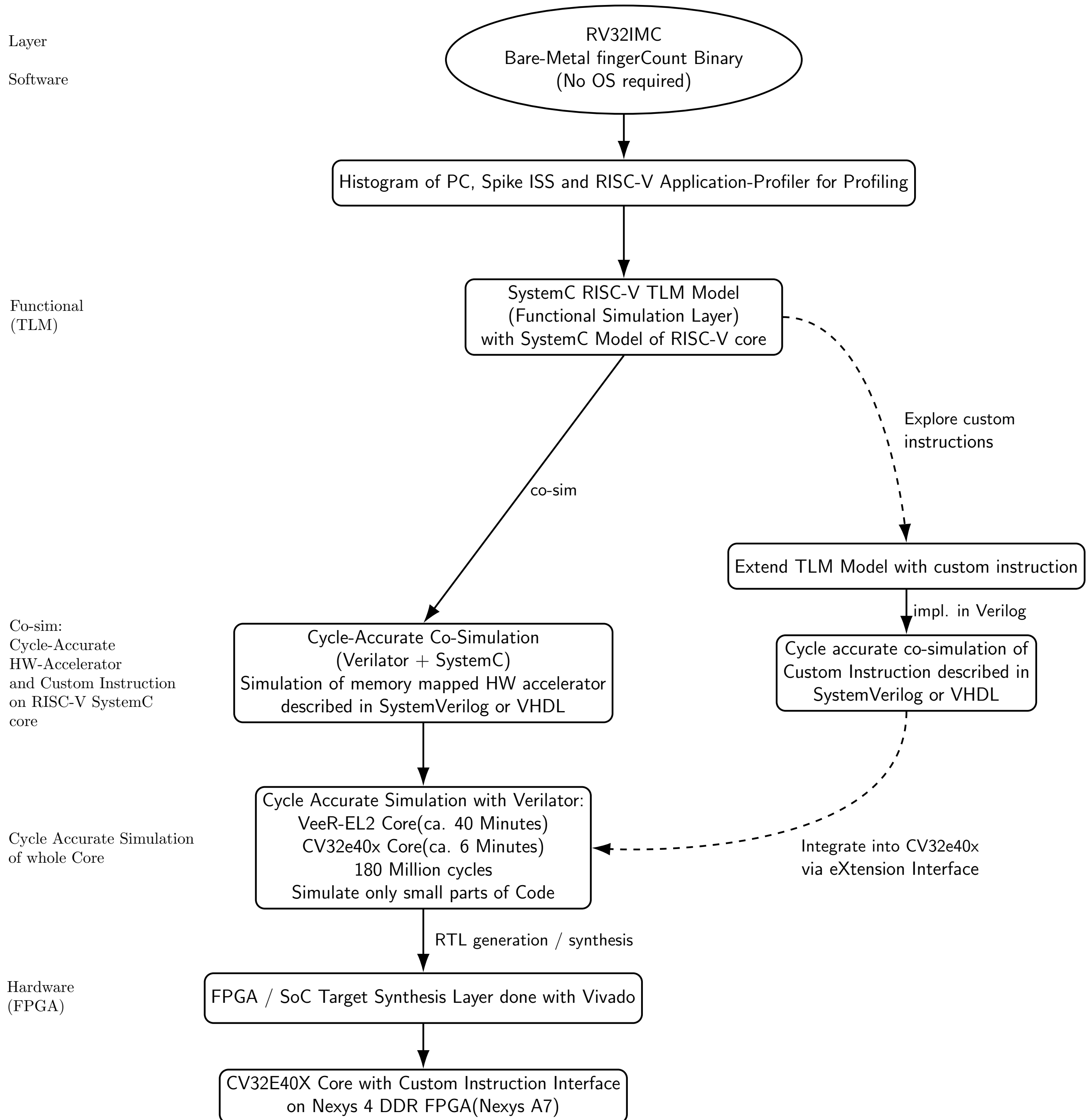

Figure 4.4: Overall tool flow used for adding, simulating, and checking the hardware accelerator and custom instruction for each abstraction layer. The cycle-accurate simulation time was measured on an i9-9900K.

Within the OpenOCD directory, a specialized make command, $makecompile_cosim$,

is available in the Makefile for TensorFlow Lite Micro. This command compiles a binary that aligns with the memory layout requirements of the co-simulator. The resulting binary can be loaded and executed to perform co-simulation within the 'simulation' folder located inside the 'rv_cosimulation' directory. This co-simulation facilitates the generation of waveforms for the hardware accelerator and allows the integration of custom instructions into the co-simulator to evaluate the reduction in instruction count achieved by incorporating a custom instruction for the forward pass. The overall runtime of the classification process can be estimated using the CPI value, which pertains to the core in use. For the binary employed, this value is approximately 1.4, as determined by executing the forward pass on the FPGA.

The output generated by the co-simulator is illustrated in Listing 4.3.

Listing 4.3: Output of co-simulation with RISC-V SystemC core and hardware accelerator

```
file: fCount.ihex
------------------------------------------------------------
Classifying test image with 5 fingers
filter input depth: 1, filter_width: 3, filter_hight: 3, output_depth: 16,
    output_hight: 60, batches: 1
filter input depth: 16, filter_width: 3, filter_hight: 3, output_depth: 32,
    output_hight: 15, batches: 1
invoke executed
0 fingers score: -108
1 fingers score: -103
2 fingers score: -93
3 fingers score: -32
4 fingers score: 32
5 fingers score: 74

Classified as 5

ECALL Instruction called, stopping simulation
*******************************************
Simulation time 1381005970 ns
*******************************************
# data memory reads: 29326549
# data memory writes: 2065887
# code memory reads: 137100598
# instructions executed: 137100597
*******************************************
ECALL
Simulated 6255685 instr/sec
```

The results of the co-simulation can be utilized to estimate the runtime, as the instruction count required for image classification is provided. This facilitates the

design of the hardware accelerator and custom instructions, as feedback can be obtained more expeditiously.

The co-simulation proved beneficial for experimenting with various accelerator designs and assessing the effects of different accelerators. The subsequent section delineates the final accelerator identified for the segment of the binary discussed in Chapter 3.3.1, which focuses on profiling and identifying computationally intensive areas of the binary.

4.5 Hardware Acceleration of the Forward Pass

The acceleration of the specified section of the convolutional layer presents several challenges that must be addressed. Co-simulation may facilitate the evaluation of whether the hardware accelerator architecture produces favorable outcomes. Key architectural decisions include whether the hardware accelerator should have independent access to memory or utilize memory-mapped registers, allowing the core to access the memory and write the necessary data for computations.

Memory-mapped registers were selected because, although the additional operation required for the core to write data to them introduces an extra step, the ability to write multiple eight-bit values into a single 32-bit register outweighs the associated costs.

Another consideration for the hardware accelerator, beyond the interface between the core and the accelerator, is its complexity and reusability across different neural network layers. To classify images captured by the camera, the two convolutional layers account for the majority of the computations, as illustrated in Figure 3.2. The primary objective of the hardware accelerator is to enhance the performance of the convolutional layer, as it offers the most significant benefits, given that most executed instructions result from the convolutions of activation values with trained kernel matrices. This topic is explored in greater detail in Section 3.3.1, which addresses the design and profiling of binaries.

The fundamental operation that the accelerator enhances is depicted in Figure 4.5.

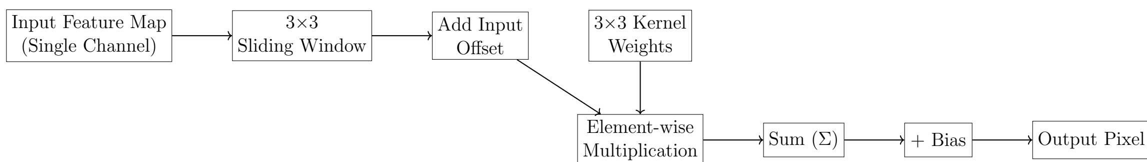

Figure 4.5: Convolution operation with input offset applied before element-wise multiplication.

The primary objective of this chapter is to elucidate the implementation of the

hardware accelerator and the tools employed in this process. Following the co-simulation of the accelerator, a cycle-accurate simulation with a custom SOC was utilized to multiplex the data bus from the CPU to either the hardware accelerator or the main memory. This approach ensures that, depending on the address of the data interface of the cv32e40x core, the data are directed to either the main memory or the memory-mapped registers of the hardware accelerator. The specific memory addresses employed are listed in Table 4.1.

In the context of convolution operations, an input offset is adjusted once per convolution, with a single register designated for the entire process of the convolution operation. Additionally, there are three 32-bit registers, each prefixed with the letter d, corresponding to every 8-bit input data value. These input values are essential for convolution at a single kernel position, representing all nine input values required for a 3×3 kernel. To ensure 32-bit alignment in memory access, four registers are employed because the memory address offsets for the input values are not divisible by four. The d_b_2 register is used for this purpose. The initial three input values are aligned in memory, beginning at an address divisible by four, without a remainder. The subsequent three values are stored in the memory with an offset determined by the resolution of the entire input image. For instance, with an input image width of 160 pixels, the values are stored with an offset of 160 plus two. The second row has an offset of 162, as two pixels are added to each side of the image for padding. This arrangement allows a 3×3 kernel to compute the convolution even at the image edge without accessing incorrect memory addresses for kernel parts overlapping with values outside the actual image. This situation arises when the central value of the kernel is at the image border without padding. Consequently, the two-value padding results in an offset of 162 for the subsequent row of input data, which is not divisible by four without a remainder. Therefore, the memory is accessed with an offset of 160, containing the correct input data values with offsets of 162 and 163. These values are stored in the d_b register, which utilizes a single memory access to write these two 8-bit values into the register. Only the value with an address offset of 164 is absent, necessitating an additional memory access and separate storage in the register named d_b_2. This register contains the final eight-bit value of the input data. Thus, each register prefixed with d for input data holds one row, with the exception of d_2, due to the non-word-aligned offset. The general structure of the data preparation for the input values in the C code is illustrated in Listing 4.4.

Listing 4.4: Transferring data from main memory to special hardware registers

```
*d_a   = *((int32_t*)(input_data + data_offset));
*d_b   = *((int32_t*)(input_data + data_offset + 160));
*d_b_2 = *((int32_t*)(input_data + data_offset + 164));
*d_c = *((int32_t*)(input_data + data_offset + 324));
```

The filter values determined through training were configured without padding, allowing each row to be accessed via a 32-bit aligned address. Consequently, each row can be retrieved with a single memory access and a single memory-mapped register, necessitating the use of three filter-value registers, as illustrated in Table 4.1.

Once the data are accurately written to the registers, the accelerator employs processing elements for each required multiplication and addition. The entire convolution process involves nine additions of the input offset and multiplication with the filter offset. The processing elements are utilized such that each input value is multiplied by the corresponding filter value after applying an input offset. The values are 8-bit wide, enabling the generation of 8-bit signed multipliers for each element required for one convolution. This approach minimizes the combinational delay and facilitates the access of multiple 8-bit values from memory, thereby enhancing the convolution speed compared with the base forward pass without the accelerator.

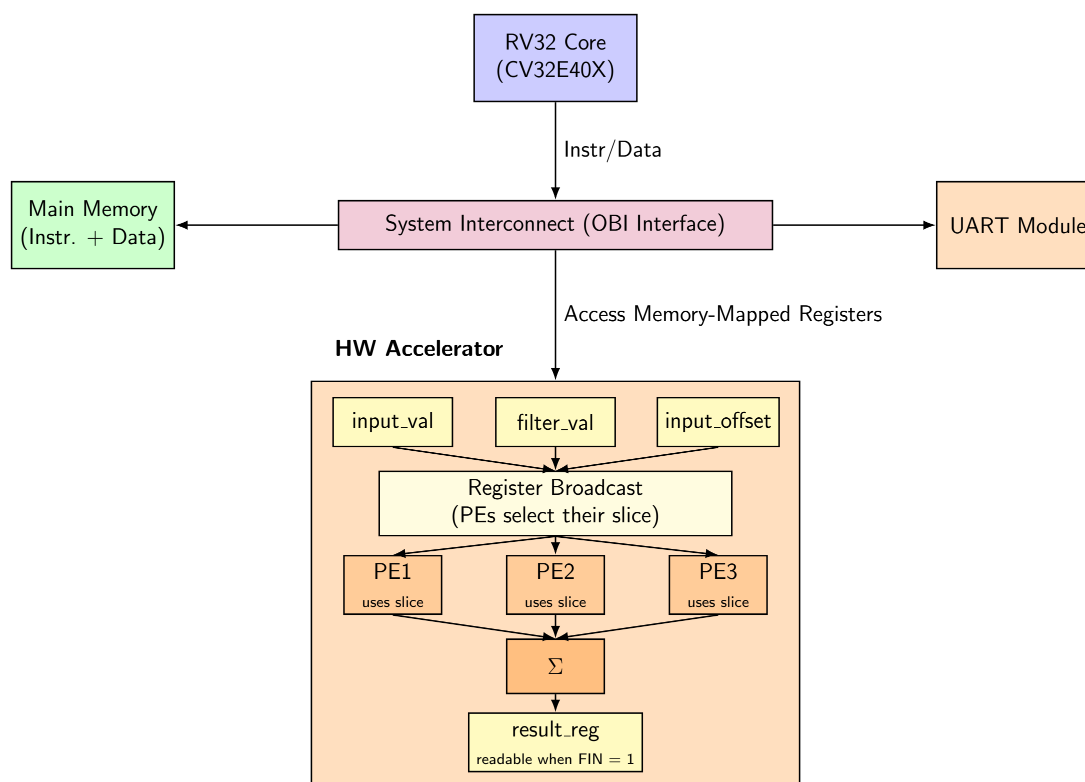

Figure 4.6: Hardware accelerator for 3×3 convolution. Each PE performs one multiplication using memory-mapped registers for the input and kernel values. The results are added in an accumulator.

Upon activating the fin register, as indicated in Figure 4.8, the result can be accessed from the result register and used for the convolution outcome. This synchronization

mechanism facilitates the immediate retrieval of the result from the result register upon completion of the computation.

The subsequent chapter elucidates the implementation of this process on the FPGA and its integration with the camera for the purpose of classifying the captured image.

The SOC comprises a camera, core (cv32e40x), and hardware accelerator. The synthesis was conducted using Vivado. To verify the functionality, classification was performed on the FPGA using the image captured by the camera. The hardware accelerator and custom instruction are employed on the FPGA, while the classification result is generated and transmitted via the UART to the terminal.

4.5.1 Implementation of the Custom Instruction

A noteworthy comparison to the implemented hardware accelerator is the custom instruction, which employs a distinct interface and therefore must be a minor computation to avoid pipeline delays. Unlike the hardware accelerator, the interface is largely predefined, utilizing two source registers and one destination register, with the goal that the computation remains minimal to ensure that the combinational delay is shorter than the longest existing delay in the default ISA. To develop a practical custom instruction, its effect can be simulated using the RISC-V SystemC implementation. Only after this simulation was the custom instruction integrated into the cv32e40x core at the register-transfer level via the custom instruction extension interface provided by the cv32e40x core.

To accommodate the custom instruction, the TLM RISC-V core [33] must be modified to support it. The opcode is incorporated into the opcode table, and the switch statement, which determines the operation to be executed, must be expanded to select the appropriate function for the new custom instruction. These modifications were implemented in the BASE_ISA.h and BASE_ISA.cpp files. These straightforward changes, along with the implementation of the computation performed by the custom instruction, can be executed to obtain feedback on the utility of the instruction.

One essential task is to incorporate the prototype instruction into the RISC-V binary. This can be achieved using the pseudo instruction insn, which is employed in inline assembly within the C source code file. This approach facilitates the generation of custom instructions with designated opcodes and register addresses, allowing the compiler to be utilized without any modifications. This method is particularly advantageous for prototyping an instruction, as it enables the manual addition of the instruction to obtain metrics on the impact of the custom instruction on the overall runtime required for the classification. Once a suitable candidate is identified, the compiler can be extended to map the custom instruction across various locations in the binary file. Listing 4.5 demonstrates the method by which the custom instruction can be incorporated into the binary via inline assembly without necessitating adjustments

to the assembler.

Listing 4.5: Inline assembly used to test the prototyped custom instruction

```
// R-type instruction format:
// opcode = 0b0001011, funct3 = 0, funct7 = 0
asm volatile (
    ".insn r 0b0001011, 0, 0, %[dst], %[rs1], %[rs2]\n\t"
    : [dst] "+r" (result)
    : [rs1] "r"  (input_val),
      [rs2] "r"  (filter_val)
);
```

The custom instruction is employed to expedite multiplication and accumulation operations. The primary strategy involves utilizing a preset input offset, which is added to the input data and subsequently multiplied by the filter value. As illustrated in Figure 4.7, the outcome of the addition and multiplication is incorporated into the accumulation register, and the entire result is stored in this register. The custom instruction used is termed add_mul_acc, and the accumulation register can be reset to zero using the add_mul_acc_reset instruction. The accumulation register retains the currently accumulated convolution sum. This principle is applied to accelerate the forward pass with a concise instruction that can add the input offset to the activation value and multiply it by the filter value while maintaining the accumulated result in the accumulation register. The add_mul_acc_reset instruction employs the funct3 value of the opcode sequence to indicate the reset of the accumulation register, unlike the standard operation. This is also depicted in Figure 4.7, where the feedback of the currently held sum within the accumulation register is only reintroduced if funct3 is 0. The result of the custom instruction, termed add_mul_acc, is written to the destination register within the core via the xif_result interface during the write-back stage.

The manual addition of instructions via inline assembly is the preliminary step to develop an initial prototype and obtain relevant metrics. If the reusability and reduced runtime demonstrate favorable outcomes, extending the compiler may be warranted. The additional effort required for a custom instruction must be justified by its overall utility, given that modifications to the compiler and core itself, even with cv32e40x providing an interface for custom instruction integration, can be substantial. The architectural overview of the custom instruction, termed add_mul_acc, is illustrated in Figure 4.8, which also demonstrates how the XIF issue and result interfaces are employed to extend the ISA. The acc register depicted in Figure 4.8 can be reset to zero using the add_mul_acc_reset instruction.

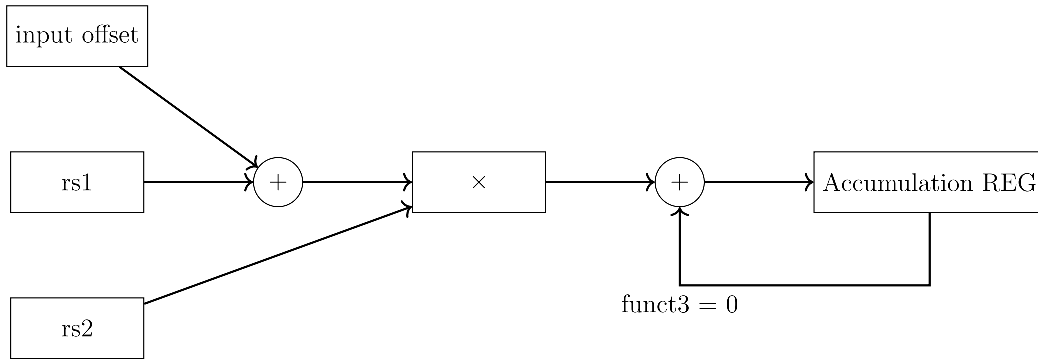

Figure 4.7: The add_mul_acc instruction can add an input offset and then multiply the result by a filter value. It uses an accumulation register to store results until they are reset to zero with the add_mul_acc_reset instruction.

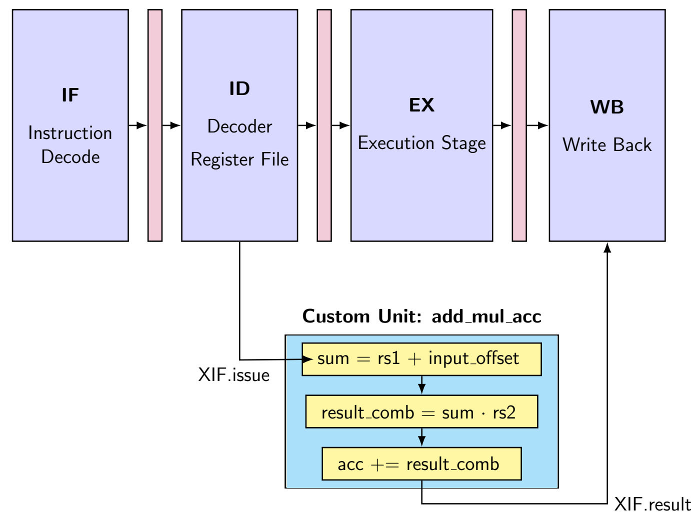

Figure 4.8: CV32E40X pipeline with a custom add_mul_acc instruction integrated via the eXtension Interface (XIF).

The eXtension interface of the cv32e40x facilitates the addition of custom instructions via the RISC-V TLM core to the synthesized core.

The general structure of the eXtension interface for incorporating a custom instruction employs various subinterfaces based on the required functionality. These include the compressed, issue, commit, memory, and result interfaces. The compressed interface is utilized only if the custom instruction is compressed, which is not applicable in this instance. The issue interface features a valid signal that becomes active to indicate that the core has identified a custom instruction and intends to offload it. The opcode and source register values are accessible as soon as the valid signal for the

issue interface is activated. To confirm that the custom instruction logic has processed the information, the ready signal can be set to notify the main CPU. The commit interface indicates whether the instruction has been terminated by the processor and whether it can be canceled. The memory interface facilitates memory access via the load/store unit of the core. The memory valid signal indicates that the offloaded instruction requires memory access, which is acknowledged by the memory ready signal. Ultimately, the result is written back to the core's register, specifically the destination register designated by the opcode. [15]

The procedure involving the eXtension interface of the cv32e40x core facilitates the integration of custom instructions through a predefined interface, thereby eliminating the need to modify the internal hardware description files of the core. Both the core and co-processor, which supply the logic for the custom instruction, are implemented in SystemVerilog. These components undergo cycle-accurate simulation via Verilator prior to synthesis using the Vivado tool. The subsequent chapter addresses the synthesis process and its outcomes.

4.6 Validation of the Synthesized Custom SOC with the cv32e40x as Core

Following the cycle-accurate simulation, which verifies the functionality of the custom instruction and hardware accelerator, the system is validated through synthesis with Vivado. This process involves testing the classification using custom instructions executed on the FPGA. The synthesis facilitates the analysis of the results and the assessment of the combinational delay induced by the designed hardware. The general SOC is depicted in Figure 4.9 as a block diagram featuring the integrated camera. This diagram illustrates the system components used for image classification. The camera, identified as ov7670, uses eight data lines to represent 8-bit pixel values. The camera configuration was achieved via a SCCM, allowing control over the resolution, image format, and automatic lighting adjustments. The camera was set to capture images at a resolution of 160×120 in YUV format, storing only the Y channel in the framebuffer to produce a grayscale image. Additionally, automatic gain control was enabled to facilitate image classification under varying lighting conditions. The classification outcome generated by the cv32e40x core was communicated to the terminal through UART. The FPGA board is connected to the host computer via USB, which converts UART signals to USB signals through the internal UART-USB bridge of the FPGA.

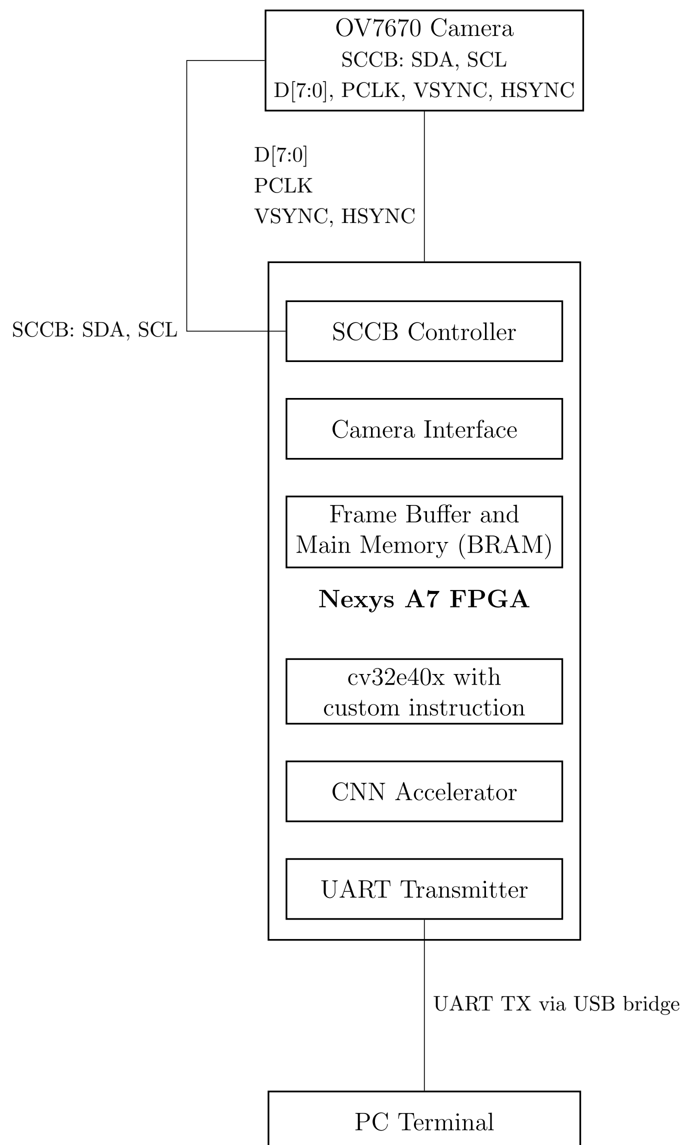

Figure 4.9: Image classification system using the OV7670 camera, SCCB configuration,cv32e40x with custom instruction and the hardware accelerator with UART output bridged via USB.

Figure 4.10 illustrates the Nexys 4 DDR FPGA board and demonstrates the general configuration for camera-based classification. Upon powering the system and loading the bit-stream onto the FPGA, the M17 button must be pressed to activate the camera. Subsequently, the system outputs the classification results via UART. An alternative mode can be selected in the binary to transmit the images via UART as well. The images captured by the OV7670 camera are shown in Figure 4.11.

Figure 4.10: Picture of the finger count classification system.

Figure 4.11: Picture of hand taken by the camera and transmitted via UART. This image is used for classification on the FPGA board.

Vivado was employed to synthesize the entire described system, and the constraint file was modified to accommodate the camera, with all ports mapped to the PMOD connectors. A clock frequency of 50 MHz was utilized for the core, which is essentially the maximum permissible frequency to avoid timing issues during synthesis.

The connection of the PMOD to the camera is illustrated in Figure 4.12. Connectors JA and JB are employed and must be connected to the output pins of the camera, as depicted in Figure 4.12.

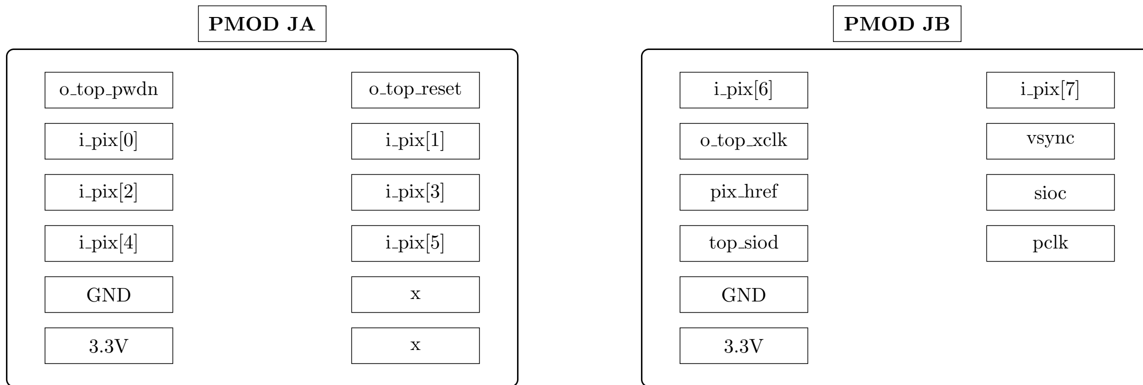

Figure 4.12: Pmod connections called JA and JB with OV7670 camera signals.

5 Results and Evaluation

The subsequent section presents an analysis of the accelerated execution of the forward-pass. Initially, the co-simulation results are examined, followed by an evaluation of the runtime of the classification on the FPGA. The co-simulation serves to provide rapid feedback on functionality and offers an initial estimation of the forward pass runtime. The synthesis process is employed to validate the design and assess whether the designed hardware can be synthesized and routed such that the combinational delay of each path meets the required timing constraints.

5.1 Results Obtained Through Co-Simulation

The hardware accelerator, the custom instruction, and the base version without any specialized architecture, which solely executes the neural network on the default ISA, were simulated, and the results are presented in Figure 5.1. The primary objective was to obtain feedback on the functionality of the custom hardware designed in an expedited manner without necessitating a cycle-accurate simulation of the entire forward pass. Similarly, feedback on the efficacy of the accelerator is acquired, as the instruction count and other metrics, such as the number of memory reads and writes for each version, can be compared and utilized to estimate the cycle count required for one classification with the aid of the CPI value.

Figure 5.1 illustrates that the instructions executed for one classification are reduced from 137 million instructions required with the default ISA to 94 million instructions when employing the hardware accelerator. The co-simulation proves to be a valuable tool, as an estimation of instructions used can be obtained by merely altering the hardware description, which, in this case, is a SystemVerilog file or a VHDL file. The simulation time was approximately 20 s to obtain the co-simulation results. The proposed add_mul_acc instruction reduces the instruction count from 137 million to approximately 99 million. The data memory writes for the custom instruction are higher than those for the default ISA. The default ISA executes approximately 2 million data writes, whereas the custom instruction executes approximately 3.6 million data writes. The final results indicate that fewer instructions are utilized, and the total runtime of the forward pass with the custom instruction is approximately 20% faster compared to the default ISA. The data supporting this can be observed in

Figure 5.2, which presents the post-synthesis results.

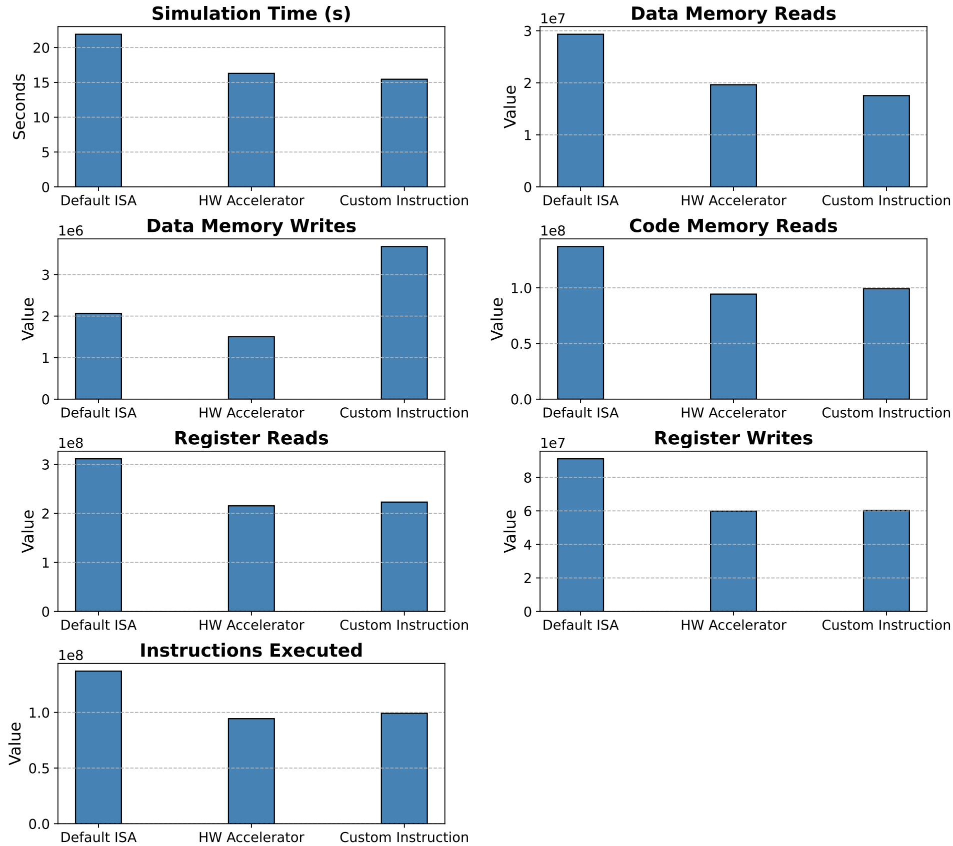

Figure 5.1: Data read and write counts and instructions executed for default ISA, hardware accelerator, and custom instruction are obtained by Co-Simulation.

To provide a comprehensive overview of the results obtained from co-simulating the various implementations, Table 5.1 presents the results. Additional metrics, such as data memory reads and writes, may also be of interest when exploring the design space for an effective hardware accelerator or a custom instruction. The hardware accelerator reduces data memory reads, as fewer instructions are executed, thereby decreasing the number of instructions that need to be fetched from memory, as shown in Table 5.1. Both the hardware accelerator and custom instruction implementation resulted in fewer data memory reads than the default ISA. The hardware accelerator accesses the memory and loads four 8-bit values simultaneously, contributing to the reduction in data memory reads. This improvement is primarily due to the more efficient utilization of the core's internal 32-bit word width compared to the default ISA implementation.

Table 5.1: Performance Comparison of Default ISA, Hardware Accelerator, and Custom Instruction via co-simulation

Metric	Default ISA	HW Accelerator	Custom Instruction
Simulation Time (s)	21.92	16.30	15.15
Data Memory Reads	29,326,549	19,630,520	17,538,871
Data Memory Writes	2,065,887	1,504,504	3,675,029
Code Memory Reads	137,100,598	94,351,793	99,162,634
Code Memory Writes	0	0	0
Register Reads	311,226,393	215,370,357	222,973,074
Register Writes	91,075,862	59,851,613	60,329,031
Instructions Executed	137,100,597	94,351,792	99,162,633

5.2 Post-Synthesis Results

The complete classification system, implemented on the Nexys 4 DDR FPGA utilizing the cv32e40x as a soft core, is employed to classify an image captured by the camera. The duration required for a single classification was determined using a cycle counter, which is accessible from the binary. The execution time was calculated based on the known cycle count used for classification and a clock frequency of 50 MHz. The resulting time required for one classification is output via UART and serves as a metric for evaluating the performance of the design. Listing 5.1 illustrates the method by which the measured time for one forward pass was obtained. This approach facilitates the comparison and analysis of the performance of the modified hardware architecture with that of the default architecture.

```
-----------------------------------------------------------------------
...
Classified as 5

HW accelerator enabled
CPU cycles used for Classification with 50Mhz: 20ns per cycle : 144892181
One classification took 2897 milliseconds
-----------------------------------------------------------------------
```

Listing 5.1: Results obtained using UART from the classification system with the hardware accelerator turned on.

To evaluate the various acceleration methods employed, Figure 5.2 illustrates that the hardware accelerator facilitates a more rapid execution of the forward pass. Specifically, the classification was achieved approximately 1133 ms faster. Consequently, the result is obtained 28 percent more swiftly compared to the execution on the default architecture. This enhanced speed is attributed to the parallel processing of multiple 8-bit activation and filter values. The execution time of the forward pass when

utilizing the enabled custom instruction is approximately 802 ms faster than that of the default architecture. This improvement is achieved through the use of an accumulation register, which can be accessed and reset using custom instructions. The addition of the input offset, multiplication with the filter value, and accumulation of the result in the accumulation register can be accomplished using the two proposed custom instructions. The microarchitecture extended with the custom instruction is approximately 20 percent faster than the default ISA. In contrast, the hardware accelerator is 10 percent faster than the forward pass when employing the custom instruction. Considering the additional effort required to adjust the compiler to support custom instructions, the hardware accelerator may prove more advantageous in this context. Nonetheless, the potential to reuse the custom instruction in other scenarios, where an accumulation register capable of accumulating the results of additions and multiplications is beneficial, might justify the effort of extending the ISA.

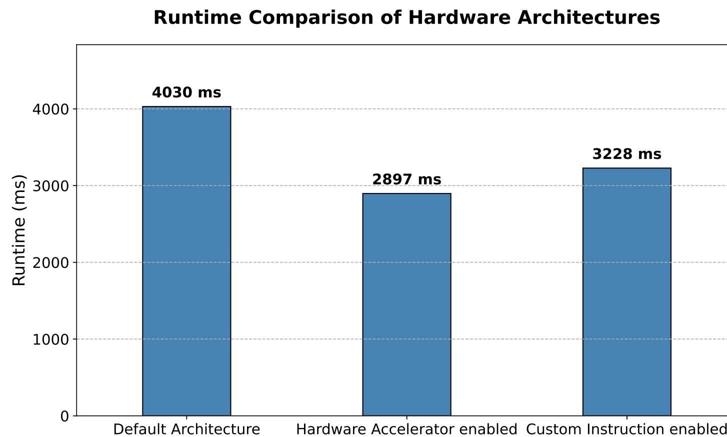

Figure 5.2: Measured times for the forward pass with the default architecture, hardware accelerator on, and custom instruction on.

5.3 Analysis of Area Utilization of the Different Architectures

The physical area occupied by the design was determined using Yosys in conjunction with the open-source sky130 **P**rocess **D**esign **K**it (PDK), which facilitates design synthesis for a 130 nm manufacturing process. The library file sky130_fd_sc_hd__tt_025C_1v80.lib, sourced from the SkyWater PDK repository [18], was used in the synthesis flow. Analyzing these area requirements is crucial for assessing the hardware overhead of the custom instructions in comparison to the default CV32E40X core. Moreover, evaluating the area of the hardware accelerator is imperative because the manufacturing

yield is inversely related to the chip size. Specifically, the cost per chip escalates exponentially with an increase in die area [30], rendering area optimization a primary design constraint. To see the difference between the designed architectures, namely, the hardware accelerator and the extended ISA of the cv32e40x core, Figure 5.3 depicts the occupied areas of each architecture.

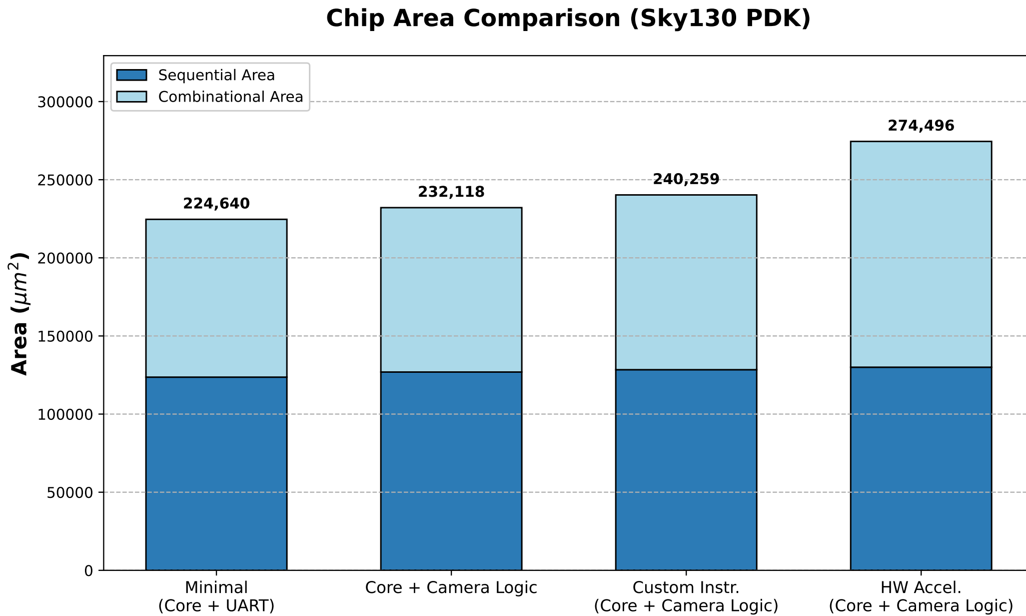

Figure 5.3: Chip size comparison obtained by Yosys for a 130nm process (SKY130) for the differnt architectures designed.

Figure 5.3 illustrates that the custom instruction increases the area utilized by only 3.5% compared to the basic core, in addition to the camera's sequential and combinational logic. This suggests that the custom instructions may serve as a beneficial extension without significantly affecting the area. Consequently, the added custom instructions do not all too negatively influence the yield during production. In contrast, the hardware accelerator increases the chip area by 18%, thereby exerting a more substantial impact on both the yield and cost of the chip. The fundamental camera logic, which facilitates interaction with the camera, its configuration, and the storage of pixels in the framebuffer, increases the size by only 3% compared to the basic core with the interconnect and UART included.

6 Conclusion

Hardware/software co-design strategies offer a means to explore the design space of both hardware architecture and software, facilitating the development of effective solutions for accelerating neural network inference on RISC-V-based systems. The hardware architecture can be designed considering the architecture of the neural network. Additionally, certain design decisions regarding the neural network can be adjusted if an alternative layer structure demonstrates superior performance on a custom-designed hardware architecture. The ability to implement changes at both the software and hardware levels to enhance neural network acceleration is instrumental in identifying a more efficient hardware architecture tailored to the selected neural network structure. The integration of hardware/software co-design methods, which concurrently address both the hardware and software layers, can be leveraged to expedite the neural network inference.

The state-of-the-art section delineates various methodologies for expediting the forward pass of an embedded system employed in the inference of neural networks. Most hardware accelerators discussed in the state-of-the-art chapter utilize PEs, which facilitate the acceleration of different neural network layers. The primary distinction among the currently employed accelerator designs lies in the extent to which different neural network layers benefit from hardware accelerators or custom instructions. Custom instructions that leverage the RISC-V vector unit and incorporate specialized hardware to accelerate neural network layers are advantageous for neural network applications; however, most are not automatically mapped by compilers. This may necessitate additional effort from the developer, but ultimately results in a more rapid execution of the forward pass.

The results presented herein suggest that the hardware accelerator significantly enhances the classification performance and substantially reduces the runtime. In comparing tightly coupled and loosely coupled interfaces, the forward pass was expedited using a custom instruction and hardware accelerator. The findings indicate that the hardware accelerator achieves a performance increase of approximately 28% over the default architecture, whereas the micro-architecture augmented with the custom instruction, termed add_mul_acc, demonstrates a 20% improvement over the default architecture. A notable advantage of the hardware accelerator is that it allows the compiler to remain unchanged, thereby minimizing the effort required for the compiler modification. Conversely, the custom instructions can be employed to accelerate the

various accumulation sections of the binary. Although these sections currently require the manual insertion of the proposed instructions, the fundamental accumulation processes of addition and multiplication can still be effectively reused.

The results presented indicate that the hardware accelerator significantly enhanced the runtime of the forward pass. Specifically, the hardware accelerator reduced the forward pass duration by 1133 ms, decreasing the time required from 4030 to 2897 ms, as illustrated in Figure 5.2. Additionally, the custom instruction contributed to a reduction in the runtime from 4030 to 3228 ms.

Although the hardware accelerator and custom instruction effectively expedite classification, there remains potential for further enhancements. To improve reusability across various neural network architectures, including those with different types of layers, it is essential to refine and adapt the hardware accelerator to accommodate the diverse requirements of other neural network layer structures.

The hardware accelerator increases the chip size by 18 percent, whereas the custom instruction results in a 3 percent increase. This represents an advantage for the proposed custom instructions, as production costs are influenced by the chip size.

Considering the reusability of the custom instruction, it is noted that although it can be manually integrated via inline assembly, its utility across different sections of the binary necessitates automatic mapping by the compiler. The hardware accelerator, as designed, is specifically optimized for accelerating the convolutional layer. However, the interface required for manual mapping by developers may present greater complexity than utilizing inline assembly for custom instructions. Conversely, the hardware accelerator offers superior speed relative to the proposed custom instruction. Therefore, when minimizing runtime is a priority, the hardware accelerator is the preferable choice. For enhanced reusability and a more standardized interface, the custom instructions are more advantageous for accelerating the forward pass. Enhancements to the compiler could facilitate the automatic mapping of the proposed instructions.

6.1 Future Work

As a future objective, it may be beneficial to assess the co-design approach in the context of recurrent neural networks (RNNs) and other layer structures beyond the current focus on convolutional layers. This includes exploring the acceleration of family3 layer types, such as LSTM and fully connected layers.

As a future objective, it is essential to further explore advanced quantization techniques that reduce the model footprint while preserving accuracy. This investigation should build upon a comparison of existing methodologies, such as Post-Training Quantization (PTQ) and Quantization-Aware Training (QAT).

In addition, a critical future goal is to analyze and compare power consumption. It

may also be beneficial to develop analytical models to predict the energy consumption and performance metrics of various architectural configurations.

Acronyms

APB Advanced Peripheral Bus. 27

BRAM Block Random Memory Access. 25

CNN Convolutional Neural Network. 6, 7, 21–23, 30, 31
CPI Cycles per Instruction. 37–39, 50, 52, 63
CPU Central Processing Unit. 24, 29, 30, 35, 54

DRAM Dynamic Random Access Memory. 23, 25

FIFO First-In-First-Out. 18, 19
FPGA Field-Programmable Gate Array. 1–6, 17, 25, 34, 43, 46, 47, 49, 52, 56, 59–61, 63, 65

IMC Basic Integer, Multiplication and Compression operations. 35, 37
ISA Instruction Set Architecture. 2, 17, 25, 26, 28, 29, 56, 57, 63, 64, 66, 67
ISS Instruction Set Simulator. 37–39

LLVM Low Level Virtual Machine. 35
LSTM Long Short-Term Memory. 21–23

MAC Multiply and Accumulate. 21–23, 25, 27
MVM Matrix-Vector Multiplication. 23
MVU Matrix Vector Unit. 28, 29

OS Operating System. 38

PDK Process Design Kit. 66
PE Processing Element. 17–19, 21, 23, 27, 41, 68
PK Primary Kernel. 38
PTQ Post-Training Quantization. 8, 12–14

QAT **Q**uantization-**A**ware **T**raining. 12–14
QNN **Q**uantized **N**eural **N**etwork. 5

RAM **R**andom **A**ccess **M**emory. 11
RCNN **R**ecurrent **C**onvolutional **N**eural **N**etwork. 21
RISC-V **R**educed **I**nstruction **S**et **C**omputer – V. 1–4, 8, 14, 16, 24, 26–30, 32, 34, 35, 37–39, 42, 49, 52, 56, 58, 68

SCCM **S**erial **C**amera **C**ontrol **M**odule. 2, 59
SD-QAT **S**elf-**T**rained-**Q**uantization **A**ware **T**raining. 13
SOC **S**ystem-on-**C**hip. 1, 2, 27, 28, 34, 39, 42, 45–47, 54, 56, 59
SRAM **S**tatic **R**andom **A**ccess **M**emory. 21

TCDM **T**ightly **C**oupled **D**ata **M**emory. 27, 28
TFLM **T**ensorflow **L**ite for **M**icrocontrollers. 1, 34, 35, 38–42, 45–47
TLM **T**ransaction-**L**evel **M**odel. 2, 47, 50, 56, 58
TPU **T**ensor **P**rocessing **U**nit. 1, 4, 15, 18, 20–22, 24, 25
TVM **T**ensor **V**irtual **M**achine. 34, 35, 39

UART **U**niversal **A**synchronous **R**eceiver/**T**ransmitter. 2, 3, 37–39, 46, 56, 59–61, 65
USB **U**niversal **S**erial **B**us. 3, 59, 60

VTA **V**ersatile **T**ensor **A**ccelerator. 35

Bibliography

[1] Ahmed J. Abdelmaksoud, Shady Agwa, and Themis Prodromakis. "DiP: A Scalable, Energy-Efficient Systolic Array for Matrix Multiplication Acceleration." In: *IEEE Transactions on Circuits and Systems I: Regular Papers* (2025), pp. 1–11. ISSN: 1549-8328, 1558-0806. DOI: `10.1109/TCSI.2025.3591960`. arXiv: `2412.09709 [cs]`. URL: `http://arxiv.org/abs/2412.09709` (visited on 12/01/2025) (cit. on pp. 18–20).

[2] *About RISC-V*. RISC-V International. URL: `https://riscv.org/about/` (visited on 01/08/2026) (cit. on p. 1).

[3] Ryan Albert Antonio et al. *An Open-Source HW-SW Co-Development Framework Enabling Efficient Multi-Accelerator Systems*. Aug. 20, 2025. DOI: `10.48550/arXiv.2508.14582`. arXiv: `2508.14582 [cs]`. URL: `http://arxiv.org/abs/2508.14582` (visited on 09/03/2025). Pre-published (cit. on pp. 29, 30).

[4] *Apache TVM*. tvm.apache.org archived with webarchive. URL: `https://web.archive.org/web/20250621040655/https://tvm.apache.org/` (visited on 06/26/2025) (cit. on p. 35).

[5] Yelysei Bondarenko, Markus Nagel, and Tijmen Blankevoort. *Understanding and Overcoming the Challenges of Efficient Transformer Quantization*. Sept. 27, 2021. DOI: `10.48550/arXiv.2109.12948`. arXiv: `2109.12948 [cs]`. URL: `http://arxiv.org/abs/2109.12948` (visited on 08/25/2025). Pre-published (cit. on p. 12).

[6] Amirali Boroumand et al. *Google Neural Network Models for Edge Devices: Analyzing and Mitigating Machine Learning Inference Bottlenecks*. Sept. 29, 2021. DOI: `10.48550/arXiv.2109.14320`. arXiv: `2109.14320 [cs]`. URL: `http://arxiv.org/abs/2109.14320` (visited on 12/02/2025). Pre-published (cit. on pp. 1, 21–24).

[7] Oliver Bringmann et al. "Automated HW/SW Co-Design for Edge AI: State, Challenges and Steps Ahead." In: *Proceedings of the 2021 International Conference on Hardware/Software Codesign and System Synthesis*. CODES/ISSS '21: 2021 International Conference on Hardware/Software Codesign and System Synthesis. Virtual Event: ACM, Sept. 30, 2021, pp. 11–20. ISBN: 978-1-4503-9076-7.

DOI: 10.1145/3478684.3479261. URL: https://dl.acm.org/doi/10.1145/3478684.3479261 (visited on 06/17/2025) (cit. on p. 5).

[8] Oliver Bringmann et al. "Automated HW/SW Co-design for Edge AI: State, Challenges and Steps Ahead: Special Session Paper." In: *2021 International Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS)*. 2021 International Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS). Oct. 2021, pp. 11–20. URL: https://ieeexplore.ieee.org/document/9603364 (visited on 08/25/2025) (cit. on pp. 25, 26).

[9] Tom Brown et al. "Language Models Are Few-Shot Learners." In: *Advances in Neural Information Processing Systems*. Vol. 33. Curran Associates, Inc., 2020, pp. 1877–1901. URL: https://proceedings.neurips.cc/paper/2020/hash/1457c0d6bfcb4967418bfb8ac142f64a-Abstract.html (visited on 08/25/2025) (cit. on p. 6).

[10] Tianqi Chen et al. *TVM: An Automated End-to-End Optimizing Compiler for Deep Learning*. Oct. 5, 2018. DOI: 10.48550/arXiv.1802.04799. arXiv: 1802.04799 [cs]. URL: http://arxiv.org/abs/1802.04799 (visited on 06/26/2025). Pre-published (cit. on p. 34).

[11] Hongrong Cheng, Miao Zhang, and Javen Qinfeng Shi. *A Survey on Deep Neural Network Pruning-Taxonomy, Comparison, Analysis, and Recommendations*. Aug. 9, 2024. DOI: 10.48550/arXiv.2308.06767. arXiv: 2308.06767 [cs]. URL: http://arxiv.org/abs/2308.06767 (visited on 08/25/2025). Pre-published (cit. on p. 26).

[12] *CORE-V CV32E40X User Manual*. v0.6.0. Accessed: 2025-11-24. OpenHW Group. Switzerland, 2020. URL: https://docs.openhwgroup.org/_/downloads/cv32e40x-user-manual/en/0.6.0/pdf/ (cit. on p. 38).

[13] Enfang Cui, Tianzheng Li, and Qian Wei. "RISC-V Instruction Set Architecture Extensions: A Survey." In: *IEEE Access* 11 (2023), pp. 24696–24711. ISSN: 2169-3536. DOI: 10.1109/ACCESS.2023.3246491. URL: https://ieeexplore.ieee.org/document/10049118/ (visited on 06/18/2025) (cit. on p. 26).

[14] Théo Dupuis et al. *Sparq: A Custom RISC-V Vector Processor for Efficient Sub-Byte Quantized Inference*. June 16, 2023. DOI: 10.48550/arXiv.2306.09905. arXiv: 2306.09905 [cs]. URL: http://arxiv.org/abs/2306.09905 (visited on 01/15/2026). Pre-published (cit. on pp. 32, 33).

[15] *eXtension Interface — CORE-V CV32E40X User Manual Documentation*. URL: https://docs.openhwgroup.org/projects/cv32e40x-user-manual/en/latest/x_ext.html (visited on 11/19/2025) (cit. on p. 59).

[16] Li Gao et al. "Convolutional Neural Network Acceleration Techniques Based on FPGA Platforms: Principles, Methods, and Challenges." In: *Information* 16.10 (Oct. 18, 2025). ISSN: 2078-2489. DOI: 10.3390/info16100914. URL: https://www.mdpi.com/2078-2489/16/10/914 (visited on 12/01/2025) (cit. on p. 18).

[17] Pierre Garreau et al. "A Survey on Versatile Embedded Machine Learning Hardware Acceleration." In: *Journal of Systems Architecture* 167 (Oct. 1, 2025), p. 103501. ISSN: 1383-7621. DOI: 10.1016/j.sysarc.2025.103501. URL: https://www.sciencedirect.com/science/article/pii/S1383762125001730 (visited on 01/12/2026) (cit. on pp. 16, 17, 27–29).

[18] *Google/Skywater-Pdk*. Google, Feb. 22, 2026. URL: https://github.com/google/skywater-pdk (visited on 02/22/2026) (cit. on p. 66).

[19] Jahid Hasan. *Optimizing Large Language Models through Quantization: A Comparative Analysis of PTQ and QAT Techniques*. Nov. 9, 2024. DOI: 10.48550/arXiv.2411.06084. arXiv: 2411.06084 [cs]. URL: http://arxiv.org/abs/2411.06084 (visited on 11/19/2025). Pre-published (cit. on p. 14).

[20] John L. Hennessy and David A. Patterson. *Computer Architecture: A Quantitative Approach*. 6th. Burlington, MA: Morgan Kaufmann, 2019. ISBN: 978-0128119051 (cit. on p. 37).

[21] Mark Horowitz. "1.1 Computing's Energy Problem (and What We Can Do about It)." In: *2014 IEEE International Solid-State Circuits Conference Digest of Technical Papers (ISSCC)*. 2014 IEEE International Solid-State Circuits Conference Digest of Technical Papers (ISSCC). Feb. 2014, pp. 10–14. DOI: 10.1109/ISSCC.2014.6757323. URL: https://ieeexplore.ieee.org/document/6757323 (visited on 08/04/2025) (cit. on p. 10).

[22] Katsuya Hyodo. *Tools to Convert ONNX Files (NCHW) to TensorFlow Format (NHWC)*. Sept. 2022. URL: https://github.com/PINTO0309/onnx2tf (visited on 12/05/2025) (cit. on p. 43).

[23] Benoit Jacob et al. *Quantization and Training of Neural Networks for Efficient Integer-Arithmetic-Only Inference*. Dec. 15, 2017. DOI: 10.48550/arXiv.1712.05877. arXiv: 1712.05877 [cs]. URL: http://arxiv.org/abs/1712.05877 (visited on 11/17/2025). Pre-published (cit. on p. 8).

[24] Benoit Jacob et al. *Quantization and Training of Neural Networks for Efficient Integer-Arithmetic-Only Inference*. Dec. 15, 2017. DOI: 10.48550/arXiv.1712.05877. arXiv: 1712.05877 [cs]. URL: http://arxiv.org/abs/1712.05877 (visited on 08/04/2025). Pre-published (cit. on p. 9).

[25] Benoit Jacob et al. *Quantization and Training of Neural Networks for Efficient Integer-Arithmetic-Only Inference*. Dec. 15, 2017. DOI: 10.48550/arXiv.1712.05877. arXiv: 1712.05877 [cs]. URL: http://arxiv.org/abs/1712.05877 (visited on 07/21/2025). Pre-published (cit. on p. 13).

[26] Vikram Jain et al. "TinyVers: A Tiny Versatile System-on-Chip With State-Retentive eMRAM for ML Inference at the Extreme Edge." In: *IEEE Journal of Solid-State Circuits* 58.8 (Aug. 2023), pp. 2360–2371. ISSN: 1558-173X. DOI: 10.1109/JSSC.2023.3236566. URL: https://ieeexplore.ieee.org/document/10022047 (visited on 01/14/2026) (cit. on pp. 27, 28).

[27] Norman P. Jouppi et al. *In-Datacenter Performance Analysis of a Tensor Processing Unit*. Apr. 16, 2017. DOI: 10.48550/arXiv.1704.04760. arXiv: 1704.04760 [cs]. URL: http://arxiv.org/abs/1704.04760 (visited on 12/01/2025). Pre-published (cit. on p. 18).

[28] Norman P. Jouppi et al. *In-Datacenter Performance Analysis of a Tensor Processing Unit*. Apr. 16, 2017. DOI: 10.48550/arXiv.1704.04760. arXiv: 1704.04760 [cs]. URL: http://arxiv.org/abs/1704.04760 (visited on 08/04/2025). Pre-published (cit. on p. 21).

[29] Raghuraman Krishnamoorthi. *Quantizing Deep Convolutional Networks for Efficient Inference: A Whitepaper*. June 21, 2018. DOI: 10.48550/arXiv.1806.08342. arXiv: 1806.08342 [cs]. URL: http://arxiv.org/abs/1806.08342 (visited on 11/19/2025). Pre-published (cit. on p. 11).

[30] N. Kumar et al. "A Review of Yield Modelling Techniques for Semiconductor Manufacturing." In: *International Journal of Production Research* 44.23 (Dec. 1, 2006), pp. 5019–5036. ISSN: 0020-7543. DOI: 10.1080/00207540600596874. URL: https://doi.org/10.1080/00207540600596874 (visited on 02/22/2026) (cit. on p. 67).

[31] Qianlin Liang, Prashant Shenoy, and David Irwin. *AI on the Edge: Rethinking AI-based IoT Applications Using Specialized Edge Architectures*. Mar. 27, 2020. DOI: 10.48550/arXiv.2003.12488. arXiv: 2003.12488 [cs]. URL: http://arxiv.org/abs/2003.12488 (visited on 12/04/2025). Pre-published (cit. on p. 5).

[32] Qiankun Liu and Sam Amiri. "Optimised Extension of an Ultra-Low-Power RISC-V Processor to Support Lightweight Neural Network Models." In: *Chips* 4.2 (2 June 2025), p. 13. ISSN: 2674-0729. DOI: 10.3390/chips4020013. URL: https://www.mdpi.com/2674-0729/4/2/13 (visited on 06/16/2025) (cit. on p. 1).

[33] Màrius Montón. *Mariusmm/RISC-V-TLM*. Dec. 4, 2025. URL: https://github.com/mariusmm/RISC-V-TLM (visited on 12/05/2025) (cit. on pp. 2, 49, 56).

[34] Bert Moons et al. *Minimum Energy Quantized Neural Networks*. Nov. 23, 2017. DOI: 10.48550/arXiv.1711.00215. arXiv: 1711.00215 [cs]. URL: http://arxiv.org/abs/1711.00215 (visited on 08/04/2025). Pre-published (cit. on p. 11).

[35] Anushree Nagvekar. "Edge AI: Revolutionizing Embedded Systems through On-Device Processing." In: *International Journal of Scientific Research in Computer Science, Engineering and Information Technology* 11.1 (Feb. 18, 2025), pp. 2871–2880. ISSN: 2456-3307. DOI: 10.32628/CSEIT251112289. URL: https://ijsrcseit.com/index.php/home/article/view/CSEIT251112289 (visited on 12/04/2025) (cit. on p. 4).

[36] *Netron Web App*. URL: https://netron.app/ (visited on 11/20/2025) (cit. on p. 37).

[37] Adam Paszke et al. "PyTorch: An Imperative Style, High-Performance Deep Learning Library." In: *Advances in Neural Information Processing Systems*. Vol. 32. Curran Associates, Inc., 2019. URL: https://proceedings.neurips.cc/paper_files/paper/2019/hash/bdbca288fee7f92f2bfa9f7012727740-Abstract.html (visited on 09/15/2025) (cit. on pp. 1, 44).

[38] *Picorv32/Picosoc/Simpleuart.v at Main · YosysHQ/Picorv32 · GitHub*. URL: https://github.com/YosysHQ/picorv32/blob/main/picosoc/simpleuart.v (visited on 11/26/2025) (cit. on p. 46).

[39] *Riscv-Collab/Riscv-Gnu-Toolchain*. RISC-V Collaboration, Nov. 24, 2025. URL: https://github.com/riscv-collab/riscv-gnu-toolchain (visited on 11/24/2025) (cit. on pp. 38, 39).

[40] *Riscv-Software-Src/Riscv-Pk*. RISC-V Software, Nov. 22, 2025. URL: https://github.com/riscv-software-src/riscv-pk (visited on 11/24/2025) (cit. on p. 38).

[41] Jacob Sander et al. *On Accelerating Edge AI: Optimizing Resource-Constrained Environments*. Jan. 28, 2025. DOI: 10.48550/arXiv.2501.15014. arXiv: 2501.15014 [cs]. URL: http://arxiv.org/abs/2501.15014 (visited on 11/19/2025). Pre-published (cit. on p. 14).

[42] Kiran Seshadri et al. "An Evaluation of Edge TPU Accelerators for Convolutional Neural Networks." In: *2022 IEEE International Symposium on Workload Characterization (IISWC)*. 2022 IEEE International Symposium on Workload Characterization (IISWC). Nov. 2022, pp. 79–91. DOI: 10.1109/IISWC55918.2022.00017. URL: https://ieeexplore.ieee.org/abstract/document/9975395 (visited on 12/02/2025) (cit. on p. 1).

[43] Wilson Snyder et al. *Verilator*. Dec. 5, 2025. URL: https://github.com/verilator/verilator (visited on 12/05/2025) (cit. on p. 50).

[44] Sarala K Surapally et al. "Evaluating FPGA Acceleration on Binarized Neural Networks and Quantized Neural Networks." In: *2022 International Symposium on Measurement and Control in Robotics (ISMCR)*. 2022 International Symposium on Measurement and Control in Robotics (ISMCR). Sept. 2022, pp. 1–5. DOI: 10.1109/ISMCR56534.2022.9950581. URL: https://ieeexplore.ieee.org/document/9950581 (visited on 06/18/2025) (cit. on p. 5).

[45] Guner Tatar, Salih Bayar, and Ihsan Cicek. "Performance Evaluation of Low-Precision Quantized LeNet and ConvNet Neural Networks." In: *2022 International Conference on INnovations in Intelligent SysTems and Applications (INISTA)*. 2022 International Conference on INnovations in Intelligent SysTems and Applications (INISTA). Aug. 2022, pp. 1–6. DOI: 10.1109/INISTA55318.2022.9894261. URL: https://ieeexplore.ieee.org/document/9894261 (visited on 07/07/2025) (cit. on p. 8).

[46] *Tensorflow/Tflite-Micro*. tensorflow, Nov. 24, 2025. URL: https://github.com/tensorflow/tflite-micro (visited on 11/24/2025) (cit. on pp. 1, 34, 35, 38, 41, 45).

[47] Karthik Wali. "Hardware-Software Co-Design for Power-Efficient Edge-AI Systems." In: *Journal of Artificial Intelligence, Machine Learning and Data Science* 2.4 (Oct. 30, 2024), pp. 2754–2761. ISSN: 25839888. DOI: 10.51219/JAIMLD/karthik-wali/580. URL: https://urfjournals.org/open-access/hardware-software-co-design-for-power-efficient-edge-ai-systems.pdf (visited on 12/04/2025) (cit. on pp. 15, 16).

[48] Hang Wang et al. "ST-QAT: Leveraging Self-Training to Enhance Quantization-Aware Training." In: *2025 International Joint Conference on Neural Networks (IJCNN)*. 2025 International Joint Conference on Neural Networks (IJCNN). June 2025, pp. 1–8. DOI: 10.1109/IJCNN64981.2025.11227682. URL: https://ieeexplore.ieee.org/document/11227682/ (visited on 11/18/2025) (cit. on p. 14).

[49] Chen Wu et al. "Low-Precision Floating-point Arithmetic for High-performance FPGA-based CNN Acceleration." In: *ACM Transactions on Reconfigurable Technology and Systems* 15.1 (Mar. 31, 2022), pp. 1–21. ISSN: 1936-7406, 1936-7414. DOI: 10.1145/3474597. URL: https://dl.acm.org/doi/10.1145/3474597 (visited on 07/07/2025) (cit. on p. 7).

[50] Jiaxiang Wu et al. *Quantized Convolutional Neural Networks for Mobile Devices.* May 16, 2016. DOI: 10.48550/arXiv.1512.06473. arXiv: 1512.06473 [cs]. URL: http://arxiv.org/abs/1512.06473 (visited on 08/04/2025). Pre-published (cit. on p. 12).

[51] Guangxuan Xiao et al. *SmoothQuant: Accurate and Efficient Post-Training Quantization for Large Language Models.* Mar. 29, 2024. DOI: 10.48550/arXiv.2211.10438. arXiv: 2211.10438 [cs]. URL: http://arxiv.org/abs/2211.10438 (visited on 08/25/2025). Pre-published (cit. on p. 6).

[52] Priyanshu Yadav. *Design and Implementation of a RISC-V SoC with Custom DSP Accelerators for Edge Computing.* June 7, 2025. DOI: 10.48550/arXiv.2506.06693. arXiv: 2506.06693 [cs]. URL: http://arxiv.org/abs/2506.06693 (visited on 11/29/2025). Pre-published (cit. on pp. 1, 17).

[53] Muhammed Yildirim and Ozcan Ozturk. *RISC-V Based TinyML Accelerator for Depthwise Separable Convolutions in Edge AI.* Nov. 26, 2025. DOI: 10.48550/arXiv.2511.21232. arXiv: 2511.21232 [cs]. URL: http://arxiv.org/abs/2511.21232 (visited on 11/29/2025). Pre-published (cit. on p. 1).

[54] *YosysHQ/Yosys.* Yosys Headquarters, Dec. 5, 2025. URL: https://github.com/YosysHQ/yosys (visited on 12/05/2025) (cit. on p. 49).

[55] Douglas Youvan. *Parallel Precision: The Role of GPUs in the Acceleration of Artificial Intelligence.* Nov. 2023. DOI: 10.13140/RG.2.2.21937.76641 (cit. on p. 25).

[56] Xiang Yu et al. "CNN Specific ISA Extensions Based on RISC-V Processors." In: *2022 5th International Conference on Circuits, Systems and Simulation (ICCSS).* 2022 5th International Conference on Circuits, Systems and Simulation (ICCSS). May 2022, pp. 116–120. DOI: 10.1109/ICCSS55260.2022.9802445. URL: https://ieeexplore.ieee.org/document/9802445 (visited on 01/14/2026) (cit. on pp. 30, 31).

[57] Xia Zhao et al. "A Review of Convolutional Neural Networks in Computer Vision." In: *Artificial Intelligence Review* 57 (Mar. 23, 2024), pp. 57–99. DOI: 10.1007/s10462-024-10721-6 (cit. on p. 36).