Project:	ISOLDE: Customizable Instruction Sets and Open Leveraged Designs of Embedded RISCV Processors
Reference number:	101112274
Project duration:	01.05.2023 - 30.04.2026
Work Package:	WP3: Accelerators and Extensions
Deliverable	**D3.3**
Title	**Accelerators Prototype Implementation**
Type of deliverable	Report
Deadline	30.04.2025
Creation Date	31.01.2025
Dissemination Level	ISOLDE - public

Authors
F. Conti, L. Ghionda - UNIBO (lead)
M. Gautschi, M. Korb, S. Lippuner - ACP
J. Abella, X. Carril, S. Alcaide, F. J. Cazorla, R. Canal - BSC
M.A. Sachian, G. Suciu - BEIA
J. Kaštil - CODA
M. Perotti - ETHZ
M. Munteanu, H. Galmeanu - FotoNation
A. Pușcașu, C. B. Ciobanu, M. Gologanu, R. I. Stancu - IMT
A.Galimberti, D.Zoni, W.Fornaciari - POLIMI
T. Terzano, F. Guella, M. Martina, G. Urgese - POLITO
A. Suresh, D. Gigena-Ivanovich - SAL
A. Stan - TUI
A. Sharma Poudel, M. Berekovic, R. Buchty - UzL

Involved grant recipients:
Alma Mater Studiorum - Università di Bologna (UNIBO (lead))
ACP Advanced Circuit Pursuit AG (ACP)
Barcelona Supercomputing Center (BSC)
BEIA Consult International SRL (BEIA)
Codasip SRO (CODA)
Eidgenössische Technische Hochschule Zürich (ETHZ)
Fotonation SRL (FotoNation)
Institutul National de Cercetare-Dezvoltare Pentru Microtehnologie (IMT)
Politecnico di Milano (POLIMI)

Politecnico di Torino (POLITO)
Silicon Austria Labs GmbH (SAL)
Technical University of Iasi (TUI)
Universität zu Lübeck (UzL)

Contacts: Francesco Conti, UNIBO, f.conti@unibo.it
Luigi Ghionda, UNIBO, luigi.ghionda2@unibo.it

Reviewers: Sylvain Girbal, TRT, sylvain.girbal@thalesgroup.com
Jan Andersson, GSL, jan@gaisler.com

Table of Contents

1 Executive Summary

Deliverable D3.3 presents the prototype implementations of hardware accelerators developed within Work Package 3 (WP3) of the ISOLDE project. These components target key computational domains—including AI/ML, cryptography, signal processing, and virtualization—supporting the project's goal of building customizable and efficient RISC-V-based platforms for embedded applications.

The deliverable includes over 20 synthesizable and evaluable IPs, organized across the following categories:

- Arithmetic infrastructure (e.g., mixed-precision FPUs, scratchpad memories);
- SIMD, vector, tensor, and AI accelerators;
- Post-quantum and classical cryptographic cores;
- Signal processing, neuromorphic units, and ASIPs.

Each IP is described via a standard template ("IP Card") detailing functionality, architecture, integration interface, maturity level, and evaluation results (e.g., FPGA/ASIC synthesis, performance, power).

This deliverable represents a transition from design to validated prototype, supporting downstream integration with WP5. It forms the technical baseline for final implementation deliverables D3.4 and D3.5.

2 Introduction

2.1 General Information

Work Package 3 (WP3) of the ISOLDE project focuses on the development and prototyping of hardware modules and architectural extensions designed to enhance the performance and capabilities of RISC-V systems. These developments build upon the foundational cores provided by WP2 and are intended to support a wide range of application domains, including AI/ML, cryptography, signal processing, and embedded control.

This deliverable, **D3.3 – Accelerators Prototype Implementation** transitions from architectural design to concrete prototype implementations. The document gathers the results and contributions of all WP3 partners and tasks, with each extension described in terms of its integration, synthesis readiness, and evaluation metrics such as area, performance, and power.

Each section of the deliverable is organized by task and domain. The architecture and performance evaluations presented here serve both as a record of current progress and as a foundation for final implementation activities.

2.2 Purpose and Scope

The purpose of this deliverable is to present the prototype implementations of the hardware accelerators and extensions developed under WP3. It documents the technical evolution from initial architectural concepts to functional and synthesizable hardware IP blocks, many of which have been evaluated through simulation or FPGA-based prototyping.

This document provides detailed descriptions of each prototype, including:

- Functional goals and supported operations;
- Microarchitectural and integration details;
- ISA extensions, interface protocols, and programmability aspects;
- Evaluation metrics (e.g., area, frequency, power, latency);
- Maturity level and readiness for integration into demonstrator platforms.

This deliverable acts as a reference for both internal and external stakeholders, facilitating the integration of WP3 outcomes into the ISOLDE use-cases. Each IP includes an *IP Card* to summarize all important details about the IP.

The contents of this report lay the groundwork for final implementation deliverables (D3.4 and D3.5 – due in Month 33), which will further refine and validate the accelerators presented herein.

3 Accelerators and Extensions

3.1 Accelerator Infrastructure, Memories, Arithmetic Units, Interfaces and Virtualization

This section presents a set of hardware components that serve as foundational building blocks for accelerator integration in RISC-V-based systems, developed as part of the activity of **Task 3.2 – Accelerator infrastructure, memories, arithmetic units, interfaces and virtualization** active from M3 to M33 and led by **UzL**. These include specialized floating-point units, configurable scratchpad memories, and associated interface logic. The IPs described in this section enable efficient arithmetic computation, flexible memory access, and support for virtualization and integration into larger SoC designs.

The following IP blocks are included in this category:

- **FPMIX – FPU for Mixed-Precision Computing (POLIMI)**: A flexible floating-point unit supporting configurable precision at design time, optimized for energy and area efficiency in mixed-precision workloads.
- **FPU – Floating-Point Unit for RISC-V (UzL)**: An open-source IEEE-754-compliant FPU supporting multiple precision levels, fault-injection analysis, and integration into RISC-V processor cores such as CVA6.
- **Scratchpad Memory (IMT)**: A banked, customizable memory architecture based on PolyMem, supporting advanced memory access schemes (e.g., rectangular, diagonal, transposed) for high-throughput data operations in compute accelerators.

These components form the basis for arithmetic acceleration, memory-efficient data processing, and extensibility of the ISOLDE computing platform.

3.1.1 FPU for Mixed-Precision Computing (FPMIX) – POLIMI

3.1.1.1 IP Card

Basic Info		
IP name	FPU for Mixed-Precision Computing (FPMIX)	
License	Open-source (bfloat16 instances), proprietary closed-source (other instances)	
Repository	https://github.com/HEAPLab/FPMIX-ISOLDE	
Architecture		
Clock	Number of clock domains Synchronous with system Clock generated internally	1 Y N
Ctrl Interface	ISA extension? Memory mapped? Protocol Address Map	N N
Interrupts	Interrupts	N
Microarchitecture		
Parametrization	Parametric no. units? Parameteric config?	N Y (FP format of each operation type)
Programmability	Contains programmable cores? ISA	N N.A.
Software		
Compiler	Requires specialized compiler? Compiler repository	N
Hardware Abstraction Layer	N/A	
High-level API	Is there a high-level API/SDK? SDK repository Is there a domain-specific compiler?	N N
Integration		
IP Distribution	Manifest type (if any) Standalone simulation? (if standalone sim) SW requirements? Integration documented / examples?	 Y AMD Vivado
Synthesis	Is the IP synthesizable? FPGA synthesis scripts/example available? ASIC synthesis scripts/example available?	Y Y N
Simulation	Closed-source simulation? Open-source simulation?	Y (AMD Vivado) N
Evaluation	PPA results available?	Y

3.1.1.2 Purpose

Our floating-point unit (FPU) implements operations with configurable amounts of precision bits in their floating-point (FP) arithmetic formats for the operands and result.

The FPMIX FPU is meant to be used in mixed-precision computing scenarios, which can fully make use of the flexibility in the precision provided by the FPU to achieve different tradeoffs between accuracy, latency, energy consumption, and area. The formats of FP operations in FPMIX can be configured at design time, also by leveraging precision tuning approaches.

3.1.1.3 Architecture

The architecture of the FPU includes internal components meant to support the FP addition/subtraction, multiplication and division operations, comparisons, and conversions between integer and FP formats. The FPU implements floating-point operations whose precision (number of mantissa bits) can be configured at design time. The precision for each type of operation is independently configurable at design time, i.e., different floating-point operations can have different precision, while the dynamic range (number of exponent bits) is fixed and the same as the widely used IEEE-754 float32 one for all the supported floating-point formats. In general, each category of operation in

the FPU, namely, additions/subtractions, multiplications, divisions, comparisons, and conversions, can indeed implement a floating-point format with a different number of mantissa bits ranging between 1 and 23, 8 exponent bits, and 1 sign bit.

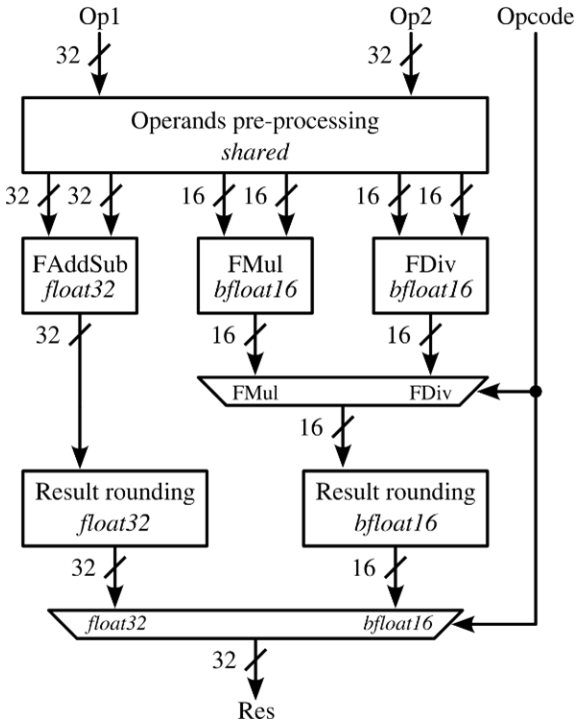

Figure 3.1: Architecture of an FPMIX instance with float32 additions/subtractions and bfloat16 multiplications and divisions.

For example, the block diagram depicted in Figure 3.1 refers to an FPU configuration with float32 additions/subtractions and bfloat16 multiplications and divisions. The float32 format has a 1-bit sign, an 8-bit exponent, and a 23-bit mantissa, whereas the bfloat16 one has a 1-bit sign, an 8-bit exponent, and a 7-bit mantissa. Values received as inputs and produced as outputs by the FPU are always encoded in the 32-bit float32 representation (in addition to 32-bit integers, in the case of float-integer conversions). Operands to lower-precision operations are truncated by discarding the corresponding number of least significant bits of the mantissa, while their results are conversely extended by padding the least significant part of the mantissa with zeros. Dedicated rounding logic is instantiated for each floating-point format used by at least one operation in the FPU. Instantiating FP operations with lower-precision formats can reduce the area occupation, power consumption, and latency of the corresponding hardware logic, improving the energy efficiency of the computing platform. Software precision tuning techniques can aid in exploring trade-offs that consider the acceptable accuracy loss for the target workloads.

Interface FPMIX can be integrated as the functional unit of a RISC-V CPU core replacing any existing FPU. Its interface consists of two 32-bit operands and an opcode as its inputs and a 32-bit

result as its output, in addition to valid and acknowledge 1-bit flags.

The compatibility with the IEEE-754 FP format, albeit possibly with reduced precision, makes it possible to integrate FPMIX into a CPU that implements the standard RISC-V F extension for single-precision FP arithmetic, i.e., FPMIX does not require any custom instruction or extension.

3.1.1.4 Evaluation

Prototype implementations of the FPMIX unit have been instantiated with support for the standard 32-bit single-precision IEEE 754 FP format as well as for the 16-bit bfloat16 one. The two formats share the 1-bit sign and the 8-bit exponent, and the former has a 23-bit mantissa while the latter has a 7-bit one.

Instances of the FPMIX functional unit were synthesized and implemented, leveraging the AMD Vivado toolchain, targeting AMD FPGA chips and an operating clock frequency of 50MHz. Simulations carried out in AMD Vivado allow evaluating the functional correctness of the FPU as well as its performance-accuracy trade-offs when making use of different FP formats for the various arithmetic operations.

Workloads used to evaluate FPMIX include benchmark applications from the PolyBench/C suite, that make wide usage of floating-point computations.

The resource utilization ranges from 1948 LUTs, 632 FFs, and 4 DSPs for an FPMIX instance with all operations supporting the 32-bit IEEE 754 single-precision FP format to 1119 LUTs, 425 FFs, and 1 DSP for an instance with all operations supporting the 16-bit bfloat16 FP format. These resource utilization results are collected after implementation on an AMD Artix-7 100 FPGA.

Both instances are configured to take, when integrated into a RISC-V CPU, 5 clock cycles for additions/subtractions and multiplications and 4 clock cycles for conversions and comparisons, while they differ in divisions which take 12 clock cycles on the float32 FPMIX configuration and 9 clock cycles on the bfloat16 one.

In addition to saving 42% LUT resources compared to the float32 FPMIX instance, using bfloat16 instance also results in a 4% reduction in terms of energy-delay product on average when executing a set of applications from the PolyBench/C benchmark suite.

3.1.2 Floating-Point Unit for RISC-V (FPU) – UzL

3.1.2.1 IP Card

	Basic Info	
IP name	Floating-Point Unit for RISC-V (FPU)	
License	Open-source (SolderPad Hardware License)	
Repository	https://github.com/openhwgroup/cvfpu	
	Architecture	
Clock	Number of clock domains	1
	Synchronous with system	Y
	Clock generated internally	N
Ctrl Interface	ISA extension?	Y
	Memory mapped?	N
	Protocol	
	Address Map	
Initiator Interface	Protocol	
	Cached?	N
	IOMMU?	N
Interrupts	Interrupts	
	Microarchitecture	
Parametrization	Parametric no. units?	Y
	Parameteric config?	Y
Programmability	Contains programmable cores?	Y
	ISA	RISC-V
	Software	
Compiler	Requires specialized compiler?	Y
	Compiler repository	
Hardware Abstraction Layer	N/A	
High-level API	Is there a high-level API/SDK?	N
	SDK repository	
	Is there a domain-specific compiler?	N
	Integration	
IP Distribution	Manifest type (if any)	Bender
	Standalone simulation?	
	(if standalone sim) SW requirements?	VCS / Verilator
	Integration documented / examples?	Example in https://github.com/openhwgroup/cvfpu/blob/develop/docs/README.md
Synthesis	Is the IP synthesizable?	Y
	FPGA synthesis scripts/example available?	N
	ASIC synthesis scripts/example available?	N
Simulation	Closed-source simulation?	VCS
	Open-source simulation?	Verilator
Evaluation	PPA results available?	N

3.1.2.2 Purpose

Floating-Point Units (FPUs) are essential components in modern processors, enabling efficient computation of floating-point operations required in many application domains. Given the highly configurable nature of the RISC-V ecosystem, it is desirable that FPUs are similarly adaptable to a variety of application-specific requirements. The project aims to develop and integrate domain-specific, configurable FPUs into System-on-Chip (SoC) designs.

3.1.2.3 Architecture

Floating-Point Units are specialized arithmetic units to calculate floating-point arithmetics. In modern systems, they are highly integrated into the processor pipeline and support different arithmetic specifications such as *IEEE 754 single precision* (32 bit) and *double precision* (64 bit). In addition, further definitions exist, addressing more specialized applications: for example, *bfloat16* is used and supported by a wide range of Artificial Intelligence (AI) applications.

The RISC-V ecosystem is highly adaptable and configurable. Therefore, it is desirable that the

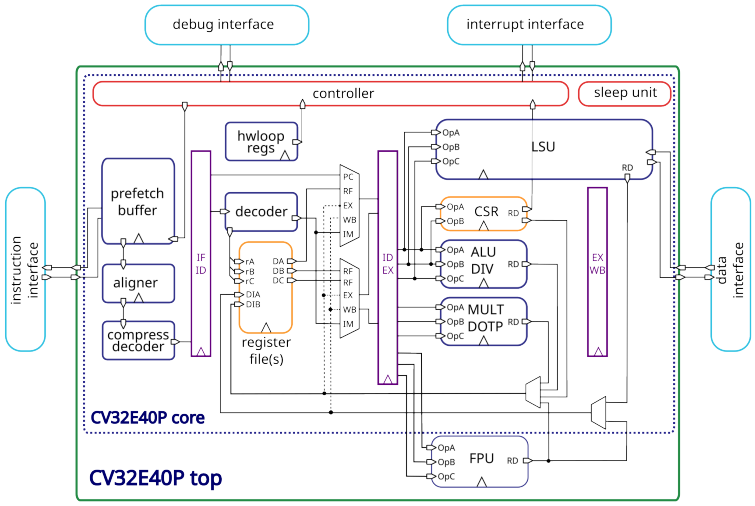

Figure 3.2: Official CV32E40P block diagram including FPU

floating-point unit of RISC-V is also configurable and adaptable for different use cases. During the current runtime of the ISOLDE project, UzL researched and tested a selection of existing open-source floating-point units that are part the RISC-V ecosystem. A most promising example is the OpenHW Group's Floating Point Unit CVFPU [1], which is capable of IEEE 754-2008 single-, double-, quad-, and half-precision specification. It was hence decided to focus on this FPU for adaption and integration into the project work, such as, e.g., the automotive demonstrator.

The following description is based on the CV32E40P Core of the OpenHW Group, but matches most of the existing FPUs. As shown in the CV32E40P core block diagram (Figure 3.2), the FPU is integrated into the processor pipeline with direct access to required operands. The FPU interface is designed to be synchronously cleared during system reset.

3.1.2.4 Evaluation

In alignment with Task 4.3, UzL is analysing the sensitivity of the CVFPU to fault-injection attacks. This contributes to the analysis of the FPU with a focus on aspects such as fault tolerance and overall reliability. The FPU is implemented in accordance with the IEEE 754 standard for single-, double- and quad-precision floating-point arithmetic. It supports a comprehensive set of operations, including addition, subtraction, multiplication, division, square root, and fused multiply-add (FMADD). We opted for single-precision arithmetic for the fault analysis. The FPU is synthesized and compiled using Synopsys VCS [2], with fault injection experiments conducted via Synopsys Z01X [3]. While the FPU is designed to be integrated with the CVA6 processor core, the fault injection campaigns are carried out on the FPU in isolation, without binding it to the full processor pipeline. This allows for more controlled fault analysis and data collection. The golden reference outputs are generated using Python, and the results of the FPU injected with the fault are com-

pared against these references to evaluate the correctness and robustness of the implementation under the fault conditions.

3.1.3 Scratchpad - IMT

3.1.3.1 IP Card

Basic Info		
IP name	Scracthpad memory	
License	Open-source (GPLv3.0)	
Repository	https://gitlab.com/catalin.ciobanu/PolyMem_System_Verilog	
Architecture		
Clock	Number of clock domains	1
	Synchronous with system	Y
	Clock generated internally	N
Ctrl Interface	ISA extension?	N
	Memory mapped?	N
	Protocol	Custom protocol
	Address Map	N/A
Initiator Interface	Protocol	Custom protocol
	Cached?	N
	IOMMU?	N
Interrupts	Interrupts	N
Microarchitecture		
Parametrization	Parametric no. units?	Y (number of data in parallel)
	Parameteric config?	Y
Programmability	Contains programmable cores?	N
	ISA	N
Software		
Compiler	Requires specialized compiler?	N
	Compiler repository	N/A
Hardware Abstraction Layer	N/A	
High-level API	Is there a high-level API/SDK?	N
	SDK repository	N/A
	Is there a domain-specific compiler?	N
Integration		
IP Distribution	Manifest type (if any)	Bender.yml
	Standalone simulation?	Y
	(if standalone sim) SW requirements?	QuestaSim
	Integration documented / examples?	N
Synthesis	Is the IP synthesizable?	Y
	FPGA synthesis scripts/example available?	Y (generated by Bender)
	ASIC synthesis scripts/example available?	N
Simulation	Closed-source simulation?	Y (QuestaSim)
	Open-source simulation?	N
Evaluation	PPA results available?	Y

3.1.3.2 Purpose

A scratchpad memory is a fast memory, similar to a cache, but managed by the user and has its own address space. The user has the responsibility to manage coherency between the main memory and the scratchpad.

Our scratchpad memory is designed for matrix accesses. The access mode is optimized for 2D addresses. The user specifies the coordinates (row and column) for the scratchpad accesses. Furthermore, the memory allows access of multiple data elements in parallel. This parameter is configurable at design time.

The data accessed in parallel can be on the same row, same column, main diagonal, secondary diagonal or could be a small matrix. For our 2D scratchpad, the user may configure the number of read ports.

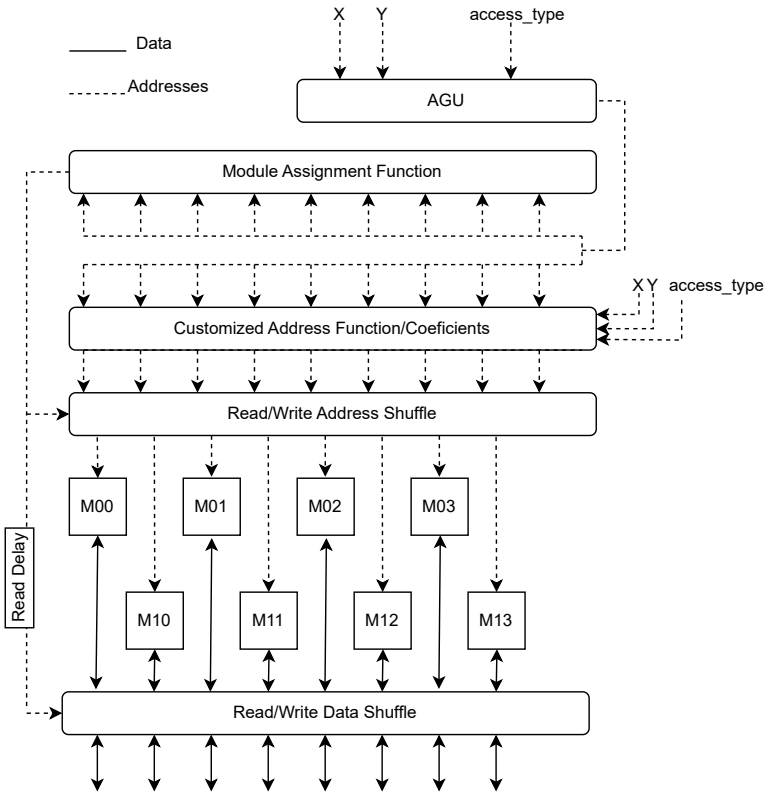

Figure 3.3: High level architecture of the scratchpad memory [5]

3.1.3.3 Architecture

The Scratchpad Memory is based on PolyMem [4] and uses the Memory Access Schemes originally used in the Polymorphic Register File [5, 6].

The main components of the Scratchpad Memory are the memory banks modules and the logic that computes the addresses for every bank based on the selected Memory Access Scheme. The Memory Access Schemes supported [5] are Rectangle Only (ReO) [6], Rectangle Row (ReRo), Rectangle Column (ReCo), Row Column (RoCo) and Rectangle Transposed (ReTr) [5]. The Rectangle Only scheme supports accessing rectangles. ReRo, ReCo, RoCo and ReTr support a minimum of two access patterns and are called multi-view memory schemes. The ReRo scheme supports memory accesses shaped as rectangles, rows (multiple elements from the same line), main and secondary diagonals. ReCo supports rectangles, columns, and main and secondary diagonals. The ReTr scheme allows access to rectangles and transposed rectangles. The RoCo scheme allows accesses to rows, columns and aligned rectangles [5].

The internal structure of the Scratchpad Memory is presented in Figure 3.3. The data is is stored in a matrix of p×q memory modules and has a capacity of N×M words. The number of data elements which can be accessed in each clock cycle is equal to p·q, which will be referred to as the number of lanes in the rest of this document. In Figure 3.3 the scratchpad has 8 lanes, p=2 and q=4 [5].

The Scratchpad module receives the start 2D index and the memory access scheme. Based on them, it generates the addresses for each individual memory bank. The data is read from the memory banks and finally, Data Shuffle rearranges the data to be passed to the user.

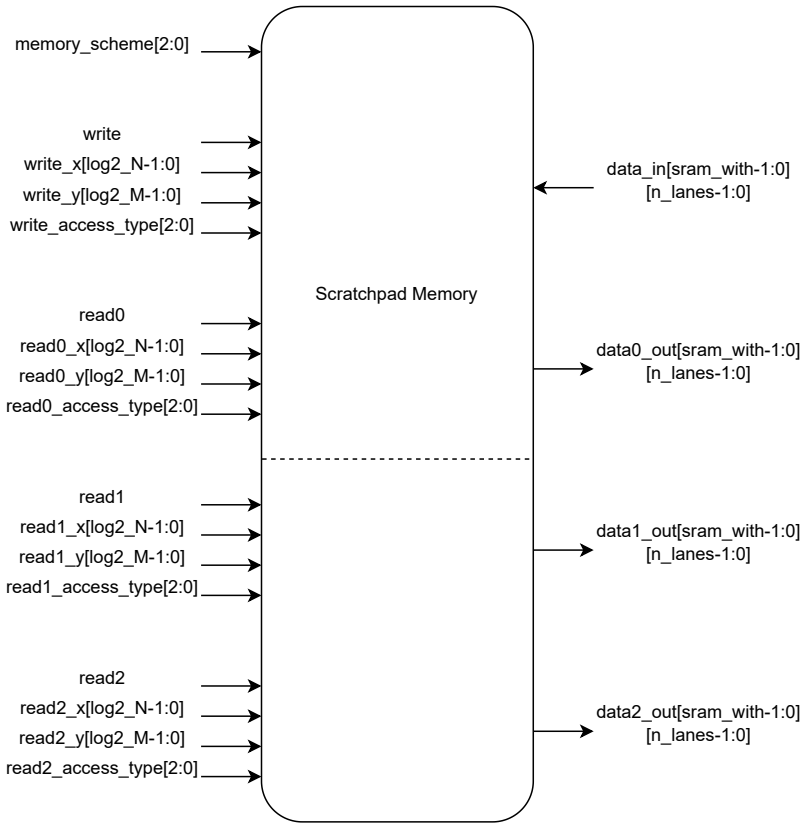

Figure 3.4: Scratchpad interfaces [5]

Interfaces: The default Scratchpad interfaces includes control signals and multiple lanes to write and read data from this memory. Also, the Scratchpad Memory allows multiple parallel reading ports, obtained by duplication of memory modules. The full interfaces are presented in Figure 3.4.

The control signals are the 2D coordinates for writing and reading, memory scheme and the

access type. The access type is one of the six supported: i) rectangle; ii) row; iii) column; iv) main diagonal; v) second diagonal; and vi) transposed rectangle. There are multiple interfaces to write or read data from Scratchpad. The number of read/write of parallel elements is called the number of lanes and is a parameter set at design time.

3.1.3.4 Evaluation

For the scratchpad we analyzed the resource utilization on a Xilinx VCU128 board, resources summary in Table 3.1. The results are in Table 3.2. The scratchpad is configured with two read ports and one write port. The element width is 32 bits and memory size is 1024×1024 elements (N=M=1024), resulting in 4MB of usable data. The internal organization scheme used in this test is *RoCo*. We set the synthesis frequency at 100MHz, the period is 10ns. Based on the worst slack, we computed the maximum frequency with the formula $F_{max} = 1000/(10 - Worst\_Slack)$.

System Logic Cells (K)	2852
HBM DRAM (GB)	8
DSP Slices	9024
Block RAM (Mb)	70.9
UltraRAM (Mb)	270

Table 3.1: VCU128 resources, from [7]

We analyzed the impact of URAM usage on the maximum frequency. Another parameter that we analyzed was the number of lanes. The bandwidth is directly proportional with the number of lanes and the clock frequency. The number of lanes impacts the resource usage because the Scratchpad Shuffle is a fully connected crossbar. Increasing the number of lanes increases the number of LUTs. The increase in the number of LUTs will decrease the maximum frequency.

Lanes	URAM	LUT	FF	BRAM	URAM	Max F [MHz]	Bandwidth [GB/s]
4	YES	2426	72	0	512	202.06	3.23
8	YES	2554	94	0	512	199.32	6.38
16	YES	20976	148	0	512	194.97	12.48
32	YES	38937	553	0	512	162.28	20.77
64	YES	76481	2026	0	512	113.90	29.16
4	NO	1338	48	1856	0	215.66	3.45
8	NO	2240	93	1856	0	214.68	6.87
16	NO	18736	187	1856	0	211.06	13.51
32	NO	39686	723	1856	0	194.17	24.85
64	NO	76481	2026	1856	0	118.46	30.32

Table 3.2: Scratchpad utilization on VCU128. All the results are after synthesis.

As expected, the bandwidth is increased with the number of lanes. Increasing the number of lanes reduces the maximum frequency due to the complexity of the interconnect.

3.2 SIMD/Vector, AI Accelerator and Tensor Processor Unit Design

This section covers a set of hardware accelerators focused on parallel and high-throughput computation, particularly targeting AI/ML workloads, tensor operations, and vectorized data processing, developed as part of the activity of **Task 3.4 – SIMD/Vector, AI accelerator and tensor processor unit design**, led by **FotoNation**. These IPs leverage SIMD and vector architectures as well as tightly integrated processing units designed for inference and training of neural networks.

The IPs included in this category are:

- **AMA – AI/ML Accelerator (FotoNation)**: A proprietary accelerator capable of executing complete neural network programs with FP16 support, high MAC utilization, and scalable configurations (up to 2048 MAC units).
- **ECNNA – CNN Accelerator for Event-Based Sparse Neural Networks (SAL)**: An event-driven sparse CNN accelerator designed around a RISC-V core and CAM memory, optimized for sparse data such as that from event-based sensors.
- **PCA – Parallel Computing Accelerator (POLITO)**: A reconfigurable systolic array with support for approximate arithmetic, designed to accelerate convolutional layers and general matrix operations.
- **TPU – Tensor Processing Unit (UNIBO)**: A heterogeneous PULP-based processing cluster featuring multiple RISC-V DSP cores and dedicated HWPEs (e.g., RedMulE TPE), aimed at GEMM and tensor workloads.
- **VPU – Vector Processing Unit (ETHZ)**: A high-performance, open-source vector unit compliant with RISC-V V 1.0, supporting multi-precision computation, predicated execution, and tight integration with the CVA6 core.
- **SIMD/Vector Accelerator (IMT)**: A RISC-V-coupled matrix accelerator with software-defined 2D registers and a polymorphic scratchpad memory, enabling efficient vector and matrix operations using custom ISA extensions.

These accelerators address the computational needs of AI/ML and signal processing applications by providing modular, scalable, and power-efficient hardware components tailored for integration in ISOLDE's heterogeneous platforms.

3.2.1 AI/ML Accelerator (AMA) – FotoNation

3.2.1.1 IP Card

Basic Info		
IP name	AI/ML Accelerator (AMA)	
License	Closed-source, proprietary	
Repository	N/A	
Architecture		
Clock	Number of clock domains	2
	Synchronous with system	1
	Clock generated internally	0
Ctrl Interface	ISA extension?	No
	Memory mapped?	Yes
	Protocol	APB
	Address Map	-
Memory	Interface Protocol	Standard SRAM
	Hierarchy Level	Top level
Initiator Interface	Protocol	AXI4
	Cached?	No
	IOMMU?	No
Interrupts	Interrupts	Yes
Microarchitecture		
Parametrization	Parametric no. units?	Number of MACS: 256, 512, 1024, 2048
	Parameteric config?	No
Programmability	Contains programmable cores?	No
	ISA	Proprietary
Software		
Compiler	Requires specialized compiler?	Yes
	Compiler repository	N/A
Hardware Abstraction Layer	Not necessary	The supplied driver will manage the interaction with the Accelerator
High-level API	Is there a high-level API/SDK?	N/A
	SDK repository	N/A
	Is there a domain-specific compiler?	Yes, under development
Integration		
IP Distribution	Manifest type (if any)	-
	Standalone simulation?	Yes
	(if standalone sim) SW requirements?	Vivado
	Integration documented / examples?	TBD
Synthesis	Is the IP synthesizable?	Yes
	FPGA synthesis scripts/example available?	Yes, proprietary
	ASIC synthesis scripts/example available?	Yes, proprietary
Simulation	Closed-source simulation?	Yes
	Open-source simulation?	No
Evaluation	PPA results available?	Estimation

3.2.1.2 Purpose

The main purpose of the AI/ML Accelerator is to accelerate the processing of complex AI models/algorithms such as Convolutional Neural Networks (CNN) or Large Language Models (LLM) to be efficiently executed on edge computing devices. The accelerator can process large programs that consist of specific instructions. Each instruction corresponds to, and accelerates a common neural network's layers/operations. Most common operations are supported, including Convolution, Pooling, Element Wise Add and Mul, Matrix Multiplication.

The Accelerator can operate almost completely autonomous or close together with an RISC-V CPU; due to the high data bandwidth required to feed the parallel hardware which can process up to 2048 multiply–accumulate (MAC) or 4096 operations per clock cycle, the autonomous mode is preferred as too much interaction with the CPU can slow down the processing.

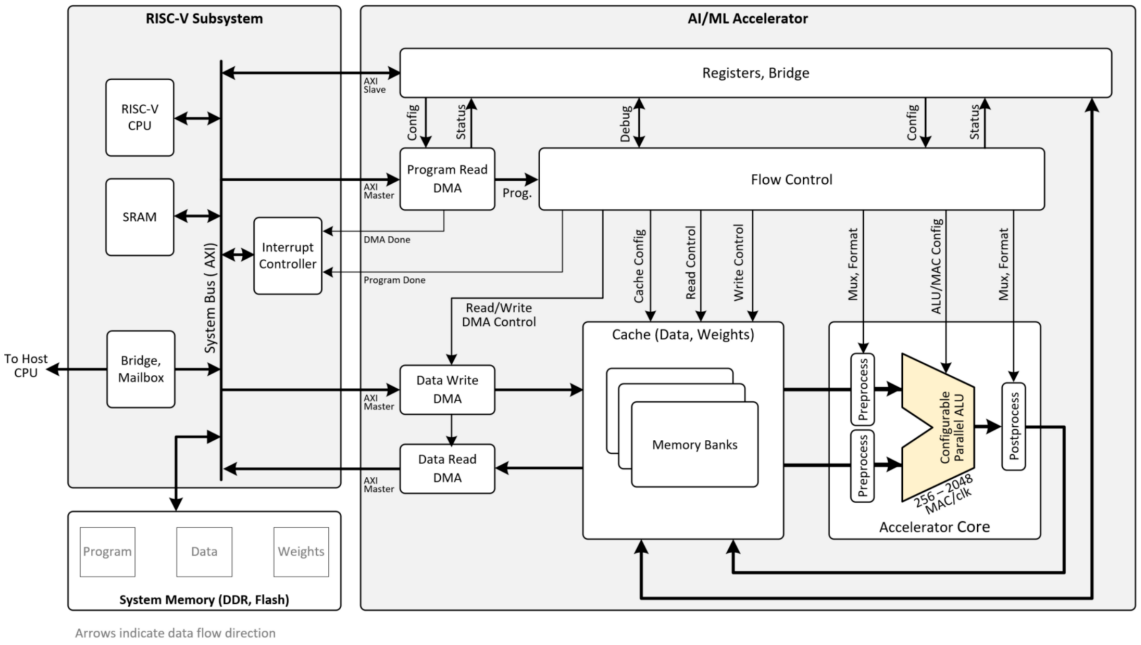

Figure 3.5: AI/ML Accelerator block diagram and associated system architecture.

3.2.1.3 Architecture

The block diagram of the AI/ML accelerator is provided in Figure 3.5; it shows all main component modules and interfaces. The computing architecture is centered around the Accelerator Core that processes the operands stored into the Memory Banks. Parallelism is achieved by storing operands in individual memories, thus achieving a processing speed in the range of 256 – 2048 multiply–accumulate (MAC) operations per clock cycle.

The Memory Banks' size and number are configurable to suit various applications. However, in all configurations it supports the same high data transfer bandwidth that is able to avoid data starving and keep the MAC utilization close to 100% most of the time for the operations that can support it.

The AI/ML accelerator can be easily integrated into any system. All interfaces are standard AXI4 interfaces, 128-bit wide. Although most of the processing is performed by the accelerator, the RISC-V subsystem jointly interacts with the accelerator from the initialization to the results gathering phase. The RISC-V processor configures, starts, and monitors the program execution by the accelerator.

Another important function of the RISC-V processor is to provide flow control for complex programs that need complex loops and conditional branching. The RISC-V core can also access the accelerator cache to perform rare operations that are not supported by the accelerator core. This increases the flexibility of the system, making it possible to implement any kind of processing, but it should be taken into consideration the fact that using the RISC-V processor for data processing

is several orders of magnitude slower in processing performance.

Another very important function provided by the RISC-V subsystem is to assist the debugging of the accelerator programs. The RISC-V can take control of the accelerator by sending instructions one by one, stop the execution after each instruction and analyze the output produced by each instruction.

Processing Flow: A typical processing with the AI/ML Accelerator has the following steps:

- Compilation of the AI/ML model and loading of the resulting program and parameters in the system memory;
- Preparation of the input maps/data in the system memory;
- Powering-up of the AI/ML power island – if it is not already on;
- Starting of the AI/ML clock – if it is not already on;
- Configuring the AI/ML registers;
- Setting of the enable configuration bit for the AI/ML Accelerator;
- Configuring of the Program DMA and starting the DMA transfer of the AI/ML accelerator program;
- The AI/ML accelerator starts fetching the program and executes the instructions:
- The Data Read/Write DMA transfers are controlled from the accelerator program;
- The CPU can monitor the progress of the program using optional interrupts and/or status registers;
- The AI/ML Accelerator asserts the “done” interrupt when the program is completed;
- The CPU can post/process or check the results;
- If the idle status bit is set and if the accelerator is not needed again, the AI/ML clock can be gated;
- Once the clock is gated, the AI/ML power island can be powered down.

Link between requirements and implementation:

- AIACC-01. Support for IEEE754 FP16
  - The entire data path is implemented to use standard FP16 arithmetic modules;
  - Data quantized in smaller resolution (int8, uint8, fp8) can be read from the system memory.
- AIACC-02. Support for NN operations: Convolution, Pooling, Activation Functions
  - The pipeline implementation natively supports these functions – there are dedicated high level instruction for each of the above functions;
  - The functionality was verified against standard PyTorch functions.
- AIACC-03a, c. Standard interfaces (AXI) for System Integration
  - The accelerator only uses standard AXI interfaces to access the system memory for read (program, data) and write (data);
  - Verified with standard protocol checkers (VIP – Verification IP).
- AIACC-03b. APB interface to registers
  - All configuration registers are accessed using a standard APB interface;
  - Verified with standard protocol checkers (VIP – Verification IP).
- IACC-04a, b, c. Integration with CPU (RISC-V)
  - The Accelerator was validated on FPGA, as part of a system with CPU;

- All CPU interaction is performed via standard buses (AXI, APB, interrupt lines), making it very easy to integrate with any CPU (Including RISC-V) that has these interfaces.
- IACC-05. Computational efficiency
  - Rigorous testing in simulation and FPGA was performed;
  - RESNET18 neural network on FPGA works at 10 fps and achieves 90% utilization of the available MAC circuits.
- IACC-06. Scalability
  - Implemented by using a parameter for number of operations: 512, 1k, 2k, 4k
  - All values are fully covered by simulation verification;
  - On FPGA up to 2k operations were verified. 4k operations do not fit on our FPGA platform's device.

3.2.1.4 Evaluation

Performance: Once the RTL code of the accelerator system was completed and passing most simulation regression tests, we prototyped the entire system on an FPGA platform. The FPGA platform is an essential part in the verification/validation process. It is used to run a large variety of instructions and programs, much larger and more complex than what is practically possible in simulation.

The FPGA board used is AMD Virtex™ UltraScale+™ FPGA VCU118 The largest complete system that could fit on this board is configured with 512 MAC (1k operations) per clock cycle. Due to the complexity of the design that does not map very efficiently on an FPGA device, the FPGA Demo reached a maximum clock frequency of 50 MHz. Although it looks low, this frequency is enough for running real time demos and to evaluate the performance of common neural networks.

One neural network that was evaluated is **ResNet18**. On the FPGA platform it reached a performance of 10 fps at input resolution of 224 x 224 pixels. Remarkable is the very high MAC utilization that reached over 90% for this network.

ASIC Synthesis: As the main target of this accelerator is the ASIC implementation, the RTL code was also synthesized for ASIC. The technology node used for ASIC synthesis is **22 nm** (slow corner). In this technology the accelerator system reaches a maximum target frequency of 600 MHz with no timing violations.

The critical path was identified inside the Configurable Parallel ALU. Initial ASIC synthesis tests indicated a lower maximum operating frequency. However, subsequent RTL optimization of the critical paths increased the operating frequency to 600MHz.

The total area of the accelerator system occupied by the logic gates is:

- 256 MACs configuration: 0.56 mm^2 = 2895K gates
- 2048 MACs configuration: 1.47mm^2 = 7346K gates

The memory size used by the memory banks for the typical case is 2048 Kbytes. In 22nm technology this occupies 2.63 mm^2. In terms of energy consumption, for the same technology we determined a value of **0.44 pJ** per single MAC operation. For a 256 MAC accelerator, running at 500 MHz, with 100% utilization of the MACs the total power consumption is **86 mW**, of which 56 mW is consumed by the logic and 30 mW by the memory banks.

3.2.2 CNN Accelerator for an Event-Based Sparse Neural Networks (ECNNA) – SAL

3.2.2.1 IP Card

Basic Info		
IP name	Event-Based Sparse NN Processing Unit	
License	Closed-source	
Repository	Not yet released	
Architecture		
Clock	Number of clock domains Synchronous with system Clock generated internally	1 Y N
Ctrl Interface	ISA extension? Memory mapped? Protocol Address Map	N Y AXI4 64b Base + 0x0 - 0x1000000
Initiator Interface	Protocol Cached? IOMMU?	AXI4 64b N N
Interrupts	Interrupts	Y
Microarchitecture		
Parametrization	Parametric no. units? Parameteric config?	Y Y
Programmability	Contains programmable cores? ISA	Y RV32I
Software		
Compiler	Requires specialized compiler? Compiler repository	Y standard toolchain for cv32e40p
Hardware Abstraction Layer	N/A	
High-level API	Is there a high-level API/SDK? SDK repository Is there a domain-specific compiler?	Y Y (Not yet released) N
Integration		
IP Distribution	Manifest type (if any) Standalone simulation? (if standalone sim) SW requirements? Integration documented / examples?	N Y Python + QuestaSim Y (Not yet released)
Synthesis	Is the IP synthesizable? FPGA synthesis scripts/example available? ASIC synthesis scripts/example available?	Y Y Y
Simulation	Closed-source simulation? Open-source simulation?	Y (QuestaSim) Work in progress
Evaluation	PPA results available?	Work in progress

3.2.2.2 Purpose

The Event-Based CNN Processing Unit (ECNNPU) (Fig. 3.6) is a CAM-based hardware accelerator designed to efficiently perform convolution and pooling operations on highly sparse data, such as event matrix representations.

3.2.2.3 Architecture

The accelerator is built around the cv32e40p—a 4-stage in-order 32-bit RISC-V core—that interfaces via an internal AMBA AHB-Lite bus with a subsystem comprising two CAM memories, each paired with a dedicated RAM partition for storing input and output feature maps in COO format. This subsystem also includes a coordinates processing unit, which operates on the sparse matrix coordinates, and an array of processing elements that handle feature computations.

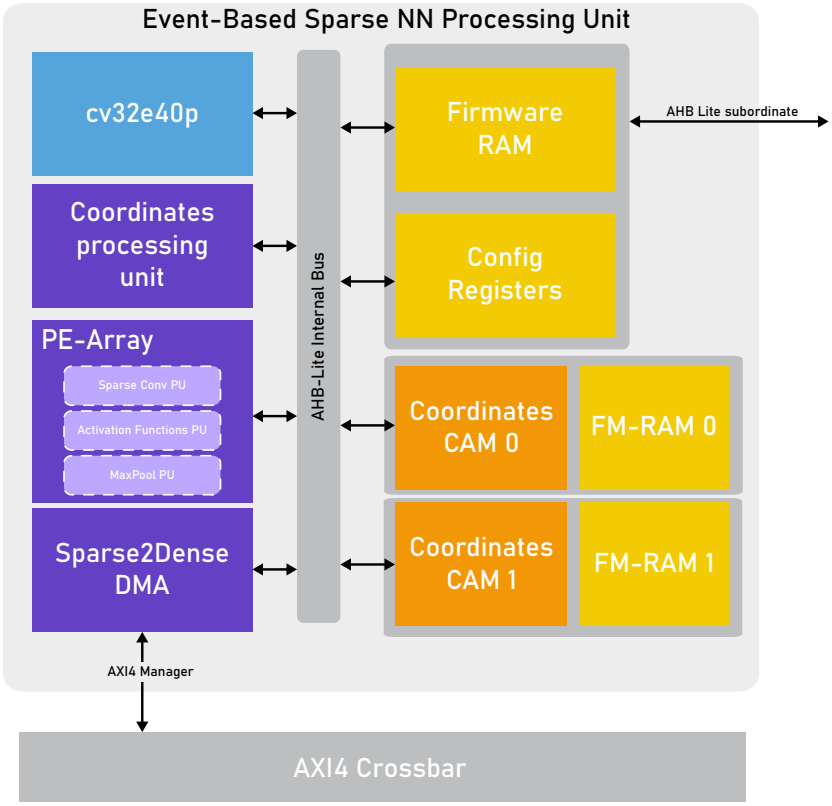

Figure 3.6: SAL Events CNN Accelerator.

Coordinates processing unit: It is a key element in the processing of convolution and maxpool operations. It is fully parametrizable and utilizing the *input size*, *kernel size*, *stride*, and *padding* signals, it calculates all necessary coordinates for processing. The unit is divided into two sub-modules: the *OC Phase* and the *IC Phase*. The *OC Phase* receives an input coordinate from the sparse matrix and computes all corresponding output coordinates affected by that input according to the specified layer parameters. These output coordinates are temporarily stored in an internal FIFO buffer. Subsequently, the *IC Phase* retrieves the stored output coordinates and computes all pairs of input and kernel coordinates required to generate each output coordinate.

PE Array: The PE Array is responsible for executing all feature-related computations. It comprises N parametrizable multiply-and-accumulate (MAC) units connected to a serial adder. This configuration supports two primary operation modes: the MAC units can operate independently on different feature maps, or they can collaboratively contribute to a single feature map by combining their outputs through the serial adder. The results are then fed into an activation function unit that performs rounding operations and supports both ReLU and Leaky-ReLU activations. Additionally, the block includes an array of comparators to facilitate MaxPool operations.

Sparse2Dense DMA: The Sparse2Dense DMA is a custom component that enables the ECN-NPU to interface seamlessly with other accelerators and peripherals. It translates matrices stored in the shared main system memory in a standard dense format into the internal CAM memory format required by the accelerator. Data transmission is handled through an AXI4 manager interface, which allows the DMA to leverage the full capabilities of the AXI bus for efficient data access. Additionally, a set of configuration registers is provided to define the accessible memory address range for this DMA, through the AHB-Lite peripheral interface.

Interfaces: The accelerator has two interfaces to interact with the main system.

- **AXI4 manager:** This interface facilitates the exchange of sparse matrices stored in the main shared memory in dense format with the internal CAM memories that hold data in COO format. A custom DMA engine (*Sparse2Dense DMA* in Fig. 3.6) performs the conversion.
- **AHB-Lite peripheral:** Designed for configuration, this interface enables firmware deployment for the cv32e40p core, layer parameter configuration, and specification of the memory address range accessed by the *Sparse2Dense unit*.

3.2.2.4 Evaluation

The accelerator has been taped out as a standalone ASIC using 65nm TSMC technology and has been successfully tested. The chip occupies a total die area of $9\ mm^2$, of which approximately $1.6\ mm^2$ is dedicated to the core. It is currently undergoing characterization to assess power efficiency across various layer configurations and operating frequencies. Preliminary measurements indicate a power consumption of 26 mW and an energy efficiency of 0.14 TMACs/W at 50 MHz and 1.8 V. The results also demonstrate that the accelerator performs efficiently under highly sparse conditions (approximately 90 % sparsity), making it particularly well-suited for computing the initial layers of a sparse CNN, especially in applications that utilize high-resolution event-based cameras in low-activity scenarios. Once the final evaluations are complete, a journal publication is planned.

3.2.3 Parallel Computing Accelerator (PCA) – POLITO

3.2.3.1 IP Card

Basic Info		
IP name	Parallel Computing Accelerator	
License	Open-source (SolderPad Hardware License v2.1)	
Repository	https://github.com/vlsi-lab/SAURIA-CHESHIRE/	
Architecture		
Clock	Number of clock domains	1
	Synchronous with system	Y
	Clock generated internally	N
Ctrl Interface	ISA extension?	N
	Memory mapped?	Y
	Protocol	AXI4 64b (SRAMs) / AXI4 Lite (Registers)
	Address Map	Base + 0x40000000 - 0x00038000
Systolic Array	Approach	Output Stationary
	On-chip SRAMs?	Y
	Operations	Convolutions / GEMMs
Interrupts	Interrupts?	Y
Microarchitecture		
Parametrization	Parametric Systolic Array shape?	Y
	Parametric SRAMs?	Y
	Configurable PEs?	Y, 256 approximation levels
~~*Programmability*~~	Contains programmable cores?	N
Software		
Hardware Abstraction Layer	Are there macros for direct register access?	Y
	Are there HAL functions?	Y
Integration		
IP Distribution	Standalone simulation?	Y
	SW requirements?	Python + QuestaSim
	Integration documented / examples?	Examples in https://github.com/vlsi-lab/SAURIA-CHESHIRE/tree/main/sw/src
Synthesis	Is the IP synthesizable?	Y
	FPGA synthesis scripts/example available?	Y
	ASIC synthesis scripts/example available?	N
Simulation	Closed-source simulation?	Y (QuestaSim)
	Open-source simulation?	N
Evaluation	PPA results available?	N

3.2.3.2 Purpose

Modern high-performance applications, such as machine learning inference, scientific computation, and signal processing, are increasingly characterized by substantial data-level parallelism, driving the need for specialized hardware accelerators capable of efficiently exploiting this parallelism. As conventional scaling slows and energy efficiency becomes paramount, systolic array-based accelerators have emerged as an effective solution for dense linear algebra operations as well as multiply-accumulate dominated workloads.

Moreover, neural networks while being computationally intensive, have shown high resilience to approximate arithmetic being perfect candidates to achive energy reduction by exploiting operand precision scaling and per-layer approximation levels, allowing fine-grained control over the trade-offs between computational accuracy, energy consumption, and throughput.

In the next section we introduce the Parallel Computing Accelerator (PCA) a flexible accelerator ready for integration into RISC-V-based system-on-chip (SoC) platforms.

3.2.3.3 Architecture

The PCA is a loosely coupled, systolic architecture with AXI interface to be connected to a CVA6-based system. The main application it has been designed for is convolutional layers acceleration

in artificial intelligence applications, with particular focus on multiply and multiply-accumulate intensive kernels.

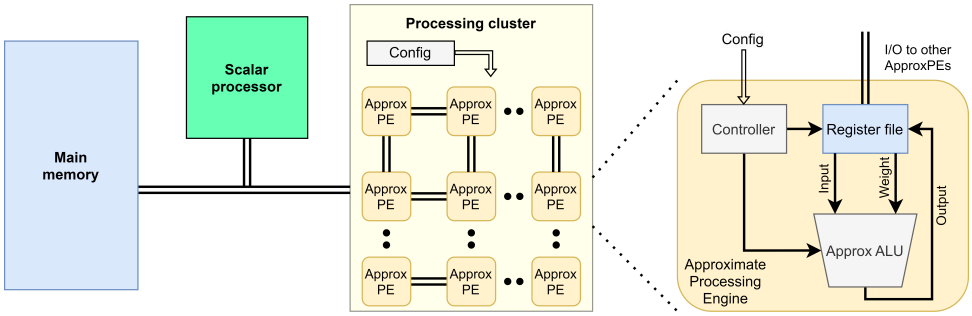

Figure 3.7: Parallel Computing Accelerator architecture.

Figure 3.7 shows the inside architecture by highlighting the configurable systolic array structure [8]. The array can be indeed customized thanks to its parametric architecture; indeed, it can be configured to use different systolic array shapes and sizes, by also adapting the local memory (scratchpad) amount. Each processing element (PE) is designed to perform a multiplication and an addition, and the architecture has been extended by the means of proper configuration information and logic with a run-time reconfigurable signed multiplier, including 256 approximation levels and the possibility to select the operands' precision [9].

The accelerator is memory mapped and equipped with 29 configuration registers and 3 local scratchpads to store the data required for the processing. The configuration registers permit setting several parameters, including the array topology (shape and interconnection between the PEs) as well as multiplication approximation and operands precision.

3.2.3.4 Evaluation

PCA architecture has been integrated in a CVA6 based system and preliminary tests on FPGA show that configuration and simple operations work correctly.

A preliminary evaluation result on 65-nm technology showed that the accelerator can be correctly implemented at 250 MHz, achieving an average power saving of around 38% in the most approximate configuration compared to an 8x8 signed exact multiplier.

3.2.4 Tensor Processing Unit (TPU) – UNIBO

3.2.4.1 IP Card

Basic Info		
IP name	Tensor Processing Unit	
License	Open-source (SolderPad Hardware License v0.51)	
Repository	https://github.com/pulp-platform/pulp_cluster/tree/lg/isolde	
Architecture		
Clock	Number of clock domains Synchronous with system Clock generated internally	1 Y N
Ctrl Interface	ISA extension? Memory mapped? Protocol Address Map	N Y AXI4 64b (async) Base + 0x0 - 0x1000000 (L1 @ Base + 0x0 - 0x40000)
Initiator Interface	Protocol Cached? IOMMU?	AXI4 64b (async) N N
Interrupts	Interrupts	Asynchronous cluster events
Microarchitecture		
Parametrization	Parametric no. units? Parameteric config?	Y (default 8 cores) Y
Programmability	Contains programmable cores? ISA	Y RISC-V (RV32IMCFXpulp2)
Software		
Compiler	Requires specialized compiler? Compiler repository	Y https://github.com/pulp-platform/pulp-riscv-gnu-toolchain
Hardware Abstraction Layer	N/A	
High-level API	Is there a high-level API/SDK? SDK repository Is there a domain-specific compiler?	Y https://github.com/pulp-platform/pulp-runtime/tree/lg-isolde N
Integration		
IP Distribution	Manifest type (if any) Standalone simulation? (if standalone sim) SW requirements? Integration documented / examples?	Bender.yml Y Python (see requirements.txt) + QuestaSim Example in https://github.com/pulp-platform/astral/tree/lg/isolde
Synthesis	Is the IP synthesizable? FPGA synthesis scripts/example available? ASIC synthesis scripts/example available?	Y Y N
Simulation	Closed-source simulation? Open-source simulation?	Y (QuestaSim) N
Evaluation	PPA results available?	https://arxiv.org/abs/2412.06321 (preliminary)

3.2.4.2 Purpose

The ISOLDE Tensor Processing Unit (TPU) is intended as a flexible, easy-to-integrate, and high-performance block that can be parametrized to perform tensor-heavy operations in edge AI systems to accelerate the execution of modern DNNs such as Convolutional Neural Networks, Recurrent Neural Networks, and Transformers. It focuses on relatively high precision (BF16) arithmetic performed either by a set of general-purpose DSP processors or by two embedded high performance engines: a Tensor Processing Engine (TPE) or an Activation Engine (AE). The TPE is designed as a parametric systolic array based on the *RedMulE* architecture.

3.2.4.3 Architecture

The ISOLDE Tensor Processing Unit (TPU) is shown in Fig. 3.8, designed to enhance onboard machine learning and AI performance. Based on the open-source PULP cluster in heterogeneous configuration, it features 8 RISC-V digital signal processing cores based on the CV32E40P architecture (RV32IMCFXpulpV2, in-order, four-stage pipeline) with dedicated FPUs, a hierarchical instruction cache, and a DMA controller for efficient data transfer across the memory hierarchy.

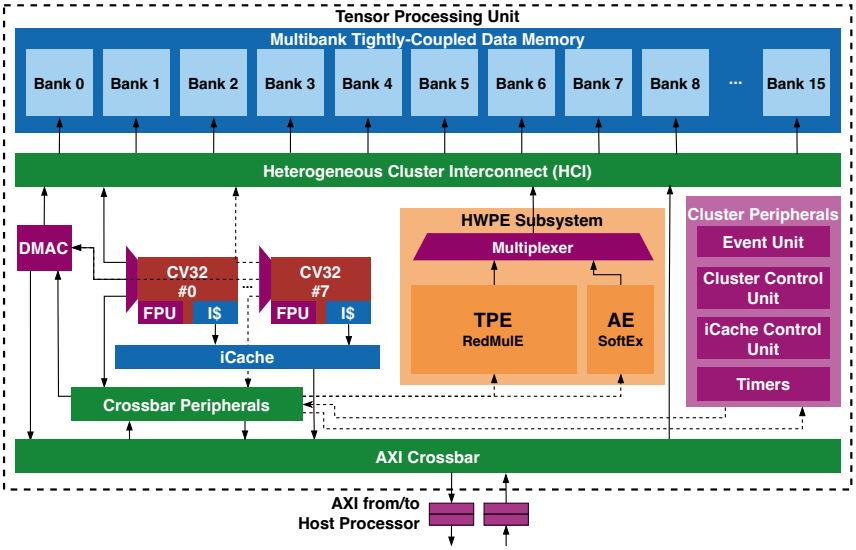

Figure 3.8: UNIBO Tensor Processing Unit architecture.

The TPU includes a 128KiB L1 Tightly-Coupled Data Memory (TCDM), organized into 32 banks, each with a 32-bit data width, shared among all the available computing entities. A low-latency, high-bandwidth heterogeneous cluster interconnect (HCI) enables the DSP cores to share the TCDM with domain-specific Hardware Processing Engines (HWPEs), enhancing performance for specific applications. The TPU includes two HWPEs: a highly parametric Tensor Processing Engine (TPE), based on the RedMulE architecture, accelerating 16-bit (FP16/BFloat16) and 8-bit (E4M3/E5M2) floating-point GEMM and other matrix operations (GEMM-Ops) [10]; and an Activation Engine (AE) based on the SoftEx architecture [11].

Tensor Processing Engine architecture: The TPE builds upon *RedMulE* `https://github.com/pulp-platform/redmule`, a domain-specific processor tailored to accelerate GEMM and GEMM-like computations, which is shown in Fig. 3.9.

At the heart of the TPE lies the *Datapath*, a two-dimensional array of Computing Elements (CEs) arranged in a systolic structure with L rows and H columns. Each row comprises H CEs connected in cascade, where each CE computes an intermediate result and forwards it to the next in a systolic manner. The final CE in each row sends its output back to the first CE in that same row, enabling accumulation. The *Datapath* of the TPE features a configurable number of internal CEs and pipeline stages (P) that can be tuned at design time. To reduce active power consumption, the TPE employs fine-grained clock gating that disables unused sections (rows or columns) of the *Datapath*. Rows that are not engaged during the computation of leftover data can be deactivated dynamically. Similarly, column-level clock gating is applied, allowing RedMulE to disable specific columns based on the current computation phase. Each CE in the *Datapath* is split into two pipeline stages, dedicated respectively to the mapping (usually multiplication) and reduction (typically summation) operations required by the ongoing GEMM-like task.

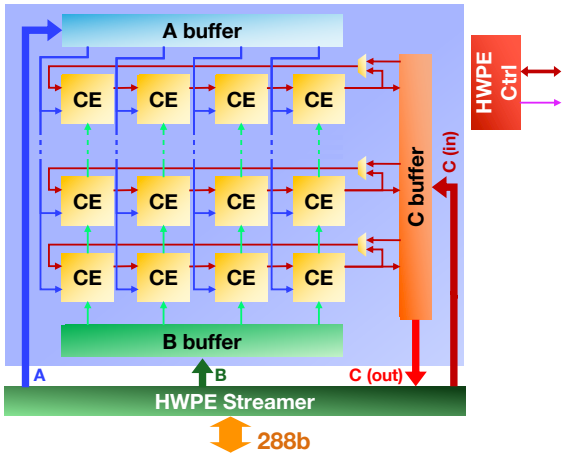

Figure 3.9: Tensor Processing Engine based on *RedMulE*.

To supply the *Datapath* with data, the TPE integrates a module called the *Streamer*, designed in line with the HWPE architectural philosophy[1]. The *Streamer* is a dedicated memory interface that links RedMulE to the TPU cluster interconnect via a single wide data port, with a parameterizable width (as a multiple of 32 bits), used for both loading and storing data. Incoming data from the interconnect is distributed by a single-input/multi-output dispatcher that enables only the appropriate output channel; in parallel, each output channel relays the interconnect stream to the accelerator's input. During store operations, output streams generated by RedMulE are routed back to the interconnect.

The TPE's control logic is split into two submodules: the *Scheduler* and the *Controller*. Together, they include a register file accessible by the cores to configure the accelerator, and they coordinate to manage its execution flow.

3.2.4.4 Evaluation

We targeted GlobalFoundries GF12LP+ 12nm technology to evaluate the TPU in terms of area and performance, using Synopsys Design Compiler for synthesis and Cadence Innovus for place-and-route with a frequency target of 500 MHz. We opted for a TPE with L=12$\times$$H$=4 8-to-16-bit Fused-Multiply-Add units as CEs; and an Activation Engine with 16 lanes. The TPU occupies 0.54mm^2 of silicon, of which the TCDM is 42%, the 8 RISC-V cores are 15% (plus another 11% for the instruction caches), the TPE is 14%, and the AE is 5.5%. The TPU achieves up to 90 GOPS on computation dominated by the TPE.

[1] https://hwpe-doc.rtfd.io

3.2.5 Vector Processing Unit (VPU) – ETHZ

3.2.5.1 IP Card

Basic Info		
IP name	Vector Processing Unit	
License	Open-source (SolderPad Hardware License v0.51)	
Repository	https://github.com/pulp-platform/ara	
Architecture		
Clock	Number of clock domains	1
	Synchronous with system	Y
	Clock generated internally	N
Main Core	RISC-V Core	CVA6
	Cache Write Policy	Write-Through
Ctrl	ISA extension?	RISC-V V 1.0
	Memory mapped?	N
	Interface Protocol	Custom accelerator, MMU, coherence intf. (sync)
Memory	Interface Protocol	AXI4 (L×32) bit (sync) – L = Number of vector lanes
	Hierarchy Level	L2 (CVA6's L1 is bypassed)
Interrupts	Interrupts	Same as for CVA6
Microarchitecture		
Parametrization	Parametric no. units?	N
	Parameteric config?	Y (Number of Vector Lanes (L), Vector Register Length (VLEN))
Programmability	Contains programmable cores?	Y (CVA6)
	ISA	RISC-V (RV64GCV)
Software		
Compiler	Requires specialized compiler?	N (RISC-V compiler compatible with RISC-V V 1.0 is enough)
High-level API	Is there a high-level API/SDK?	Y - C intrinsics (https://github.com/riscv-non-isa/rvv-intrinsic-doc/tree/main)
Integration		
IP Distribution	Manifest type (if any)	Bender.yml
	Standalone simulation?	Y
	(if standalone sim) SW requirements?	QuestaSim or Verilator (see README.md)
	Integration documented / examples?	Y (see README.md)
Synthesis	Is the IP synthesizable?	Y
	FPGA synthesis scripts/example available?	Work in progress
	ASIC synthesis scripts/example available?	N
Simulation	Closed-source simulation?	Y (QuestaSim)
	Open-source simulation?	Y (Verilator)
Evaluation	PPA results available?	https://arxiv.org/pdf/2311.07493 (preliminary)

3.2.5.2 Purpose

Modern compute-intensive applications—in particular, machine learning, scientific simulation, and signal processing—exhibit high degrees of data-level parallelism that cannot be efficiently exploited by conventional scalar cores alone. As transistor scaling trends decelerate, architects have to maximize throughput and energy efficiency through architectural innovations and streamlining. One promising way to boost processing speed and minimize power is through data-level parallelism exploitation. Vector processors, by applying the same operation to multiple data elements in a single instruction, offer a proven mechanism for leveraging data parallelism while amortizing control overhead and maximizing utilization of functional units. Moreover, vector processors support runtime-programmable vector lengths, offering vector-length agnostic programming support.

Beyond raw throughput, emerging workloads demand variable numeric precision: deep neural networks often tolerate reduced precision without sacrificing accuracy, whereas certain scientific kernels require full or extended precision to maintain numerical stability. A vector unit endowed with multi-precision capabilities can dynamically adapt its datapath effective width to the precision requirements of each kernel, thus achieving an optimal trade-off between performance, energy consumption, and result precision. For example, the vector unit can either process N 64-bit data

elements or eight 8-bit data elements in parallel. Such flexibility is critical for heterogeneous systems that have to support applications requiring a broad range of precisions.

In this work, we present the design and integration of a RISC-V V 1.0 vector processing unit featuring multi-precision capabilities tightly coupled to the CVA6 scalar core. The resulting architecture aims to deliver a scalable, energy-efficient platform that transparently accelerates data-parallel kernels across precision domains while preserving full compatibility with the standard RISC-V ISA and software ecosystem.

3.2.5.3 Architecture

The architecture of the RISC-V V 1.0 multi-precision vector processing unit (VPU) is represented in Figure 3.10.

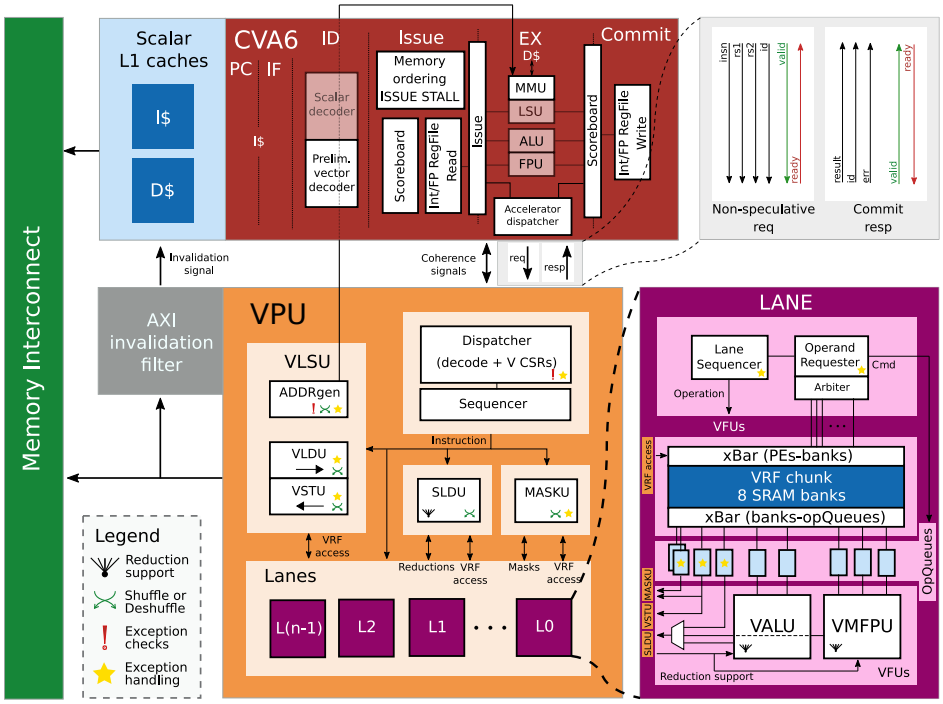

Figure 3.10: Detailed ETHZ Vector Processing Unit architecture.

The VPU is tightly coupled with an RV64GC scalar core and extends its ISA to RV64GCV. It supports integer, fixed-point, and floating-point data from 64 to 8-bit. The official RISC-V V support is not guaranteed with the default compiler for non-standard floating-point 8-bit data types, even if hardware support is in place.

Additional information about a preliminary version of the architecture can be found at [12].

Scalar Core: The VPU is compatible with every scalar core that implements its interfaces. The open-source RV64GC CVA6 RISC-V core is compliant with our VPU. In addition to the interface requirement, the scalar core features a write-through L1 cache to ensure cache coherence with the upper level of the memory, which is directly accessed by the VPU.

The VPU comes with a private memory Load/Store Unit (LSU) but uses the scalar core's Memory Management Unit (MMU) to support virtual memory, which is mandatory when running an operating system such as Linux.

In the following, we will use CVA6 when referring to the scalar core of our decoupled VPU architecture.

Interfaces: The VPU exposes multiple interfaces to its scalar core and the memory.

- **Request/Response interface**: Used to forward vector instructions and scalar operands to the VPU, and receive results (if needed) and information on the possible occurred exception. The interface is composed of an handshaked CVA6-to-VPU request (`id`, `insn`, `rs1`, `rs2`, `frm`) and a VPU-to-CVA6 response (`id`, `result`, `exception`, `fflags`). A request is started by CVA6, which forwards non-speculatively a vector instruction with the corresponding scalar source operands to the VPU. Then, the VPU answers after all the exception checks (and scalar result calculation, if needed). At this point, CVA6 can commit the vector instruction while it is being processed by the VPU (provided that no exception has occurred).
- **MMU interface**: Used to let the VPU use virtual memory. The interface is composed of a VPU-to-CVA6 request (`acc_mmu_req`, `acc_mmu_vaddr`, `acc_mmu_is_store`) and a CVA6-to-VPU response (`acc_mmu_valid`, `acc_mmu_paddr`, `acc_mmu_exception`). A request is started by the VPU, which forwards a virtual address to CVA6's Memory Management Unit (MMU). The MMU answers back with the physical address and information on the possible exception (e.g., page fault).
- **Coherence interface**: Used to keep memory consistency, coherence, and ordering between CVA6 and the VPU, and between the L1 and L2 levels of the memory hierarchy. The interface is composed of a set of CVA6-to-VPU signals (`acc_cons_en`, `store_pending`) and a set of VPU-to-CVA6 signals (`load_complete`, `store_complete`, `store_pending`). These signals are used by CVA6 and the VPU to prevent memory operation ordering violations. Also, a dedicated AXI invalidation filter sends handshaked invalidation requests to CVA6 with a 64-bit address bus to invalidate cache sets that could have been made stale by a vector store to the L2 memory.
- **Memory interface**: AXI4 interface, with L$\times$32-bit `R` and `W` channels.

VPU: Vector instructions are decoded by the VPU, which also contains the RISC-V control and status registers (CSRs), and broadcast to all the VPU's units by its main sequencer, which also keeps track of the high-level dependencies between vector instructions.

The VPU is composed of a parametric number of vector lanes. Each lane contains a vertical slice of the Vector Register File (VRF) implemented with 8 SRAM banks, a SIMD integer ALU, a SIMD integer multiplier, a SIMD integer divider, and a SIMD FPU. Every `MACC` operation is computed by VPU's unis in one cycle. Within one lane, the datapath of each VRF bank and computational unit is 64-bit wide. With default settings, each VRF slice has a size of 4 KiB.

Data is brought from the memory to the in-lane VRF slices (and vice-versa) through the vector LSU. Also, the bytes of the vector register can be moved among lanes by means of the Slide Unit (SLDU). Predicated execution is supported thanks to the Mask Unit (MASKU), which handles bit-level vector shuffling to prepare byte strobes for the computational units, to selectively enable or disable computation on particular vector element indexes.

The MASKU is also responsible for implementing vector instructions that operate on vectors with a mask-byte-layout. The `popc` and `vfirst` instructions are multicycle and work on a parametric bit-width to trade off IPC and routability (default: 16 bit/cycle). This also happens for the instructions that generate other mask vectors in the MASKU, such as `vms{b,i,o}f` (default: half of the number of lanes). Finally, the MASKU handles instructions such as `vrgather` and `vcompress` by preparing the vector indexes in a FIFO and fetching the data from the correct lane. For better routability, this instruction can fetch a maximum of one element per cycle.

All these units outside of the lanes work on a L×64 bit wide datapath. The memory bandwidth of the architecture is L×32 bit/cycle. Thus, the computation-to-memory-bandwidth ratio is 2×.

Exception handling: Exceptions are reported to CVA6, which handles them. Floating-point exception flags are reported asynchronously with a handshake interface to keep them updated in CVA6's appropriate CSR. A real exception can occur in the middle of a vector instruction only in the case of memory operations, especially when virtual memory is enabled. When this is the case, and an exception is raised, the VPU 1) reports the exception to CVA6, 2) waits until the operations on the elements before the faulty one has finished, and 3) starts a flushing procedure that clears the micro-architectural state in approximately 10 cycles. This operation is not latency-critical since CVA6 is serving as an exception handler in the meantime.

Figure 3.11 shows a hypothetical high-level architectural integration of the multi-precision VPU in an SoC.

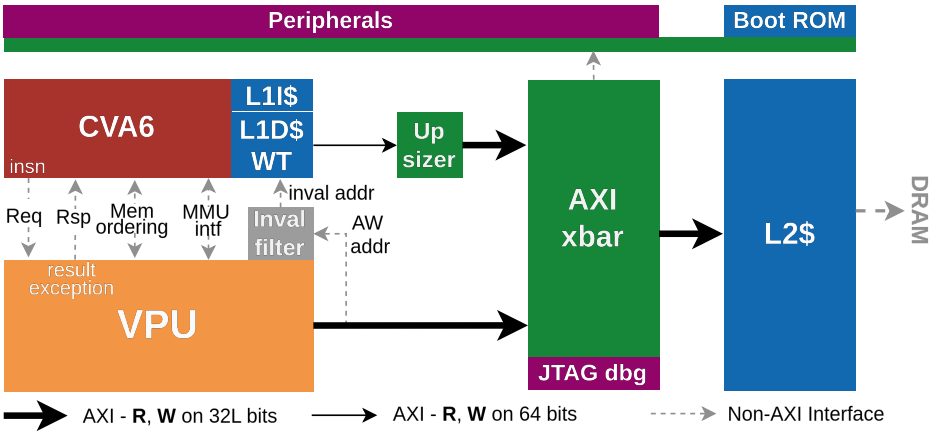

Figure 3.11: High-level integration of the ETHZ Vector Processing Unit architecture.

The VPU directly accesses the L2 memory (e.g., L2 cache) by means of a wide AXI4 port. Depending on the number of vector lanes, the data width of the R and W channels can be higher

than 64 bit, the default width of CVA6's L1 cache refill port. Therefore, an upsizer can be needed to adapt its width before entering the main interconnect (or, in the alternative, a downsizer on the VPU's memory port, which can, however, impact VPU performance).

3.2.5.4 Evaluation

The architecture is in continuous evolution thanks to feature addition, improved verification, and optimizations. A preliminary evaluation on an early and incomplete prototype of the architecture's PPA metrics in 22-nm technology showed that a CVA6 + VPU architecture (VPU configured with 4 lanes) occupies less than 4 MGE and reaches 10.7 DP-GFLOPS with 280 mW of average power consumption (i.e., with an energy efficiency of 37.8 DP-GFLOPS/W) when processing a double-precision floating-point matrix multiplication on 256x256 matrices at 1.35 GHz (TT,0.8V,25C). More information on the preliminary evaluation can be found at [12].

3.2.6 Vector-SIMD Accelerator – IMT

3.2.6.1 IP Card

Basic Info		
IP name	SIMD/Vector accelerator	
License	Open-source (GPLv3.0)	
Repository	https://github.com/alex2kameboss/MatrixAccelerator.git	
Architecture		
Clock	Number of clock domains	1
	Synchronous with system	Y
	Clock generated internally	N
Ctrl Interface	ISA extension?	Y
	Memory mapped?	N
	Protocol	CV-X-IF
	Address Map	N/A
Initiator Interface	Protocol	CV-X-IF
	Cached?	N
	IOMMU?	N
Interrupts	Interrupts	N
Microarchitecture		
Parametrization	Parametric no. units?	Y (default 8 lanes)
	Parameteric config?	Y
Programmability	Contains programmable cores?	N
	ISA	RISC-V (RV32IMAC_zicsr)
Software		
Compiler	Requires specialized compiler?	Y
	Compiler repository	https://github.com/alex2kameboss/MA-riscv-gnu-toolchain.git
Hardware Abstraction Layer	N/A	
High-level API	Is there a high-level API/SDK?	N
	SDK repository	N/A
	Is there a domain-specific compiler?	N
Integration		
IP Distribution	Manifest type (if any)	Bender.yml
	Standalone simulation?	Y
	(if standalone sim) SW requirements?	QuestaSim
	Integration documented / examples?	Example in https://github.com/alex2kameboss/MatrixAcceleratorDemo.git
Synthesis	Is the IP synthesizable?	Y
	FPGA synthesis scripts/example available?	Y (by Bender and custom scripts)
	ASIC synthesis scripts/example available?	N
Simulation	Closed-source simulation?	Y (QuestaSim)
	Open-source simulation?	N
Evaluation	PPA results available?	Y, initial evaluation in this document

3.2.6.2 Purpose

The SIMD accelerator is intended to accelerate matrix operations. Optimized hardware topologies are used for specific matrix operations.

As a main feature, our accelerator features software defined 2D registers. The data is stored in a 2D internal memory that simplifies matrix storing and accesses.

The accelerator has its own RISC-V custom extension to reduce code size and committed instructions. The instructions are not expanded by compiler and are handled by hardware inside of accelerator.

3.2.6.3 Architecture

The IMT SIMD/Vector accelerator is tightly coupled. The accelerator is connected to a RISC-V core via an extension interface, called CV-X-IF [13]. Because the accelerator is connected to a scalar core, we extended the RISC-V ISA.

Figure 3.12 presents the internal architecture of the matrix accelerator. The main components are: *Decode Unit*, *Control Unit*, *Internal Memory*, *DMA Unit*, *Vector Unit* and *Matrix Unit*.

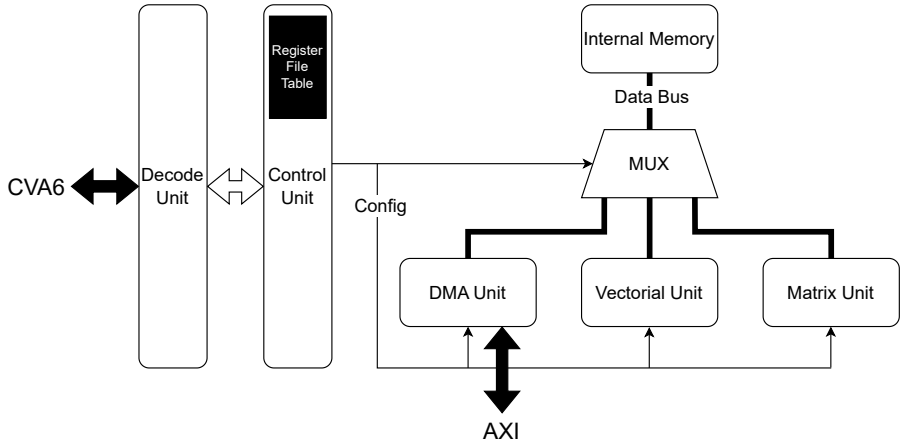

Figure 3.12: SIMD accelerator internal architecture

The *Control Unit* is the most important component. This unit incorporates the *Register File Table*. This table stores information about the 2D software defined registers. The information about a 2D register comprises of the width and height (expressed as number of elements), coordinates regarding placement in the 2D internal memory and the data type. Another task of the Control Unit is to choose the suitable arithmetic unit for the matrix operations. The integrity of the register data is handled by us in the design, no extra hardware protection is implemented.

Decode Unit handles the CV-X-IF protocol. This component validates custom instructions, checking if the unimplemented instruction is intended for our SIMD/Vector accelerator. Another task is to access the register data from the core if the instruction requires it. When the component has all data, instruction and registers data, will send the operation to the *Control Unit* to execute it. When the execution is done, this information is communicated to the core.

A key component of the matrix accelerator is *Internal Memory*. The internal memory has the same design as the Scrachpad memory described in Section 3.1.3 and it is based on the Polymorphic Register File (PRF) [5]. A key feature is the 2D organization. The memory is distributed in multiple independent memory banks and acts as a bi-dimensional address space. For this work, the internal memory is configured to use the *RoCo* memory scheme. This organization allows us to read multiple data elements in parallel with different patterns: we can access multiple data on the same row, same column or a small sub-matrix. This memory module has two read ports, and one write port. This is needed because arithmetic operations have two operands. The dual read interfaces allow us to read the operands in parallel. The write and read operations are handled in parallel using independent ports.

The accelerator has three execution modules. The *DMA Unit* has the role to load or store data in internal memory. The *Vector Unit* can handle simple matrix operations, like those that can simply be handled by vector processors. Those simple operations are matrix addition, subtraction and

multiplication element-by-element (cross product). Another arithmetic unit is the *Matrix Unit*. This unit handles matrix multiplication and convolution (this operation is currently in development).

Those units do not work in parallel. The *Control Unit* configures a multiplexer and gives access to the internal memory. Furthermore, the control unit passes information about the software defined registers used in the operation. All units compute read and write 2D addresses based on that information.

To improve memory utilization, we consider the case when processing narrow data data types that are shorter than width of the memory. The Polymorphic Register File is composed of memory banks that are 32-bits wide. When we read 8-bit data, we can pack four elements in one 32-bit word stored in memory. The hardware takes this scenario into consideration and splits the elements as needed.

An important parameter for our accelerator is the number of parallel lanes. This parameter sets the number of data elements that can be accessed in parallel from the Polymorphic Register File. The number of arithmetic units in the *Vectori Unit* is proportional with the number of lanes: one ALU for every lane. Internally, the *Matrix Unit* has a systolic array that is dependent on number of lanes. The height of this systolic array is equal with the number of lanes and the width is four times the number of lanes. The unit is wider because if we process 8-bit data we must process $4\times$ more data in parallel that for 32-bit numbers.

3.2.6.4 Experimental set-up

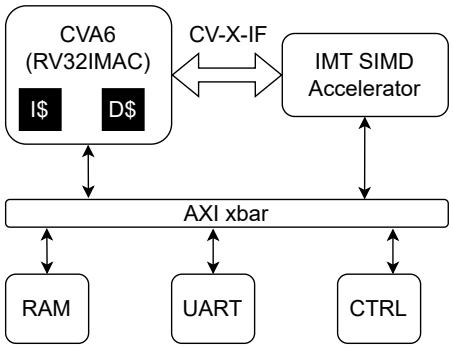

Figure 3.13: High level architecture of the matrix accelerator demonstrator

Figure 3.13 presents the high-level architecture of the demonstrator. We use the CVA6 [14] RV32I RISC-V core with *MAC_zicsr* extensions: hardware multiplication, compressed instructions, atomic extension and the Control Status Register (CSR) extension. In our configuration, the core has 4KB of instruction cache, 8KB of data cache with write-through mode and has in-order execution. A summary about the RISC-V core configuration is presented in Table 3.3.

The scalar core is connected to our SIMD accelerator via CV-X-IF [13]. Both are connected to the same interconnect and share the same address space. The system has an RAM component, but in our case also acts as a pseudo ROM because before starting we write the entire code to the main memory.

RISC-V Core	CVA6
Extensions	IMAC_zicsr
L1 I$	4KB
L1 D$	8KB

Table 3.3: Demonstrator scalar core configuration

The UART component acts as a standard output: we redirect all output text to it. In that way, we have a console for our system. The *CTRL* component stands for control. it is used to stop the simulation: after the return from the main function, the code will set a flag in this component. Furthermore, this component has a counter that is used for accurate measurements. We start the counter and stop it after the section of code measured has completed execution. We may then read the counter to collect performance metrics.

The benchmark suit has multiple tests employing various data types and matrix operations. The data types tested are integer numbers on 8, 16 and 32-bit. The tested operations are matrix addition, subtraction, cross and dot product. All the tests have the same structure. The test starts by creating the operands matrices and initializing them with random values. After that, those matrices are defined and uploaded in the accelerator. The command for the matrix operation is sent to the accelerator and when it is complete the data is written back to the main memory. Furthermore, the result of the matrix operation is computed in software. Before reading the hardware results, the data cache is flushed. In the end, the hardware and software results are compared against each other. In that way, we measure speed-up and complete functional verification of accelerator.

To analyze the speed-up, we measure the elapsed time on the accelerator and on core. The time on accelerator takes into consideration the register definition and memory operation, both load operations and storing back the result. In that way, we want to measure the real time spent when using the accelerator. The software computation time is measured before flushing the cache. In that way we prevent a cold cache scenario.

In this set-up, the Register File has a constant size. It is configured with has two read ports - one for each operand, and one write port for result. The 2D Register File has a word size of 32-bit and a capacity of 1024×1024 elements, resulting in 4MB of usable storage.

For the FPGA tests we used the Xilinx VCU128 board. The resources available on this board are sumarized in Table 3.4. The board has High Bandwidth Memory (HBM) and we will use this memory as RAM for our FPGA experiments.

System Logic Cells (K)	2852
HBM DRAM (GB)	8
DSP Slices	9024
Block RAM (Mb)	70.9
UltraRAM (Mb)	270

Table 3.4: VCU128 available resources [7]

3.2.6.5 Evaluation

Table 3.5 presents the results for our accelerator configured with 8 parallel data lanes. The results are obtained using a RTL simulator with very optimistic memory latency. This test highlights the maximum speed-up we could obtain in a scenario where a high performance L2 cache would be used. To have a functional system, the L2 memory needs to be shared between the accelerator and the scalar core.

The test inputs are two square matrices. The number of rows in the matrix is presented in the table in column *Matrix size*. The result is a square matrix with the same size. We analyze four types of matrix operations: matrix addition (*Addition*), matrix subtraction (*Subtraction*), general matrix multiplication (*Dot Product*) and element-by-element multiplication (*Cross Product*). The *Core clock cycles* column lists the number of clock cycles on the RISC-V core with simple algorithms. We measure the clock cycles of the operations on both the accelerator and scalar core.

In the accelerator there are two arithmetic units. The *Vector Unit* that computes element-by-element operations: addition, subtraction and cross product. This is the reason those operations have the same speed-up: the mathematical operation is different but data and control paths are the same. On the other hand, the matrix multiplication is handled by the *Matrix Unit*, which employs a systolic array. A systolic array is a dedicated hardware topology for matrix multiplication with large throughput [15]. This is why the reported speed-up is so high.

Increasing the data width affects the number of memory transactions. The number of clock cycles spent on the computation on the accelerator increases when employing wider data types. However, the accelerator uses AXI Burst transactions which helps moving large volumes of data.

In Table 3.6 we study the impact of the number of lanes on the runtime of the accelerator. The results are obtained from RTL simulations using the same scenario as the tests from Table 3.5. Increasing the number of lanes decreases the total number of clock cycles. However, this reduction is not linear because the measurement also includes the definition of vector registers and the memory operations - loading the data and storing the results.

Table 3.7 presents the resource utilization on the FPGA. The same presents both the resource usage for the demonstrator and the stand alone SIMD accelerator. The synthesis results are obtained using Xilinx Vivado 2024.1.

In order to evaluate the scalability of our design we performed a design space exploration. We varied the number of lanes from 4 to 32. Furthermore, we studied the impact on the maximum clock frequency when using either URAM or BRAM for the Polymorphic Register File.

The number of lanes directly impacts the number of parallel arithmetic units. The DSPs scale almost quadratically. When we double the number of lanes the systolic array size is doubled both in height and in width, leading to quadratic area increase. For regular vector operations such as addition each lane has a corresponding ALU, leading to a linear resource increase.

When doubling the number of lanes the number of flip-flops (FFs) also doubles. The pipelines in the accelerator are becoming wider as we increase the number of lanes. Furthermore, the control paths have the same resource utilization as the memory module size is not changing. The control elements resources only depend on the total capacity of the 2D Register File.

In Table 3.8 we analyze scalability of the design in terms of the maximum frequency with regard

Test type	Data type	Matrix size	Accelerator clock cycle	Core clock cycles	Speed-up
Addition	int8_t	32	474	84765	178.83
Subtraction	int8_t	32	470	84678	180.17
Dot Product	int8_t	32	514	3556085	6918.45
Cross Product	int8_t	32	463	85694	185.08
Addition	int16_t	32	666	90841	136.40
Subtraction	int16_t	32	655	91626	139.89
Dot Product	int16_t	32	818	3885403	4749.88
Cross Product	int16_t	32	662	92692	140.02
Addition	int32_t	32	1050	71449	68.05
Subtraction	int32_t	32	1046	73532	70.30
Dot Product	int32_t	32	1450	2944367	2030.60
Cross Product	int32_t	32	1039	74652	71.85
Addition	int8_t	64	1434	337409	235.29
Subtraction	int8_t	64	1430	337525	236.03
Dot Product	int8_t	64	1977	28260326	14294.55
Cross Product	int8_t	64	1430	341568	238.86
Addition	int16_t	64	2206	364147	165.07
Subtraction	int16_t	64	2200	365598	166.18
Dot Product	int16_t	64	3757	31168262	8296.05
Cross Product	int16_t	64	2200	369687	168.04
Addition	int32_t	64	3746	291650	77.86
Subtraction	int32_t	64	3742	292077	78.05
Dot Product	int32_t	64	7346	25915214	3527.80
Cross Product	int32_t	64	3740	296144	79.18
Addition	int8_t	128	5273	1345403	255.15
Subtraction	int8_t	128	5278	1345428	254.91
Dot Product	int8_t	128	11457	245374248	21416.97
Cross Product	int8_t	128	5271	1362040	258.40
Addition	int16_t	128	8362	1457864	174.34
Subtraction	int16_t	128	8356	1458075	174.49
Dot Product	int16_t	128	22724	270321852	11895.87
Cross Product	int16_t	128	8358	1474422	176.41
Addition	int32_t	128	14524	1163889	80.14
Subtraction	int32_t	128	14516	1163932	80.18
Dot Product	int32_t	128	45260	210673885	4654.75
Cross Product	int32_t	128	14518	1180163	81.29

Table 3.5: Simulation results for a 8-lane SIMD accelerator

to the number of lanes and the use of URAM. We set the synthesis frequency at 100MHz, 10ns period. Based on the worst slack, we computed the maximum frequency with the formula $F_{max} = 1000/(10 - Worst\_Slack)$.

Test type	Data type	Clock cycles				Speed-up		
		Scalar Core	4 lanes	8 lanes	16 lanes	4 lanes	8 lanes	16 lanes
Addition	int8_t	336811	1944	1434	1202	173.26	234.88	280.21
Subtraction	int8_t	337378	1940	1430	1196	173.91	235.93	282.09
Dot Product	int8_t	28260690	5036	1977	1282	5611.73	14294.73	22044.22
Cross Product	int8_t	341515	1940	1430	1196	176.04	238.82	285.55
Addition	int16_t	363983	2716	2206	1988	134.01	165.00	183.09
Subtraction	int16_t	365529	2703	2200	1975	135.23	166.15	185.08
Dot Product	int16_t	31165445	9896	3757	2292	3149.30	8295.30	13597.49
Cross Product	int16_t	369669	2703	2200	1975	136.76	168.03	187.17
Addition	int32_t	291646	4247	3746	3551	68.67	77.86	82.13
Subtraction	int32_t	291935	4252	3742	3554	68.66	78.02	82.14
Dot Product	int32_t	25913921	19624	7346	4360	1320.52	3527.62	5943.56
Cross Product	int32_t	296161	4252	3740	3554	69.65	79.19	83.33

Table 3.6: 64×64 matrices application runtime Design Space Exploration

Lanes	URAM	Demonstrator					Accelerator				
		LUT [K]	FF [K]	BRAM	URAM	DSP	LUT [K]	FF [K]	BRAM	URAM	DSP
4	YES	47.8	33.9	38	512	212	17.0	13.9	4	512	208
8	YES	65.4	54.5	42	512	800	34.5	34.5	8	512	796
16	YES	161.7	141.2	50	512	3128	130.9	121.3	16	512	3124
32	YES	1091.6	481.2	64	512	8642	951.0	461.2	30	512	8640
4	NO	55.8	33.8	1894	0	212	25.1	13.8	1860	0	208
8	NO	73.4	54.5	1898	0	800	42.0	34.6	1864	0	796
16	NO	164.2	141.3	1904	0	3128	132.9	121.3	1870	0	3124
32	NO	1096.4	481.6	1920	0	8642	955.8	461.6	1884	0	8640

Table 3.7: VCU128 resource utilization Design Space Exploration

Lanes	URAM	Demonstrator	Accelerator
		Max F [MHz]	Max F [MHz]
4	YES	133.65	196.54
8	YES	131.60	197.78
16	YES	135.28	194.97
32	YES	66.63	66.63
4	NO	143.23	196.54
8	NO	147.28	156.49
16	NO	144.68	156.49
32	NO	66.63	66.63

Table 3.8: Maximum clock frequency Design Space Exploration

The accelerator frequency is limited by the clock frequency of the internal memory, based on the Polymorphic Register File. The clock frequency observed for the accelerator is a few MHz below the stand-alone PRF. We observed that the number of lanes has also has an impact on the systolic array. On the 32-lanes design, the control path of the systolic array becomes very large and complex. The systolic array control path will require further analysis.

3.2.7 Extension Platform (EXP) – TUI

3.2.7.1 IP Card

Basic Info		
IP name	Extension Platform (EXP)	
License	Open-source (SolderPad)	
Repository	Not yet released	
Architecture		
Clock	Number of clock domains	1
	Synchronous with system	Y
	Clock generated internally	N
Ctrl Interface	ISA extension?	Y
	Memory mapped?	N
	Protocol	CV-X-IF
	Address Map	N/A
Initiator Interface	Protocol	AXI-S, AXI to AXI-S bridge, CV-X-IF
	Cached?	N
	IOMMU?	N
Interrupts	Interrupts	Y
Microarchitecture		
Parametrization	Parametric no. units?	Y
	Parameteric config?	Y
Programmability	Contains programmable cores?	N
	ISA	RISC-V (RV64IMAC)
Software		
Compiler	Requires specialized compiler?	N
	Compiler repository	N/A
Hardware Abstraction Layer	Linux kernel-mode driver	N, work in progress
	Bare metal HAL	Y
High-level API	Is there a high-level API/SDK?	N
	SDK repository	N/A
	Is there a domain-specific compiler?	N
Integration		
IP Distribution	Manifest type (if any)	N
	Standalone simulation?	Y
	(if standalone sim) SW requirements?	XSIM, QuestaSim
	Integration documented / examples?	N, work in progress
Synthesis	Is the IP synthesizable?	Y
	FPGA synthesis scripts/example available?	Y
	ASIC synthesis scripts/example available?	N
Simulation	Closed-source simulation?	Y (QuestaSim, XSIM)
	Open-source simulation?	Y (Verilator)
Evaluation	PPA results available?	N, work in progress

3.2.7.2 Purpose

The proposed accelerator is a scalable hardware unit optimized for DSP algorithms, including efficient 1D and 2D DCT/iDCT on streaming data, aimed at real-time applications. It targets RISC-V SoCs and FPGAs, providing high throughput, low latency, and flexible integration through AXI4-Stream and a lightweight control interface.

3.2.7.3 Architecture

The Extension Platform (EXP) is designed as a scalable, composable architecture for deploying heterogeneous compute engines (PEs) optimized for digital signal processing - Figure 3.14. Based on the architecture presented in the EXP, a DCT/iDCT accelerator is effectively integrated as a set of dedicated Processing Engines (PEs) within the EXP.

This vector-style hardware accelerator is designed for real-time DSP workloads, including signal transformations, feature extraction, and image/video compression, on RISC-V-based SoCs and FPGAs.

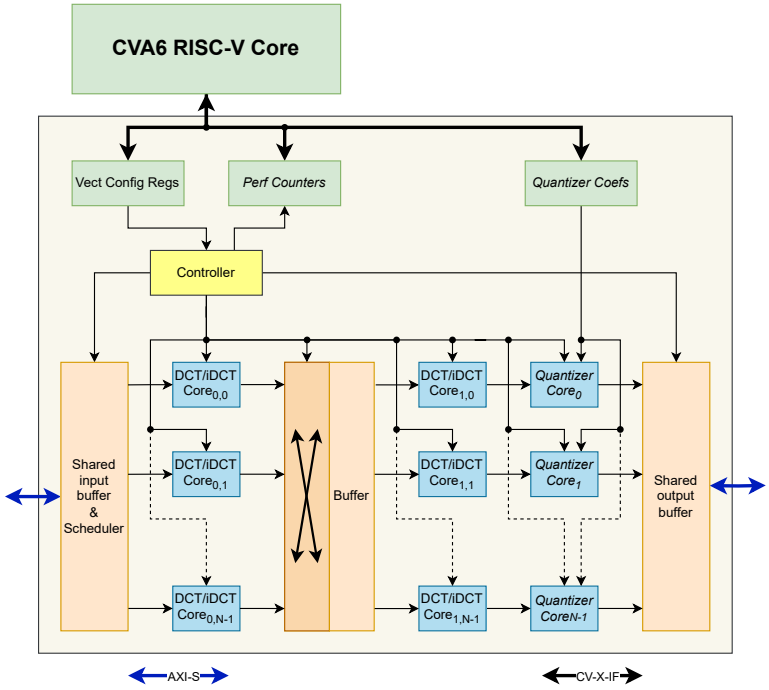

Figure 3.14: Internal architecture of the DCT/iDCT accelerator.

The DCT/iDCT accelerator is a pipelined, vector-driven architecture designed to perform 1D and 2D Discrete Cosine Transforms (DCT) and its inverse, on streaming data using integer arithmetic. At its core, the design comprises multiple parallel DCT/iDCT compute units, each implementing a 8-point 1D DCT/iDCT using fixed-point operations. The input data is received through an AXI4-Stream interface, buffered into N-length vectors (typically N=8), and distributed to available DCT/iDCT cores via a hardware scheduler that supports core masking, bypassing, vector length control, and stride-based block spacing. Each DCT/iDCT core outputs a transformed block, which is post-processed for dynamic scaling. Configuration and control are managed through a CV-X-IF interface that exposes vector length, stride, scaling factors, operation modes, and execution flags to a RISC-V host processor. Alternately, the input data may be received via CV-X-IF interface, packed into source registers.

For two-dimensional DCT/iDCT, the architecture is extended using a two-stage pipeline composed of two identical 1D DCT/iDCT cores separated by an intermediate on-chip transpose buffer. The first DCT/iDCT stage accepts streamed input representing the rows of an N×N block and performs row-wise 1D DCTs/iDCTs. These outputs are written to a local dual-port RAM-based buffer, organized to allow row-to-column transposition without halting the dataflow. Once the entire block is buffered and transposed, the second DCT/iDCT stage processes the columns of the transformed block to complete the 2D DCT/iDCT operation. The final results are streamed out over the AXI4-Stream output port.

To enable observability, tuning, and optimization, the DCT/iDCT accelerator architecture integrates a set of performance counters accessible via the CV-X-IF interface. These counters track key runtime metrics including the number of DCT/iDCT blocks processed, AXI4-Stream input and output packets, per-core utilization, active and idle cycle counts. Each DCT/iDCT core maintains its own usage counter, allowing the RISC-V host to assess load balancing and identify bottlenecks. These performance counters are exposed through dedicated registers, which can be read non-invasively during or after computation.

The DCT/iDCT accelerator is designed for seamless integration with system-level data movement and control infrastructure, featuring direct support for DMA (Direct Memory Access) and interrupt signaling. On the data side, the accelerator interfaces with memory-mapped regions via AXI4-Stream, enabling high-throughput DMA engines. The same AXI4-Stream output path supports streaming results directly into memory or into a subsequent processing pipeline.

To facilitate autonomous operation and event-driven synchronization with the host processor, the accelerator supports interrupt generation through a status output tied to the internal execution controller.

The DCT accelerator architecture is inherently scalable, making it suitable for deployment across a range of hardware platforms from resource-constrained edge FPGAs to high-performance SoCs. Internally, the number of parallel DCT/iDCT compute cores can be configured at synthesis time to match performance and area constraints. Furthermore, the modular design allows for multiple independent instances of the accelerator to be instantiated within a system, each operating on separate data streams or partitioned workloads.

With small structural modifications, the same architectural framework can support other DSP computations such as digital filtering, Fourier transforms, and matrix-vector multiplication. For digital filtering, the MAC-based cores can be reconfigured to implement finite impulse response (FIR) filters, with programmable coefficient sets and circular buffering to support streaming convolution. For spectral analysis and frequency-domain processing, the DCT cores can be replaced or augmented with fixed-point FFT cores, with the same streaming and vector control logic enabling block-wise Fourier transforms. The architecture can be adapted to perform dense matrix-vector multiplications by loading weights as static matrices and streaming feature vectors as inputs.

3.2.7.4 Evaluation

The 1-dimensional DCT and IDCT extension implementations were validated using a custom verification framework designed to ensure correctness - Figure 3.15. This framework employs three distinct black-box testing approaches to assess the functional accuracy of the extensions and measure the required cycles for result generation.

The first method involves running multiple test cases on a Verilator-based simulation testbench. The second method executes the same tests using Xsim, the default simulator provided by Vivado. The third method conducts hardware-in-the-loop testing on an Artix-7 FPGA, where a synthesized testbench communicates with a PC over UART to exchange test vectors and result data.

All test outputs are logged in XML format and compared using a Python script, which determines the final validation verdict.

Accel	DSP	FF	LUT 4	Latency
DCT	12	179	432	8
iDCT	12	405	789	14

Table 3.9: DCT/iDCT Resource usage and latency

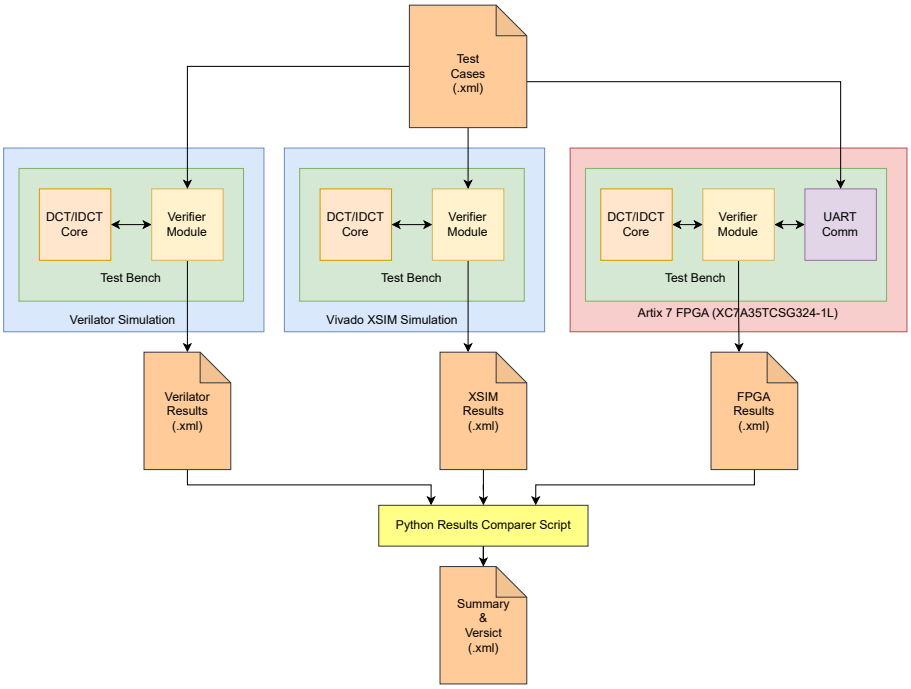

Figure 3.15: Evaluation framework

Other evaluation activity was focused on the integration and performance evaluation of embedded Streaming Hardware Accelerators (eSACs) within RISC-V-based System-on-Chip (SoC) designs [16]. The study aimed to explore architectural strategies that improve data throughput and reduce communication latency between a Central Processing Unit (CPU) and a hardware accelerator using the AXI-Stream protocol. Emphasis was placed on the influence of Direct Memory Access (DMA) configurations and data organization schemes on latency and resource utilization.

Three architecture models were implemented and evaluated as presented in Figure 3.16: Tightly-coupled Streaming, Protocol Adapter FIFO, and Direct Memory Access (DMA) Streaming. The experiments were conducted using the AMD MicroBlaze-V softcore processor integrated into a minimal FPGA-based SoC. The evaluation platform operated at 100 MHz and targeted the Digilent Nexys A7 development board. RTL simulation and an Integrated Logic Analyzer (ILA) were used to gather latency metrics and validate performance.

Key findings indicate that the DMA-based architecture significantly outperforms the others in terms

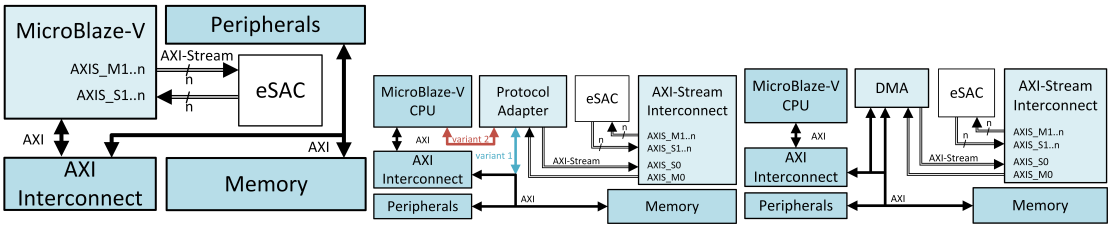

Figure 3.16: Architecture models

of latency, achieving up to a 65% reduction compared to the Tightly-coupled model. The Tightly-coupled and Protocol Adapter approaches, while resource-efficient, suffer from high CPU utilization and limited parallelism, rendering them unsuitable for high-throughput streaming applications. Among the DMA scenarios, data organization emerged as a critical performance factor. The simplest organization (1P1BD) yielded the lowest latency, whereas poor data structuring (e.g., highly fragmented or chaotic buffer descriptor setups) resulted in over a 7× increase in latency for identical data payloads.

3.3 Cryptographic and Security Accelerators

This section introduces hardware accelerators to enhance the cryptographic capabilities and security robustness of RISC-V-based systems, developed as part of the activity of **Task 3.5 – Cryptographic and security accelerators**, led by **SAL**. These IPs are designed to support post-quantum cryptographic primitives, hardware-based key encapsulation mechanisms, and integration with secure RISC-V cores.

The IPs in this category include:

- **ACC-BIKE – Accelerator for Post-Quantum Key Encapsulation Mechanism BIKE (POLIMI)**: A hardware implementation of the BIKE key encapsulation mechanism, optimized for area and power efficiency in post-quantum scenarios.
- **HLS-PQC – HLS-Based Post-Quantum Cryptographic Accelerator (BSC)**: A post-quantum cryptography accelerator implemented via high-level synthesis, allowing for flexible adaptation to evolving algorithmic standards.
- **NTT – Number Theoretic Transform Algorithms for Post-Quantum Cryptography (IMT)**: Two new generic fast NTT algorithms (complex Mersenne NTT and Hensel-Fermat NTT) applicable to all polynomial multiplications in PQC, with software implementation for vector accelerators and hardware implementation as universal NTT accelerators.
- **PQC-MA – Post-Quantum Crypto Accelerator (SAL)**: A general-purpose accelerator targeting various post-quantum algorithms with parameterizable configurations for performance and security trade-offs.
- **SEC – Secured RISC-V Processor with Cryptographic Accelerators (BEIA)**: An enhanced RISC-V core integrated with dedicated cryptographic modules, providing a secure execution environment and hardware-level protection mechanisms.

These components enable ISOLDE platforms to meet emerging requirements for post-quantum security, while offering hardware-accelerated cryptographic performance suitable for embedded, automotive, and IoT applications.

3.3.1 Accelerator for Post-Quantum Key Encapsulation Mechanism BIKE (ACC-BIKE) – POLIMI

3.3.1.1 IP Card

	Basic Info	
IP name	Accelerator for Post-Quantum Key Encapsulation Mechanism BIKE (ACC-BIKE)	
License	Proprietary closed source	
Repository		
	Architecture	
Clock	Number of clock domains	1
	Synchronous with system	Y
	Clock generated internally	N
Ctrl Interface	ISA extension?	N
	Memory mapped?	Y
	Protocol	AXI4 Lite 64b (async)
	Address Map	Base + 0x0 (status register) / Base + 0x1c (address register)
Interrupts	Interrupts	N
	Microarchitecture	
Parametrization	Parametric no. units?	N
	Parameteric config?	N
Programmability	Contains programmable cores?	N
	ISA	N.A.
	Software	
Compiler	Requires specialized compiler?	N
	Compiler repository	https://github.com/pulp-platform/pulp-riscv-gnu-toolchain
Hardware Abstraction Layer	N/A	
High-level API	Is there a high-level API/SDK?	Y
	SDK repository	
	Is there a domain-specific compiler?	N
	Integration	
IP Distribution	Manifest type (if any)	
	Standalone simulation?	Y
	(if standalone sim) SW requirements?	AMD Vivado + QuestaSim
	Integration documented / examples?	
Synthesis	Is the IP synthesizable?	Y
	FPGA synthesis scripts/example available?	Y
	ASIC synthesis scripts/example available?	N
Simulation	Closed-source simulation?	Y (QuestaSim)
	Open-source simulation?	N
Evaluation	PPA results available?	Y, work in progress

3.3.1.2 Purpose

The hardware accelerator implements the BIKE post-quantum cryptosystem, i.e., a cryptographic scheme that can be executed on traditional computers and is secure against both traditional and quantum attacks. BIKE is a code-based key encapsulation mechanism (KEM) that makes use of quasi-cyclic moderate-density parity-check (QC-MDPC) codes. Such QC-MDPC codes are employed in a scheme similar to the well-studied Neiderreiter cryptosystem, which dates to the early 1980s. The public-private keypairs, plaintexts, and ciphertexts of BIKE are represented, due to the quasi-cyclic property of BIKE codes, as binary polynomials with a bitlength in the order of tens of thousands of bits (kbits). Moreover, the moderate-density nature of the underlying code employed by BIKE further eases decoding by leveraging a sparse representation of the polynomials, with a Hamming weight in the order of few hundreds.

A KEM is a cryptographic primitive used to securely establish a shared symmetric key between two parties, typically within a public-key infrastructure. It is particularly useful in hybrid encryption schemes, where the encapsulated key is used with a symmetric encryption algorithm to protect data. A KEM consists of three core algorithms, namely, key generation, encapsulation, and decapsulation, as shown in Figure 3.17. Key generation generates a pair of keys, i.e., a public key

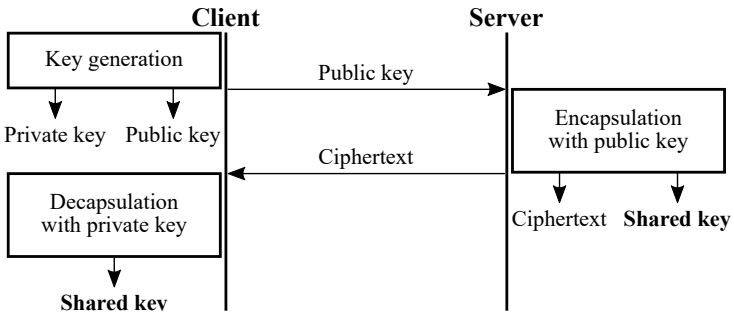

Figure 3.17: High-level diagram of a shared key exchange by means of BIKE.

and a corresponding private key. Encapsulation takes the public key as input and outputs a ciphertext that encapsulates key material and a shared secret. Decapsulation takes the private key and the ciphertext as input, and deterministically recovers the shared secret. KEMs are widely used in securing session keys in protocols such as TLS, and play a foundational role in post-quantum cryptography, where traditional key exchange mechanisms are no longer considered secure against quantum adversaries.

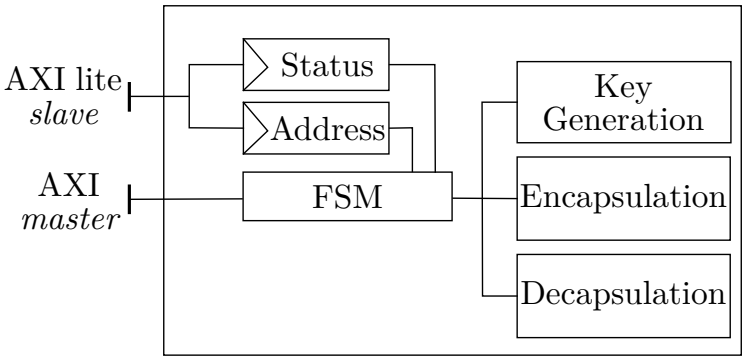

Figure 3.18: Architecture of the BIKE accelerator.

3.3.1.3 Architecture

The BIKE accelerator is designed to be integrated in platforms making use of an AXI interface, and its architecture is depicted in Figure 3.18. The accelerator features two AXI interfaces to facilitate communication with the host processor and memory. The AXI Lite slave interface is responsible for receiving commands from the host processor while the AXI master interface handles memory access to fetch input data and store processed results.

The architecture is composed of three primary functional blocks, respectively devoted to key generation, encapsulation, and decapsulation, that are orchestrated by a finite state machine (FSM), which manages the data flow and operations.

Interfaces The accelerator employs two AXI interfaces to facilitate communication with the host processor and memory.

The AXI Lite slave interface is used for control and configuration. It has a 6-bit address width and a 64-bit data width. The Status register is mapped to `BASE+0x0`, while the Address register is mapped to `BASE+0x1C`. To initiate computation, a starting bit in the Status register is set. The computation completes when the done bit is set in the Status register, and the idle bit remains set. The 3-bit Status register includes a starting bit, that is set to initiate computation, a done bit that is set when computation completes, and an idle bit. The 64-bit Address register stores the address of the input data.

The AXI master interface is responsible for memory access, with a 64-bit address width and a 64-bit data width. Data is shared between the host and the accelerator through a memory region of 8472 bytes. The last byte of this region is used to provide an opcode to select the operation mode, holding values of 0x1, 0x2, and 0x3 for key generation, encapsulation,and ecapsulation, respectively. Additionally, a random seed value must be provided as input and mapped into the shared memory region.

3.3.1.4 Evaluation

The architecture of the BIKE accelerator's prototype has been implemented by leveraging high-level synthesis (HLS) to implement the components for the key generation, encapsulation, and decapsulation primitives. The prototype accelerator is designed to be integrated into a Cheshire SoC that features a CVA6 host processor.

HLS, RTL synthesis and implementation were carried out by means of AMD Vivado, while simulation was performed in QuestaSim. Synthesis and implementation targeted an AMD VCU118 FPGA board.

The prototype implementation of the accelerator occupies an area of 120661 lookup tables, 85146 flip-flops, 114 blocks of block RAM, and 58 DSP elements and operates at a clock frequency of 100MHz.

The key generation, encapsulation, and decapsulation operations take respectively 13621788000, 632168000, and 11821833000 clock cycles.

3.3.2 HLS-Based Post-Quantum Cryptographic Accelerator (HLS-PQC) – BSC

3.3.2.1 IP Card

	Basic Info	
IP name	HLS-Based Post-Quantum Cryptographic Accelerator	
License	Open-source (SolderPad Hardware License v0.51)	
Repository	https://github.com/bsc-loca/PQC-Crystals-HLS-Accelerators.git	
	Architecture	
Clock	Number of clock domains	1
	Synchronous with system	Y
	Clock generated internally	N
Ctrl Interface	ISA extension?	N
	Memory mapped?	Y
	Protocol	AXI4-Lite 64-bit
	Address Map	Base_KEM + 0x0 - 0x48 (ML-KEM) — Base_DSA + 0x0 - 0x60 (ML-DSA)
Initiator Interface	Protocol	AXI4 32-bit
	Cached?	N
	IOMMU?	N
Interrupts	Interrupts	Asynchronous
	Microarchitecture	
Parametrization	Parametric no. units?	Y (default 8 cores)
	Parameteric config?	Y
Programmability	Contains programmable cores?	N
	ISA	RISC-V (ready for Selene RV64GCH)
	Software	
Compiler	Requires specialized compiler?	N
	Compiler repository	
Hardware Abstraction Layer	N/A	
High-level API	Is there a high-level API/SDK?	N
	SDK repository	
	Is there a domain-specific compiler?	N
	Integration	
IP Distribution	Manifest type (if any)	N
	Standalone simulation?	N
	(if standalone sim) SW requirements?	
	Integration documented / examples?	README in https://github.com/bsc-loca/PQC-Crystals-HLS-Accelerators.git
Synthesis	Is the IP synthesizable?	Y
	FPGA synthesis scripts/example available?	Y
	ASIC synthesis scripts/example available?	N
Simulation	Closed-source simulation?	Y (Xilinx Vivado)
	Open-source simulation?	N
Evaluation	PPA results available?	N

3.3.2.2 Purpose

The purpose of the HLS-Based Post-Quantum Cryptographic Accelerator (HLS-PQC) IP is to provide a high-performance and flexible hardware solution for accelerating post-quantum cryptographic schemes, specifically CRYSTALS-Kyber (ML-KEM) and CRYSTALS-Dilithium (ML-DSA). These schemes are among the finalists selected by the NIST PQC standardization process and are designed to withstand quantum computer attacks.

This IP targets integration in RISC-V SoCs, specifically within the SELENE platform (a NOEL-V multicore SoC), to enhance the security capabilities of modern embedded systems. It uses High-Level Synthesis (HLS) to ensure rapid development and portability across hardware targets. Also, it focuses on maximizing throughput, minimizing latency, and supporting concurrent execution through modular and parametric design.

By accelerating the core operations of key encapsulation and digital signatures, the HLS-PQC IP aims to reduce the computational burden on general-purpose cores and meet stringent performance and power requirements for real-world secure applications.

3.3.2.3 Architecture

This IP consists of two accelerators based on High-Level Synthesis (HLS) for post-quantum cryptography (PQC): one designed to accelerate the CRYSTALS-Kyber (ML-KEM) scheme and the other for the CRYSTALS-Dilithium (ML-DSA) scheme. The architecture of these accelerators is modular and loosely coupled, allowing for maximum parallelization and pipelined execution. As illustrated in Figure 3.19, they are designed to be integrated within the SELENE platform, together with the SafeTI and SafeSU described on deliverable D3.2, sections 3.1.8 and 3.2.3, respectively. The interfaces of the accelerators connect to the core through a Network-on-Chip (NoC) using AXI4 protocols.

Control and configuration are handled through AXI-Lite, whereas data transactions occur over dedicated AXI-Full interfaces with 32-bit-wide buses, ensuring efficient data handling. The standalone architecture supports high-frequency operation (up to 500 MHz) and includes a straightforward hardware reset strategy for reliable operation.

The design partitions algorithms into functional modules for concurrent execution, substantially enhancing processing speed. However, addressing the overhead of data fetching remains crucial to fully leveraging the accelerator's performance potential.

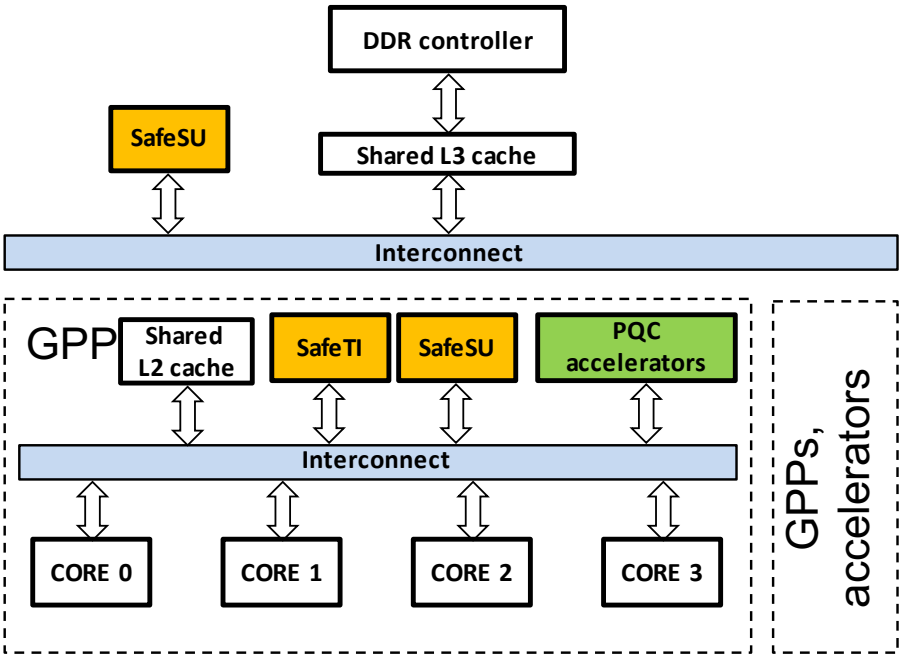

Figure 3.19: HLS-PQC - Place in the SELENE Platform SoC

3.3.2.4 Evaluation

Next, we evaluated the HLS-based accelerator for CRYSTALS-Kyber at the kyber512 security level and CRYSTALS-Dilithium at the Dilithium-3 security level. These accelerators are integrated with the SELENE SoC and implemented on a Xilinx VCU118 FPGA operating at 100 MHz.

For CRYSTALS-Kyber (kyber512), encapsulation operation requires 16.61K cycles (5.14K cycles for scheme computation alone), and decapsulation takes 13.11K cycles (7.06K cycles for scheme computation alone). FPGA resource usage includes 149K LUTs, 129K FFs, 7K BRAMs, and 341 DSP units, consuming 1.66 W of dynamic power.

In the case of CRYSTALS-Dilithium (Dilithium-3 security level, 2430-byte message), signature generation completes in 7.72K cycles (39.75K cycles for scheme computation alone), and verification takes 6.99K cycles (12.31K cycles for scheme computation alone). FPGA resources utilized comprise 174.95K LUTs, 279.09 FFs, 480 BRAMs, and 996 DSP units, with a dynamic power consumption of 1.88 W.

Overall, the results demonstrate significant potential for acceleration, highlighting the necessity of external support mechanisms to enhance the efficiency of the data fetch process.

3.3.3 Number Theoretic Transform Algorithms for Post Quantum Cryptography (NTT) – IMT

3.3.3.1 IP Card

Basic Info		
IP name	Number theoretic transform (NTT) accelerator	
License	Open-source (GPLv3.0)	
Repository	https://github.com/mgologanu/MersenneNTT	
Architecture		
Clock	Number of clock domains	1
	Synchronous with system	Y
	Clock generated internally	N
Ctrl Interface	ISA extension?	N
	Memory mapped?	
	Protocol	AXI
	Address Map	
Initiator Interface	Protocol	AXI
	Cached?	N
	IOMMU?	N
Interrupts	Interrupts	
Microarchitecture		
Parametrization	Parametric no. units?	N
	Parameteric config?	Y
Programmability	Contains programmable cores?	N
	ISA	RISC-V
Software		
Compiler	Requires specialized compiler?	N
	Compiler repository	
Hardware Abstraction Layer	N/A	
High-level API	Is there a high-level API/SDK?	N
	SDK repository	N
	Is there a domain-specific compiler?	N
Integration		
IP Distribution	Manifest type (if any)	N
	Standalone simulation?	only software version
	(if standalone sim) SW requirements?	
	Integration documented / examples?	WIP
Synthesis	Is the IP synthesizable?	WIP
	FPGA synthesis scripts/example available?	WIP
	ASIC synthesis scripts/example available?	N
Simulation	Closed-source simulation?	N
	Open-source simulation?	Y
Evaluation	PPA results available?	Software version using vectorial acceleration

3.3.3.2 Generic NTT Algorithms

We have developed two new generic algorithms for Number Theoretic Transforms (NTT) as used in post-quantum cryptography for multiplying two polynomials in finite rings. We have tested them for correctness in an integrated environment (MATLAB) and for acceleration capabilities in a C implementation targeting vector accelerators for x86 (avx2 and avx512). We have just started the hardware implementation with the help of a new hire.

The proposed algorithms are universal in the sense that they can be used both for NTT friendly and for NTT unfriendly choices of finite polynomial rings, with speeds comparable to the friendly case. They provide a generic accelerator useful for a variety of cryptographic methods.

Let $a(X)$ and $b(X)$ two polynomials in $\mathbb{Z}_q[X]/(P(X))$ where $P(X)$ is a polynomial of degree n, and we need to evaluate their product in the same ring. NTT friendly cases are defined by $P(x) = x^n \pm 1$ and the existence of roots of order n in $\mathbb{Z}_q$. This last condition is equivalent to n being a divisor of the $\Phi(q)$ with Φ Euler's totient function. For q a prime, $\Phi(q) = q - 1$. NTT

unfriendly cases are all other choices for q and $P(X)$.

The approach we use for multiplying two polynomials is well known from signal processing where a linear convolution can be obtained via circular convolutions using FFT, by extending the arrays by zeros to a sufficiently large length, typically a power of 2 for the fastest FFT. Let $k = 2^t > 2(n-1)$ be the closest power of 2 larger than twice the degrees of the two polynomials $a(X)$ and $b(X)$ in $\mathbb{Z}_q[X]/(P(X))$. We will calculate their full product $a(X)b(X)$ in $\mathbb{Z}[X]$, and then reduce the result modulo both q and $P(X)$. To use NTT, we need to work in a new polynomial ring $\mathbb{Z}_r[X]/(X^k - 1)$, with r chosen so that $\Phi(r)$ is divisible by $k = 2^t$ such that roots of unity of order k are available in $\mathbb{Z}_r$. Also, the value r must be large enough to capture the maximal possible coefficient of $a(X)b(X)$ in $\mathbb{Z}[X]$.

The first algorithm (complex Mersenne NTT) is based on the choice $r = p = 2^s - 1$, with p a Mersenne prime, with examples $2^{17} - 1$, $2^{19} - 1$, $2^{31} - 1$, $2^{61} - 1$, etc. At first view this choice is not valid, as $p - 1$ is divisible only by 2 and not by higher powers of 2. We note that the well-known Mersenne transform is used for roots of unity of small and prime order $s = 17, 19, 31, 61...$ for the examples above. On the contrary, our method is based on the existence of *complex* roots of unity up to order 2^{s+1}, appearing in the finite field extension $\mathbb{Z}_p \to \mathbb{F}_{p^2}$, the unique finite field of order p^2. In this case, roots of unity exist for all factors of $p^2 - 1 = (p-1)(p+1) = 2^s(p-1)$. One possible presentation of $\mathbb{F}_{p^2}$, valid only when -1 is not a square in $\mathbb{Z}_q$, is precisely as $\mathbb{Z}_p \bigoplus j\mathbb{Z}_p$ where j is a symbol similar to the imaginary value defined as $j^2 = -1$ mod p. As a consequence, we will have NTT for all powers of 2 up to 2^{s+1}. The price to pay is that we need to consider complex values with complex multiplication and additions. However, all properties of the complex and real FFT are preserved (symmetries, conjugate, etc.), so that we can calculate the complex NTT transform of a real array in place with $2N \log_2 N$ operations (via the split radix algorithm). While this is larger than the $1.5N \log_2 N$ operations for classical NTT without complex values, the comparison reverses when taking into account the *total* number of operations due to modular additions and multiplications, as modular reductions for Mersenne numbers are almost free (a supplementary shift and add for each operation).

The second algorithm (Hensel-Fermat NTT) is based on the choice $r = p^m$, a power of a Fermat prime, with only two convenient choices: $p = 2^8 + 1$ and $p = 2^{16} + 1$. While the Fermat transform is well known and used, the choice of a *power* of p is rather unusual, as the ring $\mathbb{Z}_{p^m}$ is not a field anymore, admitting factors of zero. However, Hensel's lemma lets us extend the existing roots of order 2^8 or 2^{16} in $\mathbb{Z}_p$ to roots in the large ring $\mathbb{Z}_p^m$. However, in order to avoid multiplication and addition of these large numbers, we propose to work with the representation in base p, where each number $x \in \mathbb{Z}_p^m$ is written uniquely as $x = x_0 + x_1 p + x_2 p^2 + ...x_{m-1} p^{m-1}$ and we work directly with the list of m coefficients $x_0, x_1, ..., x_{m-1}$, each being remainder mod p or a number in $\mathbb{Z}_p$. Addition and multiplication for such a representation can be done using additions and multiplications in the small ring $\mathbb{Z}_p$.

Again, modular reduction in $\mathbb{Z}_p$ is very simple (one shift and one addition), without any supplementary multipliers, contrary to the case of arbitrary moduli where one needs Barrett or Montgomery reduction, each using two supplementary multiplications and several shifts and additions/subtractions.

3.3.3.3 Architecture

The proposed architecture is presented in Figure 3.20. It is based on the in-place, decimation in frequency, scrambled output NTT for arrays of length a power of 2. It requires several passes through the array of coefficients, each pass being a radix-8 or radix-4 reduction. At each pass, 8x8=64 values are transferred from memory to the accelerator, and then transferred back after transformation. This configuration permits us to apply NTT to arrays of arbitrary lengths (powers of 2 greater than 64).

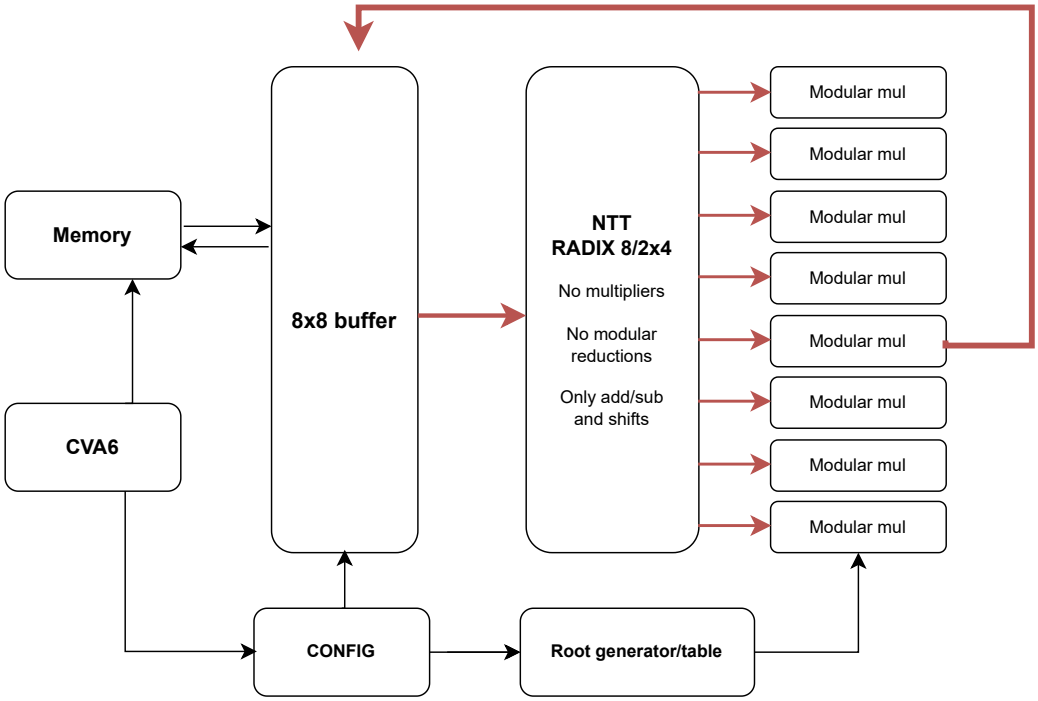

Figure 3.20: NTT accelerator for Mersenne and Hensel-Fermat numbers

The accelerator consists of:

- An addressable buffer for 64 values (8x8) read/written from memory (using contiguous locations in memory).
- One radix-8 block that applies NTT-8 to eight values or one column in the 8x8 buffer. For both Mersenne and Hensel-Fermat algorithms, this block contains only adders, without any multiplier, and with no modular reductions (using lazy reduction, as tested in the C version).
- The same radix-8 block can be used as two parallel radix-4 blocks
- Eight parallel blocks for modular multiplication with roots of unity are located at the exit of the radix-8 block. After this operation, the values are written back to the 8x8 buffer in the original column.

- A root generator or a root table; alternatively roots could also be transferred from memory in a second 8x8 buffer.
- A configurator block that controls the selection of the working column in the buffer, the selection of the relevant roots, and the configuration of the radix block as radix-8 or twice radix-4. It also reports back to the processor when all 8 columns have been evaluated.
- After the application of radix-8 to all columns in the buffer, the values are written back to memory (in-place).
- The evaluation of the columns through the radix block and multipliers can be pipelined. Alternatively, several radix-8 blocks (2/4/8) can be used in parallel to speed-up the transformation of all columns in the buffer.
- A special case appears at the end of the direct NTT, where after passing the 8 columns through the radix block, we need to pass the same values again, not as columns but as rows. This behaviour is also controlled by the configuration block. Alternatively, this step can be delegated to the processor (equivalent to an in place 8x8 transpose).

Due to the in-place, scrambled output of the NTT, we need another accelerator for the inverse NTT. It has the same structure as in Figure 3.20, but with red arrows reversed. For the Mersenne complex NTT, we can use well-known methods for real inputs, either doing two transforms at the same time, or dividing the array in two parts, and using one as real and the other as imaginary components. Alternatively, half of the radix-8 block can be used for real input, as the other half gives conjugate values.

3.3.3.4 Evaluation

The proposed NTT algorithms have been tested only in a software implementation using vectorial accelerators (avx2 and avx512) that mimic the proposed architecture. The 8x8 buffer is assimilated with 8 vectorial registers holding each 8 values. The special case requiring the application of radix-8 to columns and then to rows has been obtained using in-vector shuffles and permutations. Lazy modular reduction has been applied to all additions and subtractions. A single reduction (one shift and add) has been applied after each multiplication with a root of unity. This method does not fully reduce to a remainder below the modulus, but to some value less than some small multiple of the modulus, and will be later absorbed at the next multiplication. Practically, there is no need for comparison operations and if branches. Only at the end of the full polynomial product do we need to fully reduce the coefficients, and this can be done outside the accelerator at the same time with the reduction modulo the original prime q.

We note that the software implementations of the new algorithms have an intrinsic value, as they can easily be ported from avx2/512 to the vectorial accelerators for RISC-V. The scalar versions are also useful for use in embedded processors.

The preliminary results are encouraging. For example, using avx2 acceleration and 64 bit integers, we have obtained for Mersenne NTT with prime $2^{31} - 1$ a number of 1900 cycles for a single NTT of length 512, (as needed for Kyber PQC, a NTT-friendly case), and 9000 cycles for a single NTT of length 2048 (as needed for NTRU Prime, a NTT-unfriendly case). A more compact implementation using 32 bit integers should provide another speed-up factor of 2, as twice as more values can be squeezed in a single register. These results are promising as they are in the same ballpark as published results for NTT-friendly cases, and offer some improvement for NTT-unfriendly ones. However, a hardware implementation should offer much more gains, as the

new algorithms require a much smaller number of multipliers.

We have just started the hardware implementation of the above architecture, with the following cases:

- FPGA – Mersenne prime $p = 2^{17} - 1$, taking advantage of the 18x25 DSP multipliers. This will cover directly some PQC applications (Kyber for $q = 3329$ and module of rank 3, but also NTRU Prime for typical choices of parameters via a simple trick[2].
- FPGA – Hensel-Fermat for the power $r = (2^{16} + 1)^2 \approx 2^{32}$, again taking advantage of the 18x25 DSP multipliers, and covering directly all known NTT cases
- ASIC – Hensel-Fermat for the third power $r = (2^8 + 1)^3 \approx 2^{24}$, where only small multipliers for 8x8 are needed. In this case we have NTT only for lengths up to 256, giving only an incomplete NTT for longer arrays. In this case, the multiplication in the NTT domain is equivalent to the multiplication of two small degree polynomials and this can be done by the CPU, or implemented as a supplementary block. Note that in this case we are not targeting an ASIC but will still use a FPGA implementation.

[2] In all polyomial multiplications $a(X)b(X)$ encountered in PQC, one polynomial $b(X)$ has very small coefficients, while the other can be written as $a(X) = a^{(0)}(X) + 2^t a^{(1)}(X) + 2^{2t} a^{(2)}(X) + ...$, where each polynomial $a^{(i)}(X)$ has also relatively small coefficients. The value of t is chosen so that each product $a^{(i)}(X)b(X)$ can be calculated in the chosen ring $\mathbb{Z}_r[X]$ without overflow, and then the final sum is done directly in the original ring.

3.3.4 Post-Quantum Crypto Accelerator (PQC-MA) – SAL

3.3.4.1 IP Card

Basic Info		
IP name	Accelerator for PQC Primitives	
License	Closed-source	
Repository	Not yet released	
Architecture		
Clock	Number of clock domains	1
	Synchronous with system	Y
	Clock generated internally	N
Ctrl Interface	ISA extension?	N
	Memory mapped?	Y
	Protocol	AXI
	Address Map	TBD
Initiator Interface	Protocol	AXI
	Cached?	N
	IOMMU?	N
Interrupts	Interrupts	Y
Microarchitecture		
Parametrization	Parametric no. units?	Y
	Parameteric config?	Y
Programmability	Contains programmable cores?	N
	ISA	
Software		
Compiler	Requires specialized compiler?	Y
	Compiler repository	standard toolchain
Hardware Abstraction Layer	N/A	
High-level API	Is there a high-level API/SDK?	N
	SDK repository	N
	Is there a domain-specific compiler?	N
Integration		
IP Distribution	Manifest type (if any)	N
	Standalone simulation?	Y
	(if standalone sim) SW requirements?	Vivado or Verilator
	Integration documented / examples?	WIP
Synthesis	Is the IP synthesizable?	Y
	FPGA synthesis scripts/example available?	Y
	ASIC synthesis scripts/example available?	N
Simulation	Closed-source simulation?	N
	Open-source simulation?	Y
Evaluation	PPA results available?	Preliminary results for FPGA

3.3.4.2 Purpose

Our work aims to accelerate the hardware implementation of the Classic McEliece (CM) Key Encapsulation Method (KEM), which is a code-based system and a finalist in the National Institute of Standards and Technology (NIST) efforts to select and standardize PQC algorithms1.

3.3.4.3 Architecture

We base the overall HW design on previous FPGA based implementations of the CM cryptosystem and primitives [17], aiming to accelerate the key functions involved at multiple points in the execution of CM. This gives more flexibility in which algorithms and security levels can be implemented, and easier modification of the overall functionality of the system. The CM cryptosystem defines three main mathematical functions:

1. key generation (KEYGEN) – generates the public and private key pairs from random bits
2. encapsulation (ENCAP) – generates a cipher text and session key from a public key and random bits

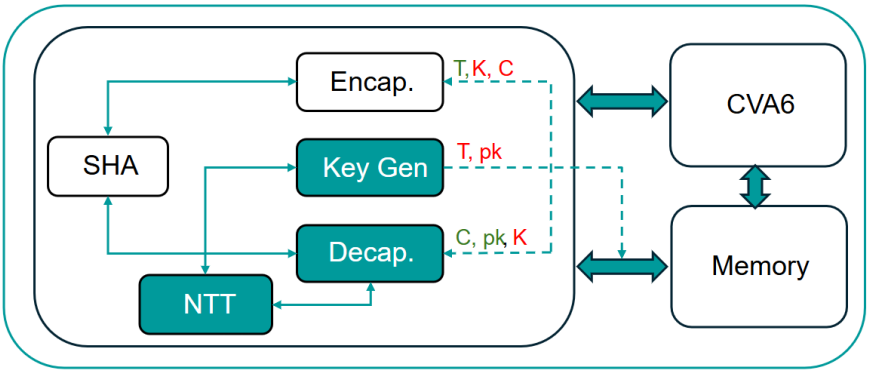

Figure 3.21: SAL Accelerator for PQC Primitives

3. decapsulation (DECAP) – outputs a session key when given a cipher text and a private key.

Interfaces: Our implementation is a memory-mapped system, planned to interface with the OpenHWGroup's CVA6 core, via the AXI-4 bus interface[18]. We also make use of the AXI crossbar and some of the peripherals based on the AXI protocol.

3.3.4.4 Evaluation

We reuse the basic frameworks for the three main modules from the open source implementation in [17], and benchmark the performance of an AMD Virtex™ Ultrascale™ FPGA. While the original implementation is parameterised to generate multiple KEMs, we currently target only the mceliece348864 parameter set, and the first estimates of the performance are summarised in Table. 3.10.

module	LUTs	latency (ms)	time × area
encap	977	0.14	0.13
decap	17109	0.16	2.74
keygen	26674	1.16	30.94

Table 3.10: Performance evaluation based on mceliece348864 parameter set and VCU128 implementation. We are aiming to parameterise and optimise an NTT implementation to suit multiple KEM/parameter sets.

3.3.5 Secured RISC-V Processor with Cryptographic Accelerators (SEC) – BEIA

3.3.5.1 IP card

Basic Info		
IP name	Secured RISC-V Processor with Cryptographic Accelerators (SEC)	
License	Open-source (Apache License v2.0)	
Repository	https://github.com/ISOLDE-Project/AES-256	
Architecture		
Clock	Number of clock domains	1
	Synchronous with system	Y
	Clock generated internally	N
Ctrl Interface	ISA extension?	N
	Memory mapped?	Y
	Protocol	AXI4
	Address Map	0x0000'1000
Initiator Interface	Protocol	AXI4
	Cached?	N
	IOMMU?	N
Interrupts	Interrupts	N/A
Microarchitecture		
Parametrization	Parametric no. cores?	N
	Parameteric config?	Y
Programmability	Contains programmable cores?	Y
	ISA	RISC-V (cv32a6)
Software		
Compiler	Requires specialized compiler?	N
	Compiler repository	-
Hardware Abstraction Layer	N	
High-level API	Is there a high-level API/SDK?	Y
	SDK repository	Not yet available
	Is there a domain-specific compiler?	N
Integration		
IP Distribution	Manifest type (if any)	-
	Standalone simulation?	Y
	(if standalone sim) SW requirements?	-
	Integration documented / examples?	Not yet available
Synthesis	Is the IP synthesizable?	Y
	FPGA synthesis example available?	Y
	ASIC synthesis example available?	N
Simulation	Closed-source simulation?	Y (Vivado simulator)
	Open-source simulation?	N
Evaluation	PPA results available?	N

3.3.5.2 Purpose

The SEC is a secure and modular component for cryptographic processing of IoT data, such as energy management time-series. The data flow starts at the microcontroller, which acquires sensor or grid data and applies AES-256 encryption to ensure confidentiality during transmission. The encrypted data is then transmitted to an FPGA that includes a pre-loaded hardware security accelerator. This component performs high-throughput AES-256 decryption to efficiently handle secure data processing. After decryption, the data is made available to a RISC-V CVA6 softcore running on the FPGA, which hosts a Linux operating system. Within this environment, OpenEMS (Open Energy Management System) operates as a software application, enabling real-time monitoring and control of distributed energy systems. Finally, the data is visualized through the OpenEMS UI, allowing users to monitor and analyze energy data through an interactive graphical interface.

3.3.5.3 Refined Architecture Descripton

The refined architecture represents a secure and modular pipeline for data acquisition, cryptographic processing, and visualization of energy management.

Figure 3.22: BEIA architecture block diagram

The data flow starts firstly at the microcontroller that acquires sensor data or grid data and applies (Advanced Encryption Standard) AES-256 encryption to deliver confidentiality during transmission. The encrypted data is then transmitted to a (field-programmable gate array) FPGA with a pre-loaded hardware security accelerator. The FPGA performs high-throughput AES-256 decryption, freeing the processor from computational burden and enabling real-time processing. The decrypted data are then passed to OpenEMS (Open Energy Management System), an open source real-time monitoring and control platform for distributed energy systems.

3.3.5.4 Interfaces

Within the FPGA, there is an embedded RISC-V soft-core processor that handles inter-module communication, task scheduling, and data exchange between the decryption engine and the OpenEMS runtime. The RISC-V core leverages internal the (Advanced eXtensible Interface) AXI that interconnects to enable low-latency, effective communication between logic blocks. After OpenEMS processing, relevant energy metrics are transferred to a user interface (UI) module—typically executed on a local or remote device—to be visualized. This layer facilitates operators access to dashboards, real-time analytics, and past trends. Scalability and edge deployment are what the architecture facilitates, as well as modular extensions in the shape of additional sensor interfaces, cloud connectivity, or machine learning accelerators, rendering this architecture well-suited for secure, smart energy infrastructure applications.

3.3.5.5 Evaluation Prototype

Prototyping has been performed preliminarily for the following implementations:

- AES_tiny: low program memory footprint at the expense of RAM usage
- AES_small: low RAM footprint at the expense of program memory usage
- AES_ni: using Intel x86's AES-NI dedicated instructions =
- AES_ni_avx2: using Intel x86's AES-NI and AVX2 extensions
- AES_ni_omp: using Intel x86's AES-NI dedicated instructions and multi-core OpenMP implementation

Speed-testing of these functions was made by encrypting and decrypting 1GB of data using CTR mode and the measurements were taken on Asus ROG X13 laptop with Ryzen 5900HS CPU.

3.4 Signal Processing, Neuromorphic and Application-Specific Instruction Set Processors (ASIPs)

This section showcases a range of accelerators tailored to signal processing tasks, neuromorphic computing, and domain-specific processing using customized instruction sets, developed as part of the activity of **Task 3.6 – Signal processing, neuromorphic and application-specific instruction set processors (ASIPs)**, led by **CODA**. These components enhance performance for workloads that demand high throughput, low latency, or specialized functional units.

The IPs included in this category are:

- **FFT – Fast Fourier Transform Algorithms for SIMD and Vector Accelerators (IMT)**: Split radix FFT algorithms for vector accelerators and FFT hardware accelerator integrated with the PolyMem scratchpad memory.
- **LDPC – Low-Density Parity Check Encoder (ACP)**: A hardware encoder for LDPC codes, commonly used in wireless and optical communication systems for robust error correction.
- **Motor Control Accelerator (CODA)**: An application-specific accelerator designed for real-time control of electric motors, suitable for industrial and automotive environments.
- **Neuromorphic HW Accelerator (POLITO)**: A hardware module inspired by the principles of neuromorphic computing, aimed at event-driven processing with ultra-low power consumption.
- **SCA – Shared Correlation Accelerator (ACP)**: An accelerator for performing correlation-based operations in signal processing pipelines, with applications in communication and sensor data analytics.
- **Turbo Decoder (ACP)**: A hardware decoder for turbo codes, commonly used in wireless and optical communication systems for robust error correction.

These accelerators address the needs of specific verticals such as communications, control systems, and bio-inspired computing, and are instrumental in enabling ISOLDE's flexible, high-performance computing platforms.

3.4.1 Fast Fourier Transform Algorithms for SIMD and Vector Accelerators (FFT) – IMT

3.4.1.1 IP Card

Basic Info		
IP name	FFT accelerator	
License	Open-source (GPLv3.0)	
Repository	https://github.com/sirazvan/twister_FFT	
Architecture		
Clock	Number of clock domains	1
	Synchronous with system	Y
	Clock generated internally	N
Ctrl Interface	ISA extension?	Y
	Memory mapped?	N
	Protocol	CV-X-IF
	Address Map	
Initiator Interface	Protocol	CV-X-IF
	Cached?	N
	IOMMU?	N
Interrupts	Interrupts	
Microarchitecture		
Parametrization	Parametric no. units?	Y (default 1 lane)
	Parameteric config?	Y
Programmability	Contains programmable cores?	N
	ISA	RISC-V
Software		
Compiler	Requires specialized compiler?	Y
	Compiler repository	See IP card for Vector-SIMD Accelerator, IMT
Hardware Abstraction Layer	N/A	
High-level API	Is there a high-level API/SDK?	N
	SDK repository	N
	Is there a domain-specific compiler?	N
Integration		
IP Distribution	Manifest type (if any)	N
	Standalone simulation?	N
	(if standalone sim) SW requirements?	
	Integration documented / examples?	WIP
Synthesis	Is the IP synthesizable?	Y
	FPGA synthesis scripts/example available?	WIP
	ASIC synthesis scripts/example available?	N
Simulation	Closed-source simulation?	Vivado
	Open-source simulation?	verilator
Evaluation	PPA results available?	Work in progress

3.4.1.2 Purpose

The Fast Fourier Transform (FFT) is largely used in various fields, from high-performance computing to low power signal processing, with many available software and hardware implementations. The split-radix FFT algorithm uses the least number of operations (multiplications and additions/subtractions), of the order of $4N \log_2 N$, compared to $5N \log_2 N$ for the Cooley-Tukey radix-2 algorithm. However, most software and hardware implementations prefer radix-2 or higher radix algorithms due to their regular structure. We explore the use of the split-radix FFT (in-place, decimation-in-frequency, scrambled output) for SIMD and vectorial accelerators and propose a constant geometry version for hardware implementation. We also explore the integration of the FFT block with the PolyMem scratch-pad memory, developed separately in this work package, to take advantage of the 2D layout and software-defined registers.

3.4.1.3 Algorithms

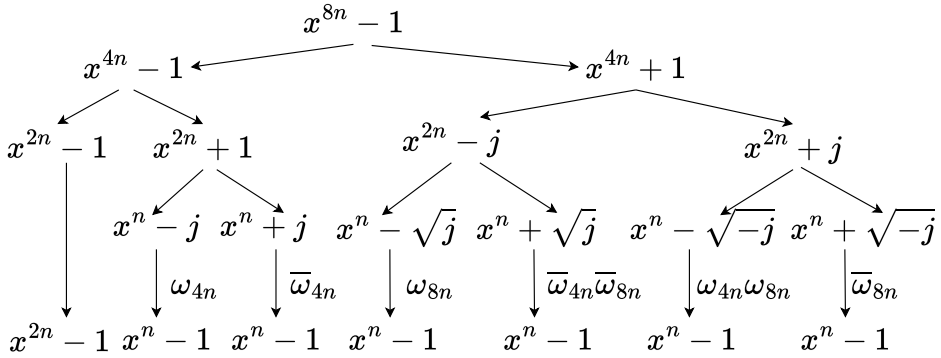

Figure 3.23: Split radix recursive 4/8 algorithm.

We have adapted the split-radix FFT algorithm for use in SIMD and vectorial accelerators. We are using an in-place, decimation-in-frequency, scrambled output, recursive split-radix approach that is able to take advantage of the high number of registers in actual accelerators. As an example, we show in Figure 3.23 the version we have used for avx2 in x86, using a well-known polynomial notation. At each level we calculate from a polynomial remainder modulo $x^{2n} - r^2$ the two polynomial remainders modulo $x^n - r$ and modulo $x^n + r$, equivalent to a butterfly with coefficient r. At the end, we twist each polynomial (a change of variable equivalent to a rotation in the complex plane), where each coefficient is multiplied by a power of some root of unity. This changes the remainder to one modulo $x^n - 1$, to which we can apply the same method recursively, until we reach polynomials of degree 1 which represent precisely the FFT. The inverse FFT (on the scrambled output) can be presented as climbing from lower degree polynomials to larger degrees or the same Figure 3.23, but with all arrows reversed.

In Figure 3.23 we work simultaneously on eight vectors (each with real and imaginary parts) and therefore we fully utilize the 16 vectorial registers present in avx2, doing the maximum possible of calculations that need only these 8 complex values, before writing them back to memory. We also need to load several complex roots that occupy supplementary registers; however, the compiler is able to schedule all instructions without spills (with a single spill for the inverse transform with all arrows in reverse order). A similar approach has been used for avx512 where there are 32 vectorial registers, starting with $x^{32n} - 1$ and going down to a couple of $x^{2n} - 1$ and the rest $x^n - 1$. All these operations are easily vectorized as long as n is larger than the number of values V in a vector register. When attaining this limit, we apply a single $V \times V$ transposition (using in-vector shuffles and permutations) and continue with vectorized calculations down to $x - 1$, or evaluation of the original polynomial at the roots of unity.

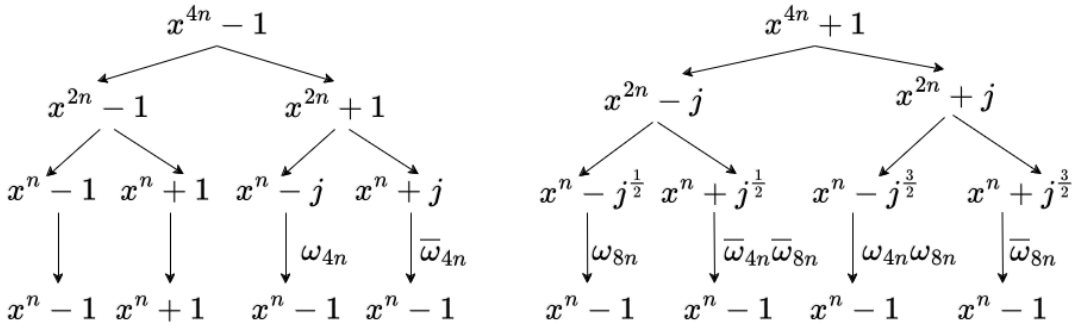

Figure 3.24: Split radix constant geometry algorithm

For a hardware implementation, we have developed a constant geometry version of the split-radix algorithm, shown in Figure 3.24. Like the radix-4 FFT algorithm, we step through two levels at each pass, but globally we use less complex multipliers. However, at each pass we may encounter either $x^{4n}-1$ or $x^{4n}+1$, and Figure 3.24 shows how each is treated. Practically, we need a single block that can be configured for either version. A second advantage is that we can use the same block for real inputs, but continue with only half of the outputs, as the other half is clearly the complex conjugate. The combination of multiplication by a root and its conjugate of two complex numbers has been called a ”twister”; it is similar to a radix-4 butterfly but uses a single root instead of two.

3.4.1.4 Architecture

Our plan is to build a generic FFT block that can be used in multiple cases. For a start, we have chosen a block for FFT16, equivalent to a transformation from $x^{16}-1$ to $16\times(x-1)$. One direct application is presented in Figure 3.25, showing an integration with the PolyMem scratchpad memory for a 16x16 = 256 FFT. We note that this software configurable memory is developed separately in ISOLDE by IMT. The well-known 4-step FFT (extensively used for GPU accelerators) orders the 256 values as a 16x16 matrix, applies FFT16 along all columns, multiplies the matrix point by point with a matrix of roots of unity, and then applies FFT16 along all rows (equivalent to two steps - a transpose and then FFT16 along columns). The use of the software configurable memory is doubly advantageous - the point by point matrix multiplication is already present as an instruction, and there is no need for a matrix transposition, as we can reconfigure directly the memory for row registers. Each column is transferred to our FFT16 block, transformed, and written back in place. To speed up this step, we can either use a single FFT16 block and pipeline all columns, or use several blocks in parallel. Internally, the FFT16 block can be built using either radix-2, radix-4 or split radix, the last with fewer multipliers. It can also be parallelized vertically using a constant geometry version and an internal register. From the software point of view, we need to add a single instruction to the accelerator.

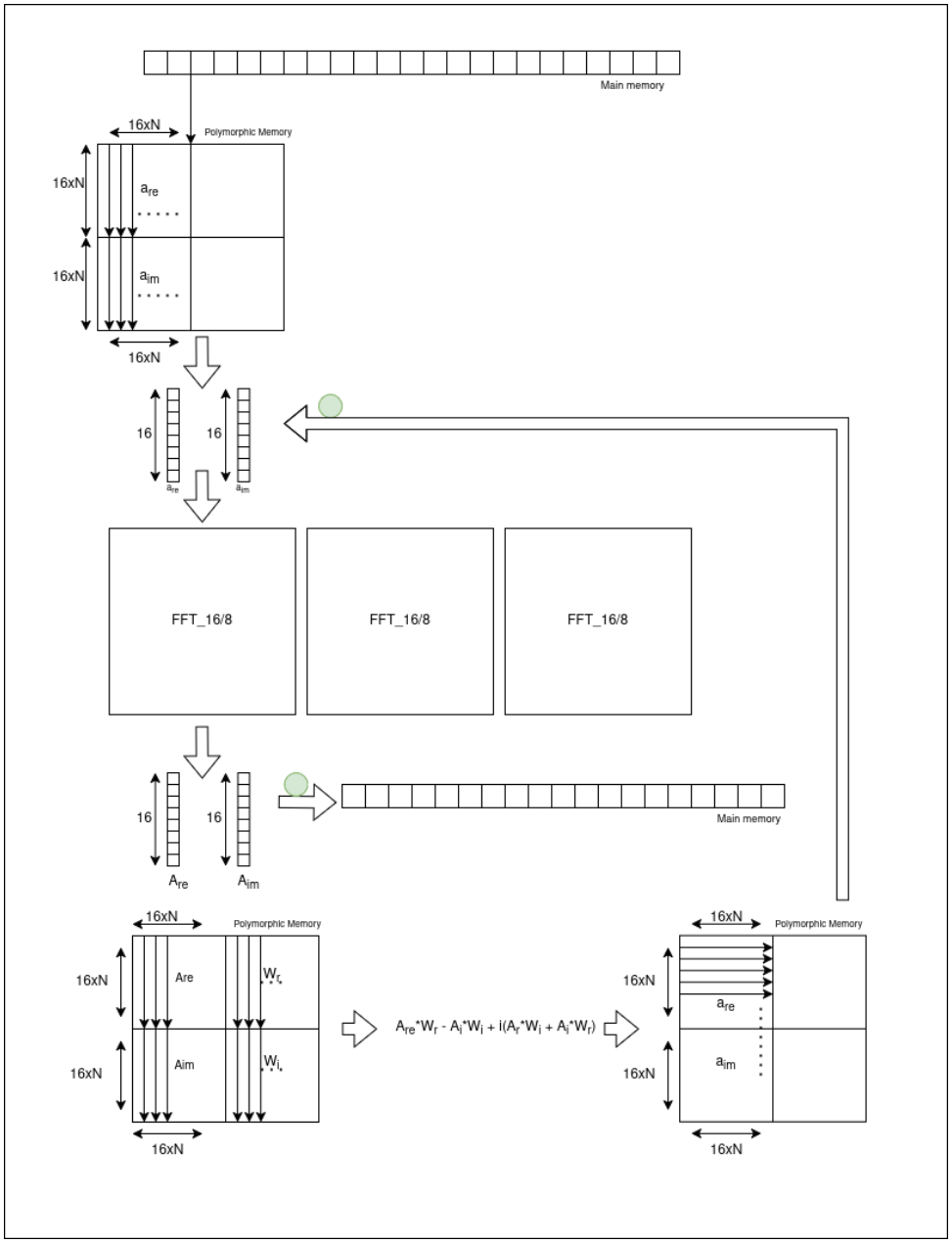

Figure 3.25: FFT accelerator and integration with PolyMeM scratchpad memory

By adding twisters at the exit of FFT16 we can adapt it to treat the reduction from $x^{16n}-1$ to x^n-1 and use it for a radix-16 FFT. Combining this with the 4-step approach gives precisely the case presented in Figure 3.25, with a FFT for a $16N \times 16N$ array. We note also that the FFT16 can

be easily configured as FFT8 by transferring only 8 values to the block and initializing the other 8 as zeros. This provides the possibility of doing $8N_1 \times 16N_2$ or $8N_1 \times 8N_2$ transforms, where the 4-step approach can be applied to a rectangular matrix. Finally, we are able to provide FFT for all lengths powers of 2 starting with 64.

We note that the same FFT16 (or a similar FFT8) block can be used independently from the scratch-pad memory, as shown for the number-theoretic transform in Figure 3.20.

3.4.1.5 Evaluation

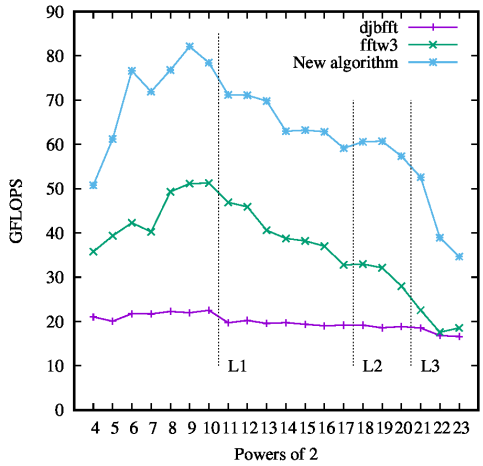

Figure 3.26: Speed comparison for vectorized FFTs (avx2), double, AMD 5800x processor. Lines show the boundaries beyond which the array is larger than the L1, L2 and L3 cache.

We have evaluated the proposed split-radix algorithms in MATLAB and separately in C to explore their vectorizing capabilities using avx2 and avx512[3]. We note that these software implementations have an intrinsic value, as they can easily be ported from avx2/512 to the vectorial accelerators for RISC-V. The scalar versions are also useful for use in embedded processors. Speed results are shown in Figure 3.26 and compared to FFTW, the state of the art open-source FFT implementation (https://www.fftw.org) and djbbft, an old scalar library by D.J. Bernstein (https://cr.yp.to/djbfft.html) which was the inspiration for our vectorized version. The timings are transformed to Giga FLOPS using the methodology proposed by FFTW: $(5N \log_2 N/t$ where t is the running time in nanoseconds. We show mean values averaged over $10^2 - 10^6$ runs (less for large arrays). While the use by FFTW of vector acceleration clearly improves upon the scalar version of djbfft, our version gives a consistent improvement of $1.5 - 2.5\times$ over the full range of lengths.

The RISC-V hardware implementation of the proposed architecture has been started recently by a new colleague and is still a work in progress.

[3]Software available at https://github.com/mgologanu/srfft.

3.4.2 Low Density Parity Check Encoder (LDPC) – ACP

3.4.2.1 LDPC IP card

Basic Info		
IP name	LDPC	
License	proprietary closed source	
Repository		
Architecture		
Clock	Number of clock domains	1
	Synchronous with system	Y
	Clock generated internally	N
Ctrl Interface	ISA extension?	N
	Memory mapped?	Y
	Protocol	APB
	Address Map	TBD
Interrupts	Interrupts	1
Microarchitecture		
Parametrization	Parametric no. units?	N
	Parameteric config?	N
Programmability	Contains programmable cores?	N
	ISA	N.A.
Software		
Compiler	Requires specialized compiler?	N
	Compiler repository	
Hardware Abstraction Layer	N/A	
High-level API	Is there a high-level API/SDK?	N
	SDK repository	
	Is there a domain-specific compiler?	N
Integration		
IP Distribution	Manifest type (if any)	
	Standalone simulation?	Y
	(if standalone sim) SW requirements?	Cadence Insicive
	Integration documented / examples?	
Synthesis	Is the IP synthesizable?	Y
	FPGA synthesis scripts/example available?	N
	ASIC synthesis scripts/example available?	N
Simulation	Closed-source simulation?	Y (Cadence Insicive)
	Open-source simulation?	N
Evaluation	PPA results available?	N

3.4.2.2 Purpose

The 5G New Radio (NR) standard utilizes a low-density parity check (LDPC) code to correct errors that occur during transmission on the wireless link. A device that wants to send an NR uplink (PUSCH) must therefore encode the payload before it is modulated and transmitted. The task of the LDPC encoder is encoding the payload according to the 5G standard, such that the base station can later recover it. The tight latency requirements and single-bit operations make this task well suited for a hardware accelerator.

3.4.2.3 Architecture

The LDPC encoder is designed to encode the NR QC-LDPC code and supports both base graphs which enable it to handle a large range of transport block sizes (TBS) and code rates. It supports all the different lifting sizes Z from 2 to 384 as required by the standard.

Figure 3.27 illustrates the block-level architecture of the LDPC encoder. Uncoded information bits are loaded via a memory interface, passed through the CRC unit that appends a 16 or 24 bit hash, and stored in the systematic section of a local memory (PUSCH MEM). The actual LDPC encoder takes advantage of the underlying structure of the LDPC codes, which consists of subrows and

subcolumns combining a configurable number of Z parity check rows and Z columns, respectively. The encoding process is then performed mainly in a subrow-by-subrow fashion each consisting of a maximum of $Z = 384$ xor-based parity check operations. Therefore, the architecture foresees parallel xor-gates that can calculate up to 384 parity checks in parallel in a streaming fashion. A control unit initiates the load of the next sub-matrix consisting of Z bits sequentially via a 32-bit memory interface, configures the circular-shift unit as specified by the active parity-check base matrix, and finally stores the resulting parity bits into the local memory (PUSCH MEM).

3.4.2.4 Evaluation

A key challenge during the design of the LDPC encoder is the highly configurable circular-shift unit that needs to support varying submatrix sizes between $Z = 2$ and 384 with shift values between 0 and $Z - 1$ in each configuration. Standard barrel-shifter approaches are unsuitable, as multiplexing and routing overhead would drastically inflate silicon area footprint. Instead, a more efficient implementation is targeted that combines two linear shifter circuits resulting in significant area savings.

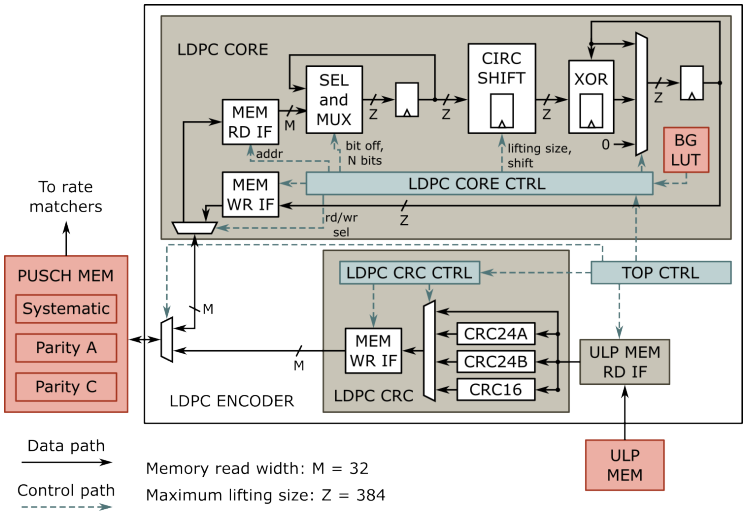

Figure 3.27: High-level diagram of the LDPC encoder.

3.4.3 Motor Control Accelerator – CODA

3.4.3.1 IP Card

	Basic Info	
IP name	Motor Control Accelerator	
License	Proprietary Codasip Licence	
Repository	N/A	
	Architecture	
Clock	Number of clock domains Synchronous with system Clock generated internally	1 Y Y (multiple clock domains generated from main core clock input)
Main Core	RISC-V Core	A730 [19]
Ctrl	ISA extension? Memory mapped? Interface Protocol	RISC-V V 1.0 N Custom accelerator, AXI5
Memory	Interface Protocol Hierarchy Level	128bit AXI5 ($AXI\_WIDTH = 2 * XLEN$) Separate L1 cache with shared L2 cache
Interrupts	Interrupts	Same as for A730
	Microarchitecture	
Parametrization	Parametric no. units? Parameteric config?	One per core N
Programmability	Contains programmable cores? ISA	Y (A730) RISC-V (RV64IMAFDCV)
	Software	
Compiler	Requires specialized compiler?	Y (Automatically generated) or RISC-V compiler compatible with RISC-V V 1.0
High-level API	Is there a high-level API/SDK?	Y - C intrinsics
	Integration	
IP Distribution	Manifest type (if any) Standalone simulation? SW requirements? Integration documented / examples?	No No (A730 Required) QuestaSim Y
Synthesis	Is the IP synthesizable? FPGA synthesis scripts/example available? ASIC synthesis scripts/example available?	Y Work in progress Work in progress
Simulation	Closed-source simulation? Open-source simulation?	Y (QuestaSim) N
Evaluation	PPA results available?	Work in progress

3.4.3.2 Purpose

Codasip is developing a motor control accelerator by customizing one of its existing application cores to better match the specific workload requirements. The target workload is a Model Predictive Control (MPC) library provided by NXP-CZ. An analysis of this implementation, conducted by ISOLDE partner NXP-CZ [20], showed that the MPC application is highly computationally intensive. Further profiling identified the *kernel_sgemm_nt_4x4_lib4* function as the primary performance hotspot, responsible for basic matrix multiplication on 4x4 tiles.

3.4.3.3 Accelerator Architecture

Based on the analysis of the MPC workload, Codasip selected the A730 application core and Codasip Studio EDA as an ideal foundation for a hardware/software co-design approach.

Codasip A730 RISC-V core: Figure 3.28 illustrates the Codasip A730, a 64-bit RISC-V application processor designed for mid-range compute workloads. It features a dual-issue in-order execution and Memory Management Unit (MMU), enabling support for rich operating systems such as Linux. The processor is available in both single-core and multi-core configurations, with

up to four cores per cluster. Designed for flexibility, the A730 is well-suited for power-constrained devices that require efficient execution of complex compute tasks.

The core includes separate instruction and data L1 caches, along with a shared L2 cache. Since the Model Predictive Control (MPC) algorithm is single-threaded, the single-core variant of the A730 was selected for porting the MPC implementation.

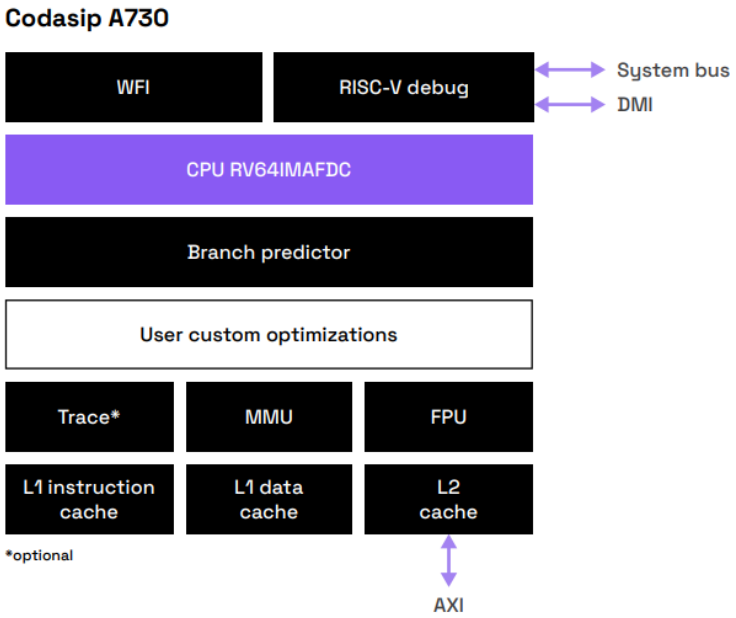

Figure 3.28: Codasip A730 RISC-V core

As part of the ISOLDE project, the Codasip team contributed to the seamless integration of a vector extension [21] into the processor pipeline. This was accomplished through the incorporation of a Vector Register File (VRF), a Vector Sequencer (VSEQ), and a Vector Processing Unit (VPU). The VPU is responsible for executing vector instructions by processing sequences of micro-operations generated by the VSEQ. This modular architecture enables both efficient execution and flexible customization. The tight coupling with the processor pipeline facilitates low-latency data exchange between the VPU and the rest of the core, significantly reducing the need for pipeline stalls when custom instructions are issued.

Codasip A730 VPU components: The VPU in the A730 is composed of the following key computational units:

- **Perm Unit** – Responsible for all bit ”movement” operations. It supports reordering input values or replicating a specific input value across multiple output positions.
- **VALU Unit** – Handles fast vector arithmetic and logical operations, such as bitwise logic, integer addition, and subtraction.
- **VFPU Short Unit** – Performs fast floating-point operations, including conversions between integer and floating-point formats.

- **VFPU Unit** – Executes floating-point vector operations such as addition and subtraction.
- **ITER Unit** – Implements division and square root operations using an iterative algorithm.
- **Multiplication Unit** – Responsible for multiplication and multiply-accumulate operations for both floating-point and integer data types.

It is clear that not all vector instructions can be executed through a single operation by the VPU. For example, RVV instructions such as *vfredosum.vs* require a series of additions to be performed sequentially. The number of required operations depends on configuration parameters such as the vector length multiplier (LMUL) and the data type (e.g., single- or double-precision). This complexity is handled by the Vector Sequencer (VSEQ), which dynamically generates the appropriate sequence of additive µ-operations for execution by the VPU.

In the current implementation, the vector length (VLEN) is set to 128 bits.

Codasip A730 VPU Interfaces: The A730 core has two main interfaces through which it can be integrated into an SoC:

- Data interface is compliant with AXI-5 (128bits) with support for atomics.
- Debug interface provided via DMI with access to the System bus.

The VPU is integrated into the Codasip A730 core through the following interfaces:

- **Memory Interface:** All memory accesses are handled via the Data Cache Unit.
- **Issue Interface:** The decode unit is extended to support new instructions, while maintaining compatibility with the existing pipeline.
- **Forwarding Interface:** Shared across all computation units, it enables efficient result reuse by bypassing the write-back stage.

3.4.3.4 Evaluation

The evaluation process is currently ongoing. To assess the impact of the customization, experiments will be conducted on both the baseline A730 core and the customized version, with their performance results compared. The application used for these experiments is described in Section *Model Predictive Control Demonstrator* of ISOLDE Deliverable D5.2. To support this evaluation, the Codasip team plans to develop an FPGA-based platform for running the core.

3.4.4 Neuromorphic HW Accelerator – POLITO

3.4.4.1 IP Card

Basic Info		
IP name	Neuromorphic HW accelerator	
License	Open-source (SolderPad Hardware License v2.1)	
Repository	Not Yet Released - refer to https://github.com/CHFrenkel/reckon/ for the original design	
Architecture		
Clock	Number of clock domains	1
	Synchronous with system	Y
	Clock generated internally	N
Ctrl Interface	ISA extension?	N
	Memory mapped?	Y
	Protocol	AXI4 64b (Data, commands) / SPI (Configuration)
	Address Map	?
Interrupts	Interrupts?	Y
Microarchitecture		
Parametrization	Parametric Systolic Array shape?	Y
	Parameteric SRAMs?	Y
	Configurable PEs?	Y
Programmability	Contains programmable cores?	Y
Software		
Hardware Abstraction Layer	Are there macros for direct register access?	Y
	Are there HAL functions?	Y
Integration		
IP Distribution	Standalone simulation?	Y
	SW requirements?	QuestaSim
	Integration documented / examples?	Refer to main Cheshire repo from pulp-platform https://github.com/pulp-platform/cheshire
Synthesis	Is the IP synthesizable?	Y
	FPGA synthesis scripts/example available?	Y
	ASIC synthesis scripts/example available?	N
Simulation	Closed-source simulation?	Y (QuestaSim)
	Open-source simulation?	Y
Evaluation	PPA results available?	N

3.4.4.2 Purpose

As Artificial Intelligence (AI) applications are increasingly deployed on resource-constrained devices, there is a growing need for efficient hardware acceleration of Deep Neural Networks (DNNs). One promising approach is the simulation of Spiking Neural Networks (SNNs) on dedicated neuromorphic hardware, which can emulate brain-like processing of time-varying signals and enable real-time inference on input data.

The key strength of SNN hardware accelerators is the small amount of resources required to perform parallel computation on data, enabling brain-inspired computation at the edge and on IoT devices.

In the next section, we show how a neuromorphic accelerator can be integrated with a RISC-V-based processor using the AXI bus to bring edge AI to heterogeneous systems.

3.4.4.3 Architecture

The architecture of the SoC depicted in Figure 3.29 is composed of four main elements:

1. The Cheshire processor, a Linux-capable open-source processor built around the RISC-V core CVA6. It is provided with peripherals like SPI, UART, VGA, and the AXI controller.

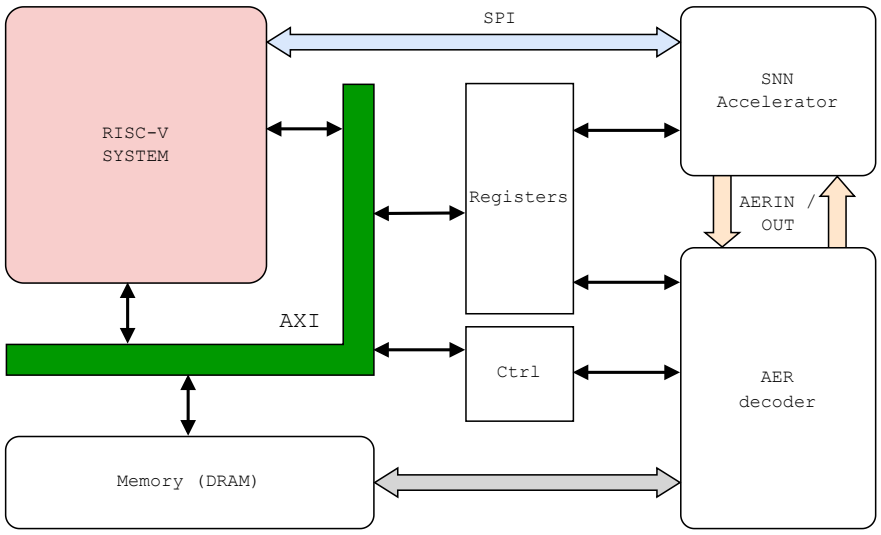

Figure 3.29: Neuromorphic HW accelerator architecture.

2. The AXI subsystem, which collects the AXI-accessible peripherals used to communicate between the host and the accelerator: a Register File that exposes some registers for run-time configuration during training or inference; a Control Interface used to communicate with the Address Event Representation (AER) decoder. In addition, the external on-board DDR memory can also be accessed with the EXI bus from the main SoC.
3. The AER decoder, the binding element between the controllers and the SNN accelerator, is responsible for decoding the 32-bit words stored in the buffer memory into the related input spikes and sending them to ReckOn. At the end of each sample, the per-epoch accuracy is updated during training and validation by reading the inferred results generated by the network. The resulting accuracy is finally transferred to the main host through the AXI Register File.
4. The neuromorphic accelerator, open-source project ReckOn [4]. The accelerator runs a Recurrent Spiking Neural Network with an embedded weight update mechanism based on the e-prop algorithm that allows on-chip training. Weights and network status are stored in internal memories implemented in BRAMs.

3.4.4.4 Evaluation

The evaluation process is currently underway. In order to assess the impact of the IP, experiments will be conducted on various Edge AI tasks (listed below) on both the baseline version implemented on the ARM core and the Cheshire version, and the performance results will be compared.

[4] https://github.com/ChFrenkel/reckon

Performance

- Evaluation environment: Architecture design implementation on ZCU102 development board.
- Clock Frequency: Accelerator runs at 15 MHz

Edge-AI tasks

- Benchmark on a binary navigation classification dataset.
- Test on Braille handwritten digits recognition (7 classes).
- Final demonstration on ESA anomaly detection for spacecraft predictive maintenance.

3.4.5 Shared Correlation Accelerator (SCA) – ACP

3.4.5.1 SCA IP card

Basic Info		
IP name	Shared Correlation Accelerator (SCA)	
License	proprietary closed source	
Repository		
Architecture		
Clock	Number of clock domains	1
	Synchronous with system	Y
	Clock generated internally	N
Ctrl Interface	ISA extension?	N
	Memory mapped?	Y
	Protocol	APB
	Address Map	0x000: Control 0x004 to 0x17c: Configuration
Initiator Interface	Protocol	PULP TCDM
	Cached?	N
	IOMMU?	N
Interrupts	Interrupts	1
Microarchitecture		
Parametrization	Parametric no. units?	N
	Parameteric config?	N
Programmability	Contains programmable cores?	N
	ISA	N.A.
Software		
Compiler	Requires specialized compiler?	N
	Compiler repository	
Hardware Abstraction Layer	N/A	
High-level API	Is there a high-level API/SDK?	N
	SDK repository	
	Is there a domain-specific compiler?	N
Integration		
IP Distribution	Manifest type (if any)	
	Standalone simulation?	Y
	(if standalone sim) SW requirements?	Cadence Insicive
	Integration documented / examples?	
Synthesis	Is the IP synthesizable?	Y
	FPGA synthesis scripts/example available?	Y
	ASIC synthesis scripts/example available?	Y
Simulation	Closed-source simulation?	Y (Cadence Insicive)
	Open-source simulation?	N
Evaluation	PPA results available?	Y

3.4.5.2 Purpose

In wireless communication, known sequences are commonly used to facilitate initial synchronization as a first step towards establishing a communication link. The receiving device then attempts to find these known sequences in the received signal. The maximum-likelihood approach for this task involves computing many cross-correlations with the known sequences for multiple frequency offset hypotheses. The shared correlation accelerator (SCA) enables this approach by efficiently computing these correlations at high throughput.

3.4.5.3 Architecture

Figure 3.30 shows the architecture of the SCA. The SCA primarily relies on a streaming length-2048 FFT unit. It is used to transform both the correlation sequences and the received samples to the frequency domain. Correlations with multiple frequency offset hypotheses can easily be computed by circularly shifting the correlation sequence in frequency domain. The correlation is computed using a simple multiplication in the frequency domain. Afterwards, the FFT is used

again, in inverse mode, to transform the correlation results back to time domain. The final accumulation block decimates the time-domain correlation results and accumulates them. As the memory capacity requirement for the correlation values is large, it is stored in the TCDM. This allows the memory to be time shared with other functions. A sample buffer at the input stores the incoming samples and absorbs the uneven consumption rate of the correlator. It also implements filtering, decimation, and amplitude normalization to reduce the fixed-point width within the accelerator.

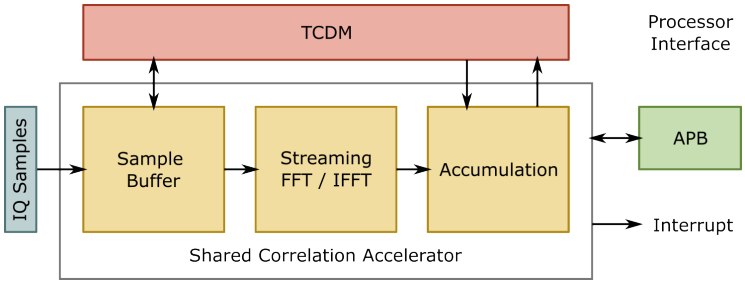

Figure 3.30: High-level diagram of the SCA.

3.4.5.4 Evaluation

The accelerator has been integrated into a modem SoC. The SoC has been implemented on an FPGA and synthesized in 22 nm CMOS.

FPGA

- Evaluation platform:
  - AMD Versal Prime VMK180 Board with two custom expansion PCBs
  - Implemented at reduced 100 MHz clock
    - Live operation possible with reduced number of frequency hypotheses.
  - Resource utilization: 9.3k FFs, 18k LUTs
  - Bring-up completed with RISC-V controller CPU (Ibex)
  - Successful cell detection on RF signal

ASIC

- CMOS technology: 22 nm
- Clock frequency: 400 MHz (worst-case corner)
- Synthesis area: 0.2 mm^2 = 520 kGE

3.4.6 Turbo Decoder – ACP

3.4.6.1 Turbo IP card

Basic Info		
IP name	Turbo Decoder	
License	proprietary closed source	
Repository		
Architecture		
Clock	Number of clock domains	1
	Synchronous with system	Y
	Clock generated internally	N
Ctrl Interface	ISA extension?	N
	Memory mapped?	N
	Protocol	N.A.
	Address Map	N.A.
Initiator Interface	Protocol	PULP TCDM
	Cached?	N
	IOMMU?	N
Interrupts	Interrupts	1
Microarchitecture		
Parametrization	Parametric no. units?	N
	Parameteric config?	N
Programmability	Contains programmable cores?	N
	ISA	N.A.
Software		
Compiler	Requires specialized compiler?	N
	Compiler repository	
Hardware Abstraction Layer	N/A	
High-level API	Is there a high-level API/SDK?	N
	SDK repository	
	Is there a domain-specific compiler?	N
Integration		
IP Distribution	Manifest type (if any)	
	Standalone simulation?	Y
	(if standalone sim) SW requirements?	Cadence Insicive
	Integration documented / examples?	
Synthesis	Is the IP synthesizable?	Y
	FPGA synthesis scripts/example available?	Y
	ASIC synthesis scripts/example available?	Y
Simulation	Closed-source simulation?	Y (Cadence Insicive)
	Open-source simulation?	N
Evaluation	PPA results available?	Y

3.4.6.2 Purpose

Wireless communication standards employ forward error correction to correct errors that occur during the transmission. The main data channel in LTE uses a turbo code, which can achieve excellent error-correcting performance. The purpose of the turbo decoder is to iteratively process the soft information from the demodulator and recover the transmitted code block. If the data was decoded successfully, it is then forwarded to the upper protocol levels, otherwise, the base station will attempt a re-transmission.

3.4.6.3 Architecture

Figure 3.31 shows the architecture of the turbo decoder. The turbo decoder receives soft information on the received bits in the form of LLRs. It then processes them iteratively, in multiple half-iterations, to recover the transmitted data. In each half-iteration, the LLRs are first read from the input and extrinsic LLR memory. After preprocessing, the decoder first computes the path and secondly the state metrics for each input bit. To limit the internal memory requirements, this processing is performed in local windows of 64 bits. Then, the unit derives the updated extrinsic

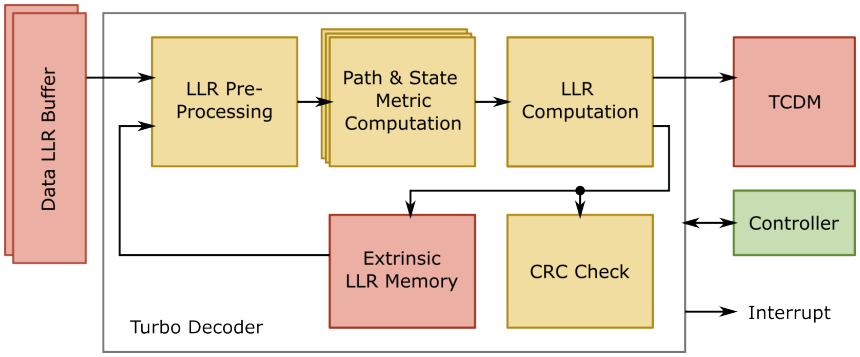

Figure 3.31: High-level architecture diagram of the turbo decoder.

LLRs from the state metrics and writes them back to the extrinsic memory. Simultaneously, the hard decoded bits are computed and forwarded to the CRC check unit. If the CRC matches, or the maximum number of half-iterations has been reached, the decoded bits are written to the TCDM for further use by the RISC-V processor.

3.4.6.4 Evaluation

The turbo decoder has been integrated into a cellular modem SoC. The SoC has been implemented on an FPGA and synthesized in 22 nm CMOS.

FPGA

- Evaluation platform:
  - AMD Versal Prime VMK180 Board with two custom expansion PCBs
  - Implemented at reduced 100 MHz clock
    - Live operation possible with lower number of half-iterations.
  - Resource utilization: 2.1k FFs, 10k LUTs
  - Bring-up completed with RISC-V controller CPU (Ibex)
  - Successful decoding of data channel from RF signal

ASIC

- CMOS technology: 22 nm
- Clock frequency: 400 MHz (worst-case corner)
- Synthesis area: 0.07 mm^2 = 190 kGE

4 Conclusion

This deliverable has presented the prototype implementations of the hardware accelerators and extensions developed within Work Package 3 of the ISOLDE project. The described IPs span a broad range of domains—including arithmetic units, AI/ML and vector accelerators, cryptographic modules, and application-specific processors—demonstrating both architectural diversity and technical maturity.

Each section has documented the functional scope, integration readiness, and preliminary evaluation results of the IPs, supporting their continued refinement and validation. These contributions represent a key step toward full system integration in WP5 demonstrators.

Deliverable D3.3 (together with D3.2) forms the basis for the upcoming finalization and optimization phases in D3.4 and D3.5.

Acronyms and Definitions

Acronym	Description
ACC-BIKE	ACCelerator for post-quantum key encapsulation mechanism BIKE
ADC	Analog-to-Digital Converter
AES	Advanced Encryption Standard
AHB	Advanced High-performance Bus
AI	Artificial Intelligence
ALU	Arithmetic Logic Unit
AMA	AI/ML Accelerator
AMBA	Advanced Microcontroller Bus Architecture
APB	Advanced Peripheral Bus
ASCON	Lightweight authenticated block cipher
ASIC	Application-Specific Integrated Circuit
ASIP	Application-Specific Instruction Set Processor
ASLR	Address Space Layout Randomization
AXI	Advanced eXtensible Interface
AXI-MM	AXI Memory Mapped
AXIS	AXI Stream
BCFI	Backward-Edge Control Flow Integrity
BRAM	Block RAM
BS	Base Station
CA-PMC	Context-Aware Performance Monitor Counter
CA-PMC-IF	Context-Aware PMC Interface
CBD	Contract Based Design
CCS	Contention Cycles Stack
CE	Computing Element
CFI	Control Flow Integrity
CMOS	Complementary Metal-Oxide Semiconductor
CNN	Convolutional Neural Network
CORDIC	Coordinate Rotation Digital Computer
COP	Call-Oriented Programming
CPU	Central Processing Unit
CPS	Cyber-Physical Systems
CRC	Cyclic Redundancy Check
CSR	Control and Status Register
CTM	Cryptographically Tagged Memory
CV-X-IF	Core-V eXtension Interface
DBB	Digital Base Band
DDR	Double Data Rate Synchronous Dynamic Random Access Memory
DES	Data Encryption Standard

DFT	Discrete Fourier Transform
DFU	Decoder Functional Units
DMA	Direct Memory Access
DMR	Dual Modular Redundancy
DSP	Digital Signal Processor
DSS	Digital Signature Schemes
DVS	Dynamic Vision Sensor
ECC	Error Correction Code
EMI	Enclave Memory Isolation
ECNNA	Event-based CNN Accelerator
EXP	EXtension Platform
FCFI	Forward-edge Control Flow Integrity
FFT	Fast Fourier Transform
FP	Floating Point
FPGA	Field Programmable Gate Array
FSM	Finite State Machine
FIFO	First-In-First-Out
FIR	Finite Impulse Response
FMA	Fused-Multiply-Add
FPMIX	FPU for MIXed-precision computing
FPU	Floating Point Unit
GEMM	GEneral Matrix Multiply
GPIO	General Purpose Input/Output
HARQ	Hybrid Automatic Repeat Request
HCI	Heterogeneous Cluster Interconnect
HDK	Hardware Development Kit
HLS	High Level Synthesis
HLS-PQC	HLS-based Post-Quantum Cryptographic accelerator
HMAC	Hash-based Message Authentication Code
IEE	Inline Encryption Engine
IEE-RV	Inline Encryption Engine RISC-V ISA extension
INET	Interconnection NETwork
INTT	Inverse Number Theoretic Transform
IP	Intellectual Property
ISA	Instruction Set Architecture
ISE	Instruction Set Extension
IUHF	Inverse Universal Hash Function
JOP	Jump-Oriented Programming
KEM	Key Encapsulation Mechanism
KMAC	KECCAK Message Authentication Code
LDPC	Low Density Parity Check Decoder
LIF	Leaky Integrate and Fire (neuron model)
LSW	Least Significant Word
LLR	Log Likelihood Ratio

M	Machine Mode
MAC	Multiply-Accumulate
MC	Memory Controller
MCCU	Maximum Contention Control Unit
MDPC	Moderate-Density Parity-Check
ML	Machine Learning
ML-DSA	Module-Lattice-based – Digital Signature Standard
ML-KEM	Module-Lattice-based – Key Encapsulation Mechanism
MMIO	Memory Mapped Input/Output
MMU	Memory Management Unit
MPSoC	Multiprocessor System on a Chip
MSW	Most Significant Word
NIST	National Institute of Standards and Technology
NoC	Network on Chip
NR	New Radio
NTT	Number Theoretic Transform
ONNX	Open Neural Network eXchange
OVI	Open Vector Interface
PC	Program Counter
PCA	Parallel Computing Accelerator
PE	Processing Engine
PMP	Physical Memory Protection
PMU	Performance Monitor Unit
POR	Power-On Reset
PPA	Power, Performance, and Area
PQC	Post-Quantum Cryptography
PQC-MA	Post-Quantum Crypto Accelerator
PRF	Polymorphic Register File
PRINCE	Low-latency block cipher
PRNG	Pseudorandom Number Generator
QC	Quasi-Cyclic
QUARMAv2	Lightweight tweakable block cipher
RDC	Request Duration Counter
ReO	Rectangle Only
ReRo	Rectangle Row
ReTr	Rectangle Transposed
RF	Radio Frequency
RFOG	Register File Organization Table
RoCo	Row Column
ROM	Read-Only-Memory
ROP	Return Oriented Programming
RoT	Root-of-Trust
RSA	Rivest–Shamir–Adleman
RTL	Register Transfer Level

RTPM	Run-Time Power Monitoring instrumentation
RV32	32-bit RISC-V processor model
RVV	RISC-V Vector extension
S	Supervisor Mode
SafeSU	Safety-related Statistics Unit
SafeTI	Safety-related Traffic Injector
SCA	Shared Correlation Accelerator
SCH	SCHeduler
SCMI	System Control and Management Interface
SDK	Software Development Kit
SDRAM	Synchronous Dynamic Random Access Memory
SEC	SECured RISC-V processor with cryptographic accelerators
SHA	Secure Hash Algorithms
SIMD	Single Instruction Multiple Data
SLDU	SLiDe Unit
SLH-DSA	Stateless Hash-Based Digital Signature Standard
SM	Security Monitor
SNN	Spiking Neutral Networks
SoA	State of the Art
SoC	System on a Chip
SPI	Serial Peripheral Interface
SRAM	Static Random-Access Memory
TBS	Transport Block Sizes
TCCP	Time Contract monitoring Co-Processor
TCCP-CO	Time Contract monitoring Co-Processor COmpiler
TCDM	Tightly-Coupled Data Memory
TI	Tweak Input
TLUL	TileLink Uncached Lightweight bus
TMR	Triple Modular Redundancy
TPU	Tensor Processing Unit
U	User Mode
UE	User Equipment
UHF	Universal Hash Function
IUHF	Inverse Universal Hash Function
VLSI	Very Large Scale Integration
VMFPU	Vector Multiplier and Floating-Point Unit
VPU	Vector Processing Unit
VRF	Vector Register File
WCET	Worst-Case Execution Time
XIF	eXtension InterFace

Bibliography

[1] S. Mach, F. Schuiki, F. Zaruba, and L. Benini, “Fpnew: An open-source multiformat floating-point unit architecture for energy-proportional transprecision computing,” *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 29, no. 4, pp. 774–787, 2020.

[2] Synopsys, “Vcs: Functional verification solution,” 2025, accessed: 2025-04-09. [Online]. Available: https://www.synopsys.com/verification/simulation/vcs.html

[3] ——, “Z01x functional safety assurance,” 2025, accessed: 2025-04-09. [Online]. Available: https://www.synopsys.com/verification/simulation/z01x-functional-safety.html

[4] C. B. Ciobanu, G. Stramondo, C. de Laat, and A. L. Varbanescu, “Max-polymem: high-bandwidth polymorphic parallel memories for dfes,” in *2018 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*. IEEE, 2018, pp. 107–114.

[5] C. B. Ciobanu, *Customizable Register Files for Multidimensional SIMD Architectures*. Delft University of Technology, 2013. [Online]. Available: https://resolver.tudelft.nl/uuid:6da2ee07-99df-450d-93bd-2367725f4f70

[6] G. Kuzmanov, G. Gaydadjiev, and S. Vassiliadis, “Multimedia rectangularly addressable memory,” *IEEE Transactions on Multimedia*, vol. 8, no. 2, pp. 315–322, 2006.

[7] (2022, 05) Vcu128 evaluation board user guide (ug1302). AMD. [Online]. Available: https://docs.amd.com/r/en-US/ug1302-vcu128-eval-bd

[8] J. Fornt, P. Fontova-Musté, M. Caro, J. Abella, F. Moll, J. Altet, and C. Studer, “An energy-efficient gemm-based convolution accelerator with on-the-fly im2col,” *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 31, no. 11, pp. 1874–1878, 2023.

[9] F. Guella, E. Valpreda, M. Caon, G. Masera, and M. Martina, “Temet: Truncated reconfigurable multiplier with error tuning,” in *Applications in Electronics Pervading Industry, Environment and Society*, F. Bellotti, M. D. Grammatikakis, A. Mansour, M. Ruo Roch, R. Seepold, A. Solanas, and R. Berta, Eds. Cham: Springer Nature Switzerland, 2024, pp. 370–377.

[10] Y. Tortorella, L. Bertaccini, L. Benini, D. Rossi, and F. Conti, “RedMule: A mixed-precision matrix–matrix operation engine for flexible and energy-efficient on-chip linear algebra and TinyML training acceleration,” *Future Generation Computer Systems*, vol. 149, pp. 122–135, Dec. 2023.

[11] A. Belano, Y. Tortorella, A. Garofalo, L. Benini, D. Rossi, and F. Conti, “A Flexible Template for Edge Generative AI with High-Accuracy Accelerated Softmax & GELU,” Dec. 2024.

[12] M. Perotti, M. Cavalcante, R. Andri, L. Cavigelli, and L. Benini, “Ara2: Exploring Single- and Multi-Core Vector Processing With an Efficient RVV 1.0 Compliant Open-Source Processor,” *IEEE Transactions on Computers*, vol. 73, no. 7, pp. 1822–1836, Jul. 2024.

[13] OpenHW Group. OpenHW Group Specification: Core-V eXtension interface (CV-X-IF). [Online]. Available: https://github.com/openhwgroup/core-v-xif

[14] ——. CVA6 RISC-V CPU. [Online]. Available: https://github.com/openhwgroup/cva6

[15] A. Puşcaşu, C. B. Ciobanu, and O. Buiu, "Systolic array matrix multiplication accelerator," in *2024 International Semiconductor Conference (CAS)*. IEEE, 2024, pp. 207–210.

[16] C.-T. Axinte, A. Stan, and V.-I. Manta, "Embedded streaming hardware accelerators interconnect architectures and latency evaluation," *Electronics*, vol. 14, no. 8, 2025. [Online]. Available: https://www.mdpi.com/2079-9292/14/8/1513

[17] P.-J. Chen, T. Chou, S. Deshpande, N. Lahr, R. Niederhagen, J. Szefer, and W. Wang, "Complete and improved fpga implementation of classic mceliece," *IACR Transactions on Cryptographic Hardware and Embedded Systems*, p. 71–113, Jun. 2022. [Online]. Available: https://tches.iacr.org/index.php/TCHES/article/view/9695

[18] (2023, nov) Cva6: An application class risc-v cpu core. OpenHW Group. [Online]. Available: https://docs.openhwgroup.org/projects/cva6-user-manual/index.html

[19] Codasip, "A730 Product Brief," 2023. [Online]. Available: https://codasip.com/wp-content/uploads/2023/10/Product-Brief-A730-2023-EN.pdf

[20] M. Kostal, "Model Predictive Control Acceleration on RISC-V CPU," 2025.

[21] K. Asanovic, "RISC-V "V" Vector Extension." [Online]. Available: https://github.com/riscvarchive/riscv-v-spec/releases/tag/v1.0