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	**D3.1**
Title	**Initial Architecture Definition**
Type of deliverable:	Report
Deadline:	30.04.2024
Creation date:	09.01.2024
Dissemination Level:	PU - Public

Authors: Marcus Borrmann, Erik Kraft, Marcel Medwed - NXP-AT

Daniel Gracia Pérez, Sylvain Girbal - TRT

Stefan Wallentowitz, Matthias Rupp - HM

Holger Blasum - SYSGO

Jaume Abella, Francisco Fuentes, Sergi Alcaide, Francisco J. Cazorla, Ramon Canal, Feng Chang, Juan Carlos Rodríguez, Juan Antonio Rodríguez, Xavier Carril - BSC

Esther Soriano, Vicente Nicolau - FENTISS

Wolfgang Ecker, Endri Kaja, Jad Al Halabi - IFX

Mladen Berekovic, Christopher Blochwitz - UZL

Emanuele Parisi, Francesco Conti, Yvan Tortorella - UNIBO

Andrea Galimberti - POLIMI

Alexandru Pușcașu, Cătălin Ciobanu, Mihai Gologanu - IMT

Behnam Razi Perjikolaei - OFFIS

Mihai Munteanu, Honorius Galmeanu - FotoNation

Diego Gigena Ivanovich, Ambily Suresh, Andrew Wilson, Manuel Freiberger - SAL

Maurizio Martina, Gianvito Urgese - POLITO

Matteo Perotti - ETHZ

Andrei Stan, Cristian-Tiberius Axinte, George Uleru - TUI
Mari-Anais Sachian, Kejsi Koci, Cristina Tudor, George Suciu - BEIA
Michael Gautschi, Reza Ghanaatian, Stefan Lippuner - ACP
Jan Kaštil - CODA

Involved grant recipients:
NXP Semiconductors Austria GmbH & Co KG (NXP-AT)
Thales SA as Thales Research & Technology (TRT)
Hochschule München University of Applied Sciences (HM)
SYSGO GmbH (SYSGO)
Barcelona Supercomputing Center (BSC)
Fent Innovative Software Solutions S.L. (FENTISS)
Infineon Technologies AG (IFX)
Universität zu Lübeck (UZL)
Universita di Bologna (UNIBO)
Politecnico di Milano (POLIMI)
Institutul National de Cercetare-Dezvoltare Pentru Microtehnologie (IMT)
OFFIS e.V. (OFFIS)
Fotonation SRL (FotoNation by Tobii)
Silicon Austria Labs GmbH (SAL)
Politecnico di Torino (POLITO)
Eidgenössische Technische Hochschule Zürich (ETHZ)
Technical University of Iasi (TUI)
BEIA Consult International SRL (BEIA)
ACP Advanced Circuit Pursuit AG (ACP)
Codasip SRO (CODA)

Contacts:
Marcus Borrmann, NXP-AT, marcus.borrmann@nxp.com
Erik Kraft, NXP-AT, erik.kraft@nxp.com
Marcel Medwed, NXP-AT, marcel.medwed@nxp.com

Document History

Version	Author	Date	Notes
0.1	Marcus Borrmann	09.01.24	Initial template draft
0.2	Marcus Borrmann	25.01.24	Incorporating partner feedback
0.3	All Partners	09.04.24	First version including partner contributions
0.4	Erik Kraft	24.04.24	Restructuring
0.5	Honorius Galmeanu, Frank K. Gürkaynak	03.05.24	Review
0.6	All Partners	14.05.24	Review comments solved
0.7	Erik Kraft	17.05.24	Final cleanup

Table of Contents

1 Executive Summary

The ISOLDE project aims to create high-performance processing systems and platforms targeting different use cases (space, automotive, smart home, cellular IoT) based on the free, open-source RISC-V instruction set architecture. This document defines the initial architecture of the required hardware modules and extensions (called extensions in the following) developed within the work package WP3 “Accelerators and Extensions” of the ISOLDE project to reach this goal. It encompasses contributions from all tasks (T3.1 to T3.6) and partners within WP3.

The extensions described in this report are grouped into different domains matching the scope of the different tasks within WP3:

1. Extensions enhancing the safety and security of RISC-V systems (T3.1)
2. Accelerator infrastructure, memories, arithmetic units, interfaces, and virtualization (T3.2)
3. Extensions that allow monitoring of the foundational core and accelerators (T3.3)
4. Accelerators speeding up vector, tensor, and other AI-related operations (T3.4)
5. Accelerators speeding up cryptographic primitives (T3.5)
6. Accelerators speeding up signal processing, neuromorphic operations, and application-specific instruction set processors (T3.6)

For each extension, this document contains general information (type, dependencies, and license) and an initial architecture description giving a first insight into its purpose and internals. These initial architecture descriptions answer core questions about each extension:

- What is the purpose of the extension?
- Where in the system is it integrated?
- How does the extension work?
- How is the extension connected with the rest of the system? How can they interact?
- How is the extension verified?

WP5 “Use Cases and Demonstrators” will combine the foundational cores developed by WP2 “Open-source Foundation Cores” and selected features from WP3, building diverse demonstrators (space, automotive, smart home, cellular IoT) that highlight benefits and opportunities enabled by individual extensions. Further, WP4 “System Software, Development Tools and Automation” will provide the required software support (e.g., toolchains, operating system support, drivers). Hence, the contributions of this deliverable are crucial for further collaboration with these work packages. In the context of WP3, this deliverable is the basis for the follow-up deliverables covering the prototype and final implementations of the extensions (D3.2, D3.3 in M24 and D3.4, D3.5 in M33). The components described in this deliverable are aiming at different maturity levels and aiming for different certifiability. Further, this document represents the first iteration of the architecture definitions and hence not all contributions have the same level of maturity. A short survey of the ISOLDE partners for certification intentions (including WP3 components) will be later provided by SYSGO as part of WP1 work.

2 Introduction

2.1 General Information

Work Package 3 (WP3) focuses on developing hardware modules and extensions enhancing RISC-V systems based on the foundational cores provided by WP2 to create and demonstrate high-performance computing systems within WP5.

The purpose of Deliverable D3.1, titled "Initial Architecture Description", is to document the preliminary design of these hardware modules and extensions. This document is intended for public release and includes the hardware extensions' initial architecture and design specifications. These initial architecture and design specifications encapsulate the extensions' technical research and developmental progress.

It is crucial to note that this document represents the first iteration of the Deliverable, marking the commencement of a bottom-up approach to address all ongoing activities comprehensively. Throughout the project's duration, the report will undergo multiple revisions to encapsulate the breadth of work completed to date.

WP3 organizes the project into distinct tasks covering different domains, with various partners contributing to each. The deliverable is structured first to introduce WP3 and its place in the overall ISOLDE project in Section 3. The subsections of Section 3 outline the tasks ranging from Task 3.1 to 3.6, offering an initial summary of each task followed by a table that outlines the extensions and the respective partners involved. The remainder of these subsections contains in-depth technical information about the specific extensions.

2.2 Purpose and Scope

This document serves as an essential guide to the ISOLDE project's activities, detailed through its technical descriptions of the hardware extensions. These descriptions not only review the past endeavours but also set the stage for the upcoming tasks. The primary goal of the initial architecture descriptions within this document is to offer preliminary insights into the project's features. This includes detailing the purpose, dependencies, and architectural nuances of each feature—covering aspects such as their system placement, block diagrams, instruction set architecture (ISA), interfaces, sub-modules, and strategies for clocking, resetting, power management, verification and debugging. Note that these aspects are not explicitly covered for every hardware extension in this document either because the aspect is not relevant for the extension, the information is not noteworthy (e.g., standard practice) or the development state is not yet mature enough. Such detailed information is crucial for seamless integration and collaboration with related work packages: WP2, which focuses on the development of foundational cores; WP5, which merges these cores with selected features; and WP4, which develops the requisite software support. Within the framework of WP3, this deliverable lays the groundwork for subsequent reports that will document the prototyping and final implementations of these features, specifically Deliverables D3.2 and D3.3 due in Month 24, and D3.4 and D3.5 scheduled for Month 33.

3 Accelerators and Extensions (WP3)

WP3 focuses on two directions: i) safety, security and monitoring infrastructure and ii) acceleration infrastructure, domain specific accelerators and ASIPs for applications such as cryptography, machine learning and signal processing.

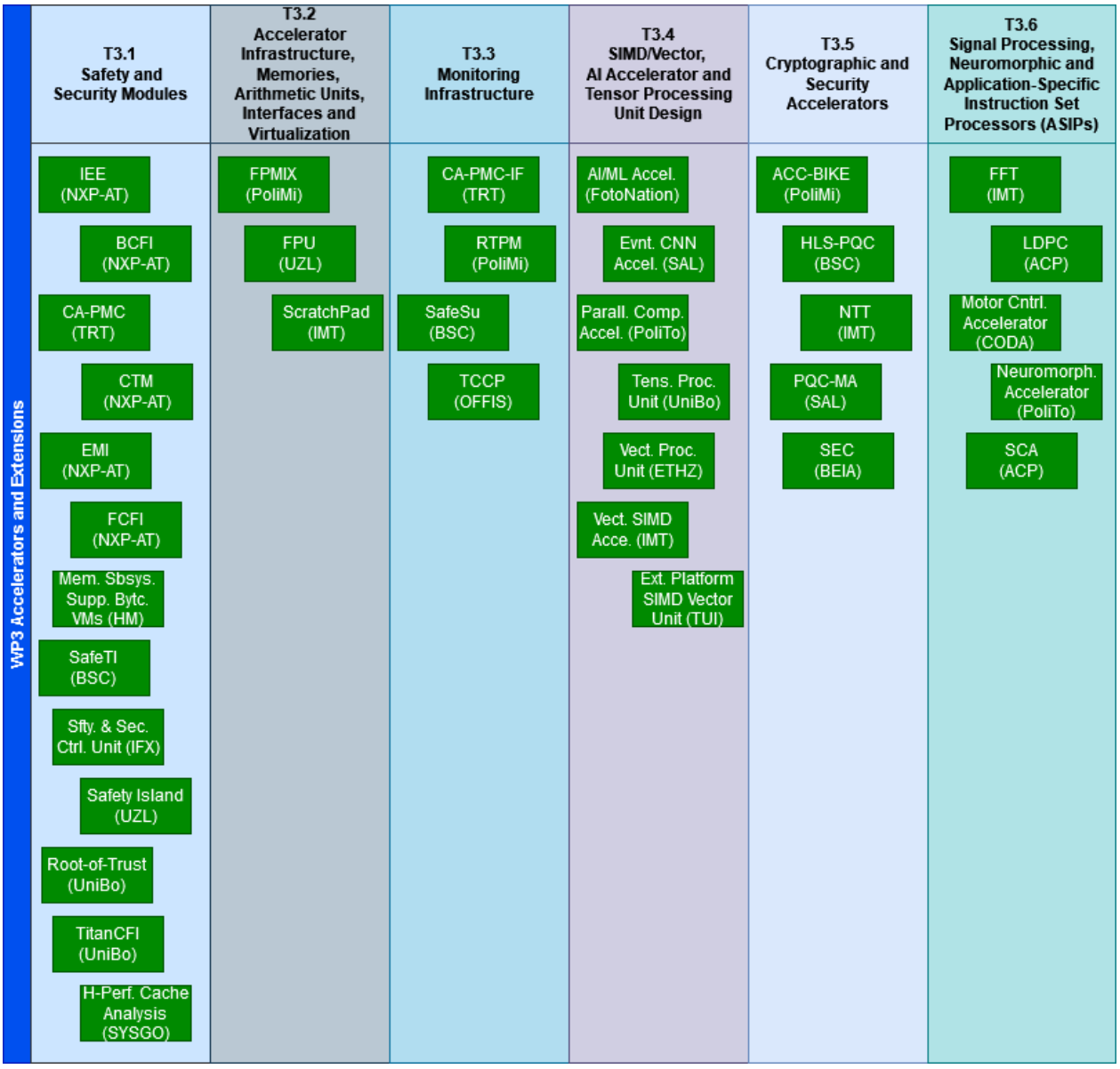


Figure 3-1: Overview of the IPs developed in WP3.

Work Package 3 “Accelerators and Extensions” receives the requirements and specifications from Work Package 1 and delivers accelerators and extensions to Work Package 5 “Use Cases and Demonstrators”. WP3 cooperates with Work Package 2 “Open-source Foundation Cores” to integrate the accelerators and Safety & Security Extensions with the general-purpose cores using interfaces and drivers. WP3 designs are used in WP4 to develop software tools. WP6 provides WP3 feedback regarding potential software

licensing issues. Regarding exploitation of the results, WP3 peripherals, modules and accelerators are made available via the developing partners or via the WP6 open-source repository channels. WP3 contributes to Milestone MS2 – Foundations. Figure 3-2 shows the interaction between WP3 and the other work packages.

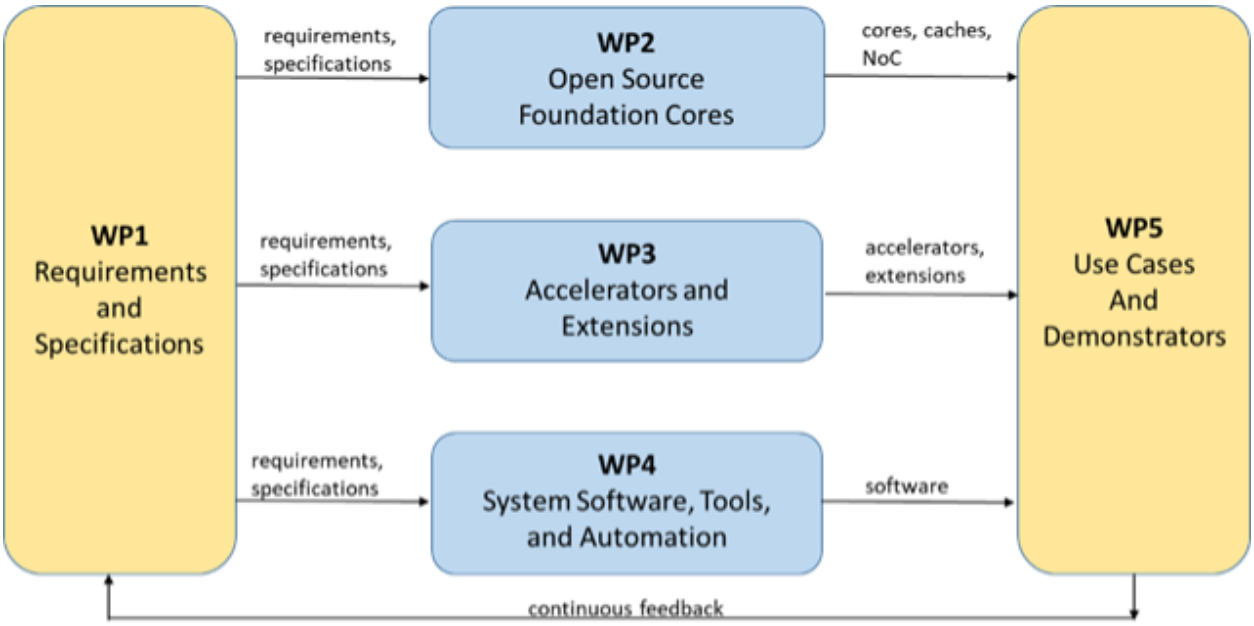


Figure 3-2: PERTT chart showing Sequence and Interactions in the ISOLDE Project

WP3 is organized in six tasks, and all tasks run in parallel from M3 to M33: T3.1 “Safety & Security Modules”; T3.2. “Accelerator infrastructure, memories, arithmetic units, interfaces and virtualization”; T3.3 “Monitoring infrastructure”; T3.4. “SIMD/Vector, AI accelerator and tensor processor unit design”; T3.5. “Cryptographic and security accelerators”; T3.6. “Signal processing, neuromorphic and application-specific instruction set processors (ASIPs)”. A top-view description of the work carried on in this package is synthesized by Figure 3-1. WP3 contributes to the following Specific Objectives (SOs): SO2, SO4, SO5, SO6 and SO7.The overall WP3 objectives as listed in the Description of the Action are:

- Encourage the development of open-source accelerators and extensions while ensuring their compatibility with closed-source IP, helping to widen the RISC-V ecosystem (SO2, SO5, SO6, SO7);
- Design hardware modules and extensions supporting and enhancing RISC-V safety and security (SO4);
- Design hardware accelerators targeting specific applications domains and exploiting available parallelism (SO2, SO5);
- Incorporate custom fixed- and floating-point units to allow trade-offs between precision and throughput (SO2, SO5);
- Design scratchpad memories for hardware accelerators (SO2, SO5);
- Develop a monitoring infrastructure to expose the relevant figures of merit for the hardware designs developed in this WP (SO2, SO5);
- Each hardware block developed in this WP will be verified at the RTL level, including unit testing and a testbench which verifies the required functionality (SO6).

3.1 Safety and Security Modules

Task 3.1, M3-M33, Task Leader: BSC

Task 3.1 focuses on the development of technologies supporting safety and security. The developed security modules include components for memory encryption, control flow integrity (CFI), memory isolation, hardware support for bytecode virtual machine interpreters, and hardware root of trust. On the safety side, the T3.1 modules include a context-aware performance monitor counter, a traffic injector for platform validation, a safety control unit collecting and processing detected on-chip errors, and an interface between the safety island and the processing system.

IP	Lead Beneficiary	Type	Domain	Dependencies	Licensing
IEE	NXP-AT	RISC-V Core extension; Core	Security	None	Proprietary closed source
BCFI	NXP-AT	RISC-V Core extension	Security	RV32I processor, IEE	Proprietary closed source
CA-PMC	TRT	Core	Safety	Integration target, CA-PMC-IF	TBD
CTM	NXP-AT	RISC-V Core extension	Security	RV32I processor, IEE	Proprietary closed source
EMI	NXP-AT	RISC-V Core extension	Security	RV32I processor, IEE	Proprietary closed source
FCFI	NXP-AT	RISC-V Core extension	Security	RV32I processor	Proprietary closed source
Memory Subsystem Support for Bytecode VMs	HM	RISC-V Core extension	Security	None	Permissive open source (SHL-2.1, Apache-2.0)
SafeTI	BSC	Core	Safety	None	Permissive open source (MIT)
Safety and Security Control Unit	IFX	Core	Safety, Security	None	Open source
Safety Island	UZL	Core	Safety	CVA6	Open source
RoT	UNIBO	Core	Security	OpenTitan	Permissive open source (Apache / SHL)
TitanCFI	UNIBO	CVA6 extension; OpenTitan extension	Security	CVA6, OpenTitan	Permissive open source (Apache-2.0)
High-Performance Cache Analysis	SYSGO	Analysis	Security	CVA6, CEA cache (TRISTAN)	Not applicable

Table 3.1-1: Overview of contributions in Task 3.1

3.1.1 Inline Encryption Engine (IEE) – NXP-AT

Part of Task 3.1 Safety & Security Modules.

3.1.1.1 General Information

Different workloads and secrets are usually isolated using the processor's privilege modes and logical isolation techniques (e.g., memory protection and management units). The RISC-V ISA defines three privilege levels: machine (M), supervisor (S), and user (U) mode. At any point, a RISC-V core is running in one of these privilege levels. The different privilege levels enable the isolation of different units in the software stack. For example, software running in M mode (e.g., the firmware) has unrestricted access to all resources and configuration registers. On the other hand, software running in S (e.g., the operating system) or U (e.g., user applications) mode is more restricted. In this setting, trusted software (e.g., the operating system) running in a higher privilege mode with the capability to configure the logical isolation sets up a restricted memory view for the workloads running in the lower privilege modes. However, this isolation breaks as soon as physical attacks are considered. Examples of such attacks include:

- Attacks physically probing the external memory bus between the processor and memory chip.
- Attacks injecting faults in the logical isolation primitives or during their configuration by the trusted software [Nashimoto2021].
- Attacks using fault injection to modify the data in the memory [Roscian2013]. Note that software-based attacks can also trigger fault injections in memory, as *Rowhammer* [Kim2014] showed.

Major processor vendors started integrating memory encryption engines into their architectures to mitigate these threads transparently [Kaplan2021, Intel2021]. The memory encryption engine encrypts all data before it reaches the external bus. The resulting ciphertext depends on the physical address (used as tweak) in addition to the secret key to counter attacks exchanging the encrypted data words. The hardware samples the memory encryption key from a random number generator at each reset and stores it in a non-accessible register.

In systems with an enabled memory encryption engine, all data on the external bus and in memory is encrypted. Therefore, the attacker can no longer leak data or easily inject controlled modifications. If a pure encryption scheme is used as a primitive for memory encryption, then the confidentiality of the data on the external bus and in memory is protected. However, the integrity of the data cannot be ensured, meaning the attacker can perform modifications without detection. Still, the attacker cannot change the plaintext in a controlled way, assuming the relationship between plaintext and ciphertext is unknown, i.e., the attacker has no way to build a dictionary of plaintext-ciphertext pairs for the address. Alternatively, authenticated encryption schemes can be used, which also provide data integrity but have a higher memory overhead.

Memory encryption engines prevent or raise the bar for many physical attacks, but they also impose considerable area and latency overhead while not increasing resilience against logical attacks.

3.1.1.2 Purpose and Scope

The Inline Encryption Engine (IEE) module adds a tweakable memory encryption engine based on a low-latency block cipher to the base RV32 core. The difference to the schemes mentioned in the previous section is that we use a tweak input to make the resulting ciphertext depend on additional metadata. This metadata is provided by other modules contributed by NXP-AT, which add defense mechanisms against logical attacks. For more information about these modules, please refer to the architectural description of the Backward-Edge Control Flow Integrity (BCFI; see Section 3.1.2), Cryptographically Tagged Memory (CTM; see Section 3.1.4), and Enclave Memory Isolation (EMI; see Section 3.1.5) module provided by NXP-AT. Systems that already include a memory encryption engine especially benefit from this approach, as adding the mentioned defenses against logical attacks comes at little cost.

3.1.1.3 Place in the System

The IEE module targets RV32 cores without a Memory Management Unit (MMU). Figure 3.1.1.3-1 shows that the IEE module is placed before the memory controller (MC), acting as a wrapper for the memory requests issued by the cache subsystem. Hence, the IEE module has two bus interfaces (e.g., Arm Advanced Microcontroller Bus Architecture Advanced High-performance Bus – AMBA AHB), one connecting it to the cache subsystem and the other to the memory controller. Further, the IEE module requires the IEE-RV ISA extension and changes to the cache subsystem which will all be described in the following. Note that the IEE module can also be integrated into processors without a cache, but then the latency of the memory encryption and decryption has more impact on the core's performance.

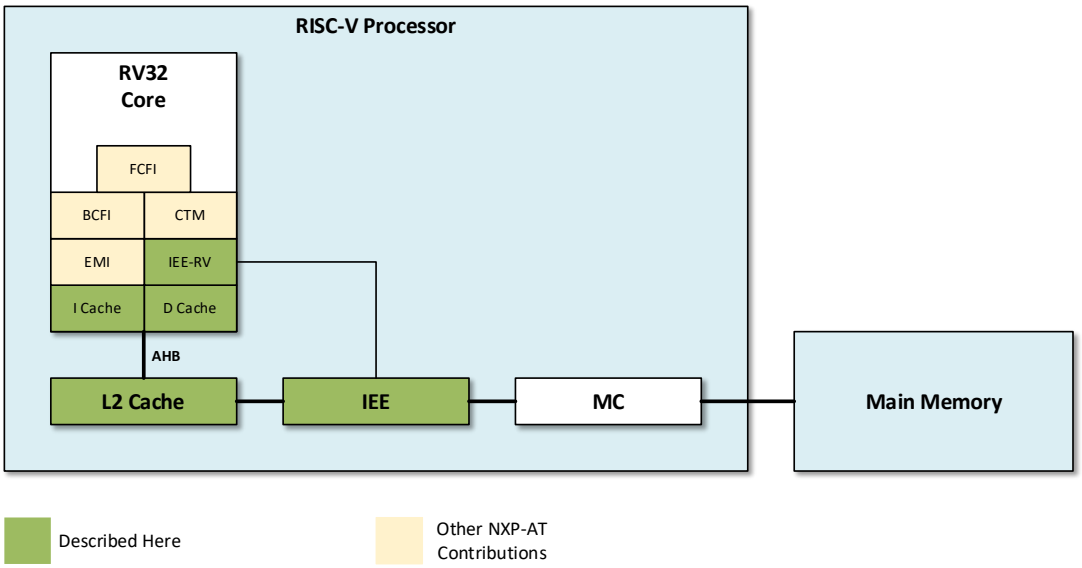


Figure 3.1.1.3-1: Overview of the IEE module in the system

3.1.1.4 Block Diagram

Figure 3.1.1.4-1 shows the internals of the IEE module. The IEE module is placed before the memory controller to ensure that all data on the bus to and in the memory is encrypted. Therefore, it is connected to the cache subsystem which issues memory requests (`ahb_cache`) and the memory controller (`ahb_mem`). The cache subsystem also provides a tweak (`tweaki`) together with the memory request. The tweak together with the secret key (`ieekeyi`) defines the permutation used for encryption and decryption. Therefore, an attacker cannot determine the plaintext associated with a certain ciphertext block without knowing the correct key and tweak. The contents of the tweak are passed along with the memory requests issued by the processor (tagged transactions) and are defined by NXP-AT's countermeasures built on top of the IEE module (BCFI, EMI, CTM). Attacks within the scope of these modules cannot trigger memory requests with a tweak matching the genuine tweak of any protected data in memory. Hence, an attacker cannot leak or perform controlled modifications of the protected data. For more information, see the architecture description of the modules mentioned above. Finally, the IEE module is connected to the Control and Status Registers (CSRs) of the main RV32 core containing the encryption key (`ieekey`) and the top address of the encrypted memory region (`mieeencend`). These ISA modifications are part of the IEE-RV ISA extension and are described in more detail in the Section 3.1.1.5. If the requested memory location is greater or equal to the value in the `mieeencend` CSR, then the memory encryption is bypassed.

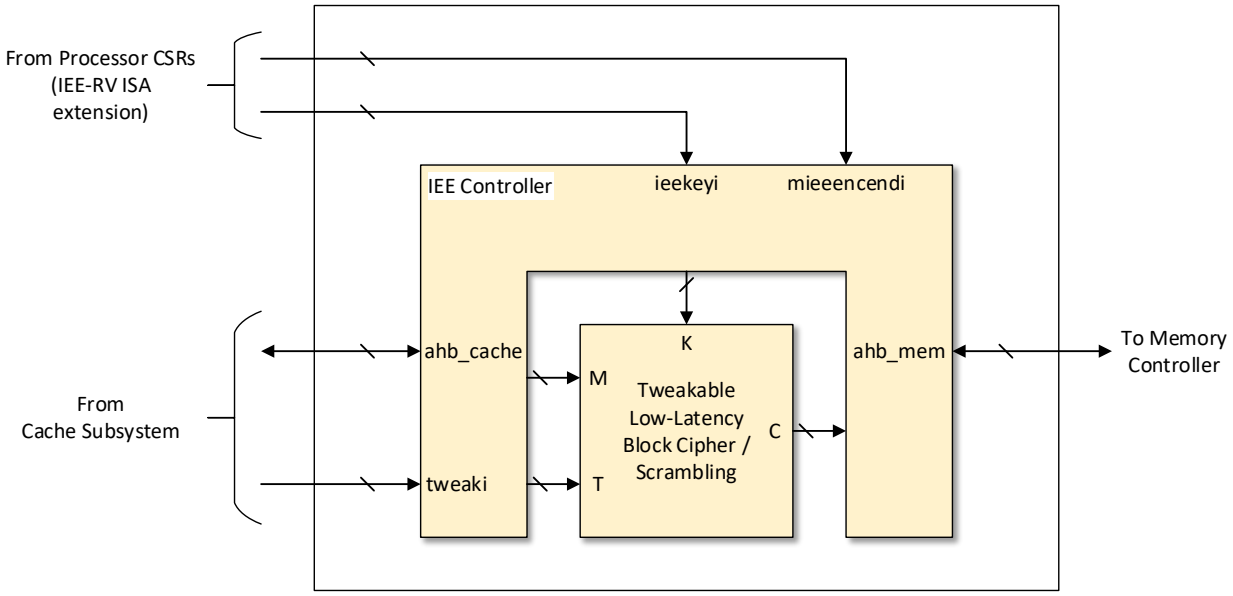


Figure 3.1.1.4-1: Internal architecture of the IEE module

The selection of the actual block cipher used in the IEE module depends on the specific use case, performance requirements and threat model. More information on the requirements of a suitable block cipher in our threat model, which includes logical attacks, is presented in Section 3.1.1.7.

If the processor design includes caches, then they must be adapted so that the tweak inputs generated by NXP AT's countermeasures are able to propagate from the processor through the cache subsystem to the memory encryption engine, where they are needed to form the final encryption tweak. Figure 3.1.1.4-2 shows the required changes using an L2 cache as example.

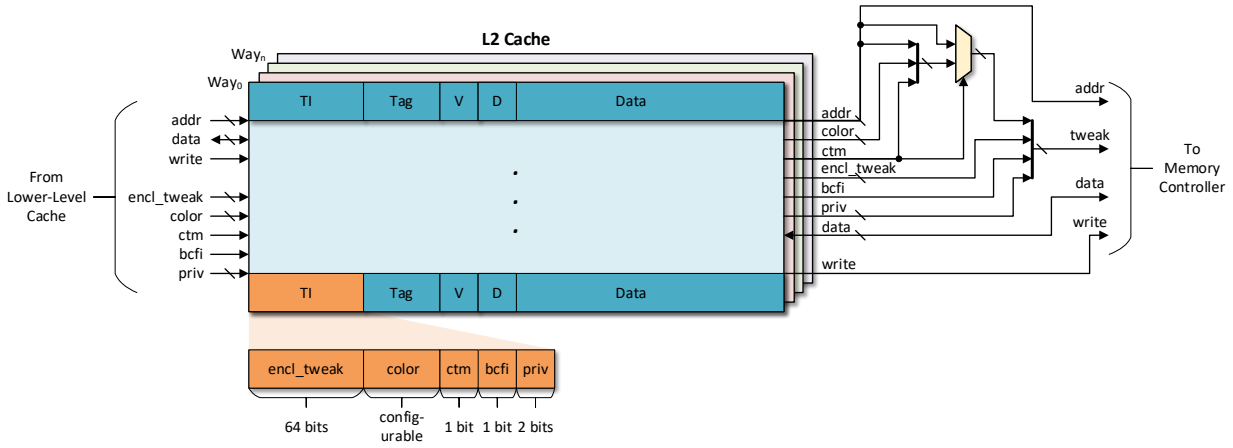


Figure 3.1.1.4-2: IEE - Modifications to the L2 cache

As can be seen in Figure 3.1.1.4-2, an additional tweak input (`TI`) field was added to the cache line which contains outputs of NXP-AT's countermeasures that should influence the encryption:

- `encl_tweak`: Provided by NXP-AT's EMI module enabling enclave isolation and confidential compute.

- `color, ctm`: Provided by NXP-AT's CTM module defending against exploitation of memory safety vulnerabilities.
- `bcfi`: Provided by NXP-AT's BCFI module enabling a cryptographic-isolated shadow stack protecting return addresses spilled on the stack.
- `priv`: Included to enforce cryptographic isolation of different privilege modes.

When issuing a memory request to the memory controller these fields are combined to form the final encryption tweak as follows ("||" denotes a bitwise concatenation of bit vectors where the most significant bit vectors are the leftmost ones):

$$tweak = encl\_tweak \,||\, tagged\_addr \,||\, bcfi \,||\, priv$$

tagged_addr denotes a CTM-protected address including the color and CTM selection bit (see NXP-AT's CTM module architecture description in Section 3.1.4 for more information).

The same changes apply for L1 data caches. However, for L1 instruction caches, the `TI` field will not contain the `color, ctm` and `bcfi` subfields as the related countermeasures are not relevant for instruction memory, instead these fields are hardcoded to 0. The cache modifications also include changes to the cache hit logic and replacement policy to guarantee the security claims and ensure correct function:

- A cache hit only occurs if the `Tag` and `TI` field of the cache line match the ones of the memory request.
- A cache miss occurs if the `Tag` field does not match, or if the `Tag` field matches and the `TI` field does not. In the second case, the replacement logic must always select the cache line with the mismatching `TI` field as replacement candidate. This behavior avoids cache aliasing where multiple cache lines would be associated with the same physical address. Otherwise, cache aliasing could lead to inconsistency and security problems.

Note that the block size of the cipher shall match the minimum access size performed by the cache subsystem to avoid unnecessary read-modify-write scenarios. Further, the encrypted region must be cacheable if caches are included in the design. Access to non-cacheable regions bypasses the memory encryption as such access would come with a higher latency penalty especially for sub-word stores.

The described redesign of the cache subsystem inflicts a significant overhead in the area consumed by the caches because of the large `encl_tweak` subfield in the `TI` field. For example, if the cache line size is 32 byte and 12-bit colors are used the overhead would be 31.25%. However, in practice, the overhead can be significantly reduced by storing the enclave tweaks in a separate lookup table and linking them to the appropriate cache lines in the `TI` field instead of including the full enclave tweak. If an enclave tweak in the lookup table is replaced in such an implementation, then all cache lines linking to this enclave tweak must also be evicted.

In the described design, other peripherals which function as bus managers apart from the main RV32 core, like a Direct Memory Access (DMA) controller, can only exchange data with the main core by operating in the unencrypted memory area (greater or equal to `mieeencend`). These bus managers shall not be able to access the encrypted region, and they shall use a special tweak isolated from the tweaks of the main RV32 core (for example, by using the unused encoding `10` for the `priv` field), enforcing cryptographic isolation in addition. Enabling other bus managers to access an encrypted region requires the trusted software to be able to bind them to a security domain (e.g., only a specific enclave can access the peripheral) and provide a proper encryption tweak to the peripheral so that it can access a shared encrypted region.

3.1.1.5 ISA

The IEE module requires an ISA extension of the RV32 core (IEE-RV) to allow the main processor to provide the encryption key and address range where the encryption is active. The next sections describe modified and new design parameters, CSRs, instructions, and exceptions introduced by the IEE-RV extension. In systems with multiple bus masters the described CSRs shall rather be implemented as

memory-mapped registers of the IEE module. Otherwise, the value of the registers might change during the operation of the IEE module leading to inconsistency problems. If the CSRs are moved to memory-mapped registers, then the bus must include appropriate privilege information so that only privileged and trusted software of the main core can access these registers.

Design Parameters

Parameter	Value	Function
IEE_ENCRYPTION_BASE	TBD[1]	Base address of the encrypted memory region
IEE_ENABLE	1	Determines if the Inline Encryption Engine module is included in the design. If it is not included, then also none of the modules based on it can be included (NXP-AT's BCFI, CTM, EMI).

Table 3.1.1.5-1: IEE module design parameters

CSRs

CSR	Index	Privilege	Bits	Function
ieekey0	TBD[2]	M-RW	31-0	Least Significant Word (LSW) of the key for memory encryption
ieekey1	TBD[2]	M-RW	31-0	32-bits of the key for memory encryption
ieekey2	TBD[2]	M-RW	31-0	32-bits of the key for memory encryption
ieekey3	TBD[2]	M-RW	31-0	Most Significant Word (MSW) of the key for memory encryption
mieeencend	TBD[2]	M-RW	31-0	One byte beyond the last address of the encrypted memory region
uieeencend	TBD[2]	U-R	31-0	User-mode read-only alias of mieeencend (if, e.g., allocator needs this info)

[1] This parameter depends on the memory map of the system to which the IEE module should be added.

[2] These indices depend on the free CSR addresses in the processor to which the IEE-RV extension should be added.

Table 3.1.1.5-2: CSRs added by the IEE-RV module.

Note that writes to these CSRs should only be retired after flushing the pipeline and after all pending memory operations are completed. Otherwise, memory access by subsequent instructions may unintentionally still use the previous configuration or pending memory access might already use the new configuration. Both scenarios lead to a bypass of the cryptographic isolation in the worst case.

3.1.1.6 Interfaces

The IEE module uses the same interface as used to connect the L2 cache and memory controller (e.g., AHB). Additionally, the L2 cache provides the tweak inputs to the IEE module.

3.1.1.7 Sub-Modules

The IEE module requires a suitable implementation of a tweakable low-latency block cipher. The requirements for suitable schemes are described in the following section.

Tweakable Low Latency Block Cipher

A tweakable low-latency block cipher suitable for our threat model including logical attacks shall fulfill the following requirements:

- The block size shall match the minimal access size performed by the cache subsystem (if the design includes caches) but shall be at least 32-bit.
- The algorithm shall provide 128-bit security against key recovery.
- The tweak shall be large enough to accommodate the previously described tweak inputs (depending on which countermeasures are included) and provide security against online tweak recovery.
- The block cipher shall be non-malleable.
- The acceptable latency of the encryption and decryption operations depends on the performance requirements. If the design includes caches (especially a last-level cache with write-back policy), then the IEE has less impact on performance as not every memory access has to pass through the memory encryption engine. Hence, in this case the latency requirements can be more relaxed.

Depending on the performance requirements and additional security requirements (e.g., integrity protection) solutions built on top of different block ciphers, for example, PRINCEv2 [Božilov2020], QUARMAv2 [Avanzi2023] or ASCON [Dobraunig2016] can be built.

3.1.1.8 Clocking Strategy

The IEE wrapper and included block cipher clock frequency depend on the latency requirements for memory encryption and decryption.

3.1.1.9 Reset Strategy

The IEE module uses the same reset line as the memory controller.

3.1.1.10 Power Management Strategy

If the memory controller and its connected memory are power gated or going to a power saving mode, then likewise the IEE module can do so.

3.1.1.11 Debugging Strategy

The RISC-V processor's debug module shall be limited to accessing the memory from a hart's point of view using the program buffer and not via the system bus access if the IEE module is included in the design. The system bus access mode cannot provide the appropriate tweaks as they depend on the state of the RV32 core. Hence, loads and stores with addresses in the encrypted region would lead to wrong results. Even if memory is accessed via the program buffer, the RV32 core must still be in an appropriate state. For example, if a user wants to access a memory location related to Enclave 2, then debug mode must have been entered from Enclave 2, or the debugger must have set the appropriate enclave modifiers.

Suppose the RV32 core is in debug mode. In that case, the memory encryption engine shall be bypassed for instruction fetches to allow normal execution of the debug ROM and instructions in the debug program buffer. Further, the debugger must set the memory access privilege to the privilege mode the hart was executing in before entering debug mode. Otherwise, the memory encryption tweaks will not match the current state of the art. Hence, the debug module implementation must not tie `dcsr.mprven` to 0 [Donahue2024]. Then, the debug translator can set the proper memory access privilege by performing the following sequence after entering debug mode:

1. Set `dcsr.mprven` to 1 (if not fixed).
2. If `dcsr.prv` is M and `mstatus.mprv` is set, skip the following steps.
3. Back up the state of the `mstatus.mprv` and `mstatus.mpp` fields.
4. Set `mstatus.mprv` to 1 and `mstatus.mpp` to `dcsr.prv`.

When returning from debug mode, the debug translator shall restore the `mstatus.mprv` and `mstatus.mpp` field values.

Additionally, if the debug translator wants to modify text sections (e.g., for inserting software breakpoints), it must follow these steps:

1. Back up the value of the effective enclave store modifier selected using the `dcsr.prv` field (see also the architecture description of NXP-AT's EMI module in Section 3.1.5).
2. Set the effective enclave store modifier to 0.
3. Perform the necessary changes in the instruction memory region.
4. Restore the effective enclave store modifier.

As mentioned above, the core must be in an appropriate state to correctly access memory, which also applies to instruction memory. Alternatively, the user can add hardware breakpoints implemented using the trigger module. Those are directly compatible with NXP-AT's IEE module without changes.

3.1.2 Backward-Edge Control Flow Integrity (BCFI) – NXP-AT

Part of Task 3.1 Safety & Security Modules.

3.1.2.1 General Information

According to Google Project Zero, memory corruption vulnerabilities are the most used path to gain unintended remote control over digital devices [GoogleProjectZero2024]. In 2023, 75% of zero-day exploits in the wild were based on memory corruption vulnerabilities. Programming languages like C and C++ that offer neither memory nor type safety are especially affected. While memory-safe programming languages (like Rust) gain momentum, C is still one of the most popular programming languages [Cass2022], especially for embedded system development. Making matters worse, constrained embedded environments include only subsets of the defense mechanisms employed in larger systems (e.g., no address space layout randomization or low entropy), leading to easier exploitation. Hence, during the transition period to memory-safe programming languages or for legacy code, additional security layers are needed to mitigate these attack paths, especially for constrained embedded devices.

3.1.2.2 Purpose and Scope

The exploitation of memory safety vulnerabilities allows an attacker to unintendedly corrupt, or leak program data. On the one hand, introducing malicious changes may enable an attacker to modify the program behavior and take over control. On the other hand, unintended leakage of data may lead to the attacker learning sensitive information like the value of cryptographic keys. Memory safety vulnerabilities can be separated into spatial bugs (out-of-bound reads and writes) and temporal bugs (reusing a dangling pointer after the associated memory block was given back to the allocator).

One typical target of exploits arising from memory safety issues are return addresses spilled on the stack as modifying them allows the attacker to jump to any wanted address. Such exploits, which aim to modify the backward-edge control flow of programs, are among the most common and known ones. The typical attack path is to exploit wrong or missing bounds checks of a user-controllable input that is written to a buffer on the stack leading to overwritten stack contents including return addresses. Early attacks overwrote the return address and injected the attacker code on the stack, jumping to the injected code using the modified return address.

Subsequent countermeasures, like *W^X* (no execution of writeable memory regions), resulted in more advanced attack techniques like *return-to-libc*, *Return-Oriented Programming* (ROP*), Jump-Oriented Programming* (JOP), or *Call-Oriented Programming* (COP). These attacks do not inject shellcode crafted by an attacker, but instead chain together available gadgets (snippets of code useful for the attacker) in the victim's instruction memory to achieve the attacker's goal.

Modern operating systems include more advanced countermeasures like *stack canaries* and *Address Space Layout Randomization* (ASLR) to increase the resilience regarding these advanced attack types. Stack canaries aim to protect return addresses from buffer overflows by inserting a unique value between the function data and the saved return address on the stack. The canary will be overwritten if the attacker tries to overwrite the return value by exploiting a buffer overflow. As the canary value is compared to the expected one in the function epilogue, the attack is detected before returning from the function. ASLR randomizes the start location of a program's address space portions (code, stack, heap, libraries). This randomization hides addresses of useful gadgets from an attacker leading to probabilistic mitigation.

Still, all these countermeasures have weaknesses. Stack canaries do not help against an attacker with an arbitrary write primitive, as the canary can just be skipped. ASLR may not be available on 32-bit embedded systems or has low entropy there. Further, both are vulnerable to information disclosure attacks.

Hence, shadow stacks were invented as a stronger alternative. A shadow stack duplicates all return addresses on the regular stack in a separate isolated memory region. The isolation guarantees that only special instructions can access the shadow stack while regular memory operations cannot access it.

Therefore, an attack can only overwrite return addresses on the regular stack, but not on the shadow stack. Before executing the return in the function epilogue, the return address from the regular stack is compared with the return address on the shadow stack. If they do not match, then an attack is detected, and the execution terminates with an appropriate exception. RISC-V International specified a shadow stack design for RISC-V in the *Zicfiss* extension [RV-SS-LP-TG2024].

The shadow stack specified in Zicfiss requires that the processor includes a MMU to realize the isolation and hence may not be suitable for small 32-bit platforms (microcontrollers). Instead, our BCFI module isolates the shadow stack cryptographically using a tweakable memory encryption engine like NXP-AT's IEE module (Section 3.1.1). Such an approach is beneficial for systems which already include a memory encryption engine to fulfil their security requirements. Then, this memory encryption can be reused to implement a shadow stack at little additional cost. Shadow stack operations use encryption tweaks different from other memory operations to implement the before mentioned isolation cryptographically. However, while encrypting the shadow stack with tweaks different to other memory regions could be already achieved by using NXP-AT's CTM module (Section 3.1.4), it is not sufficient against all attack scenarios. For example, even if the encryption additionally depends on the physical address, an attacker can still exchange return addresses located at the same stack depth (physical address) but recorded at different points in time.

Therefore, our approach is an adoption of the scheme proposed by the PACStack paper [Liljestrand2021]. At every function call, the return address is used to update a hash value stored in an isolated register (`sstca`). The intermediate hash values are spilled to the stack (plus encrypted by the BCFI-specific memory encryption) instead of the return addresses. For the hash computation we use an invertible universal hash function (`UHF` and its inverse `IUHF`). Hence, knowing the topmost hash value and the intermediate hash values, the return addresses can be reconstructed in the function epilogues. The topmost hash value represents all return addresses on the stack, i.e., if any return address would change, then this change would lead to a statically unique topmost hash value. Therefore, any successful attack would require either modifying the topmost hash value or a value on the shadow stack in a controlled way, both are not feasible (mitigated by the isolated register and the memory encryption respectively). If an attacker still attempts to modify a value on the shadow stack, then the resulting return address will be random, and the attack will be detected as it does not match the value on the regular stack. This concept is illustrated in the Figure 3.1.2.2-1.

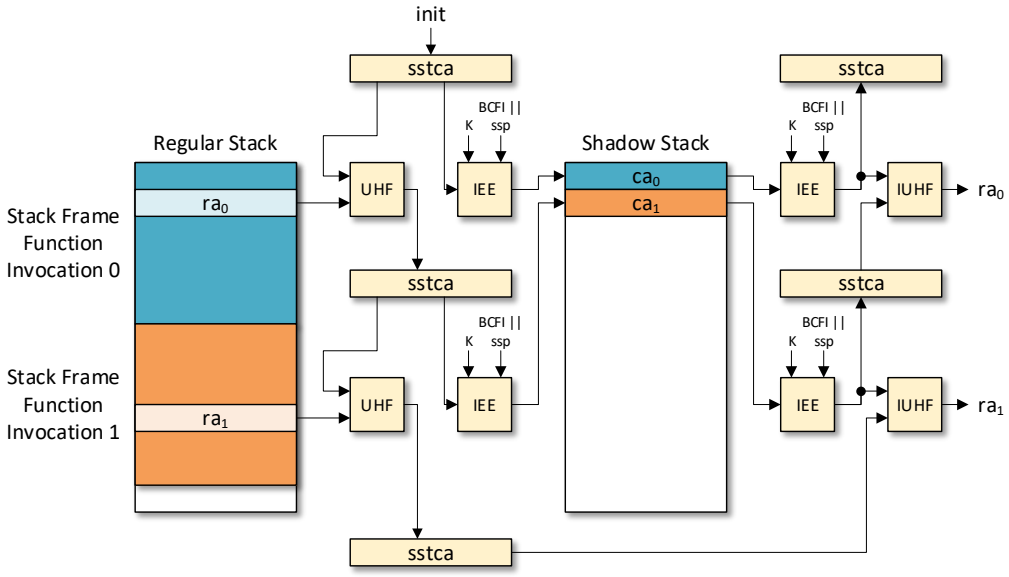


Figure 3.1.2.2-1: BCFI - Cryptographically isolated shadow stack

Note that instead of duplicating the return addresses, they can only be stored on the shadow stack resulting in no memory overhead (also called *control stack mode*). However, then attacks cannot be detected anymore, instead the reconstructed return address will be random and likely lead to a fault preventing exploitation.

3.1.2.3 Place in the System

The BCFI module is an ISA extension for RV32 cores as can be seen in Figure 3.1.2.3-1. Further, it requires changes to the cache architecture and depends on a suitable tweakable memory encryption engine. These contributions are not described here, but in the section describing NXP-AT's IEE module (Section 3.1.1).

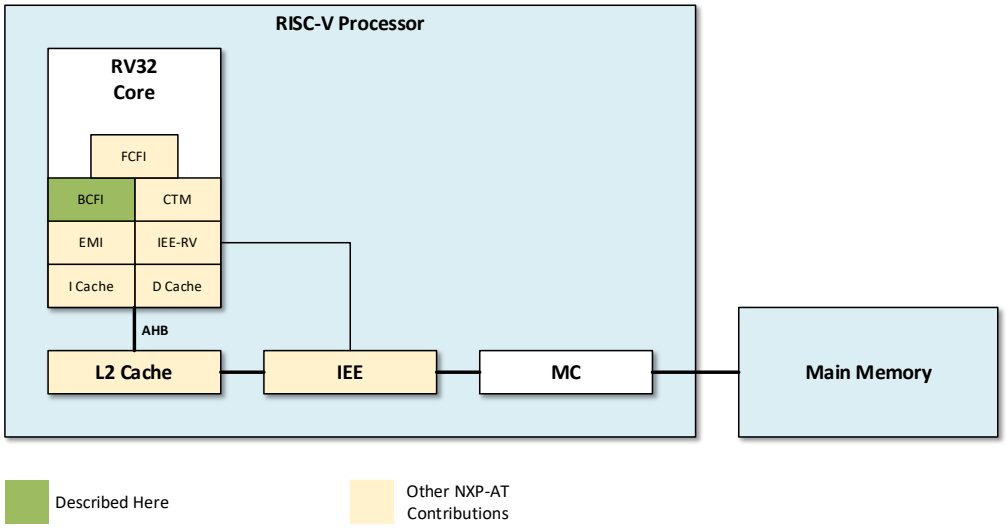


Figure 3.1.2.3-1: Overview of the BCFI module in the system

3.1.2.4 Block Diagram

Figure 3.1.2.4-1 shows the internal architecture of the BCFI module. The figure also indicates to which pipeline stages and modules inside the RV32 core the BCFI module will be connected. The description assumes that the base core has 7 pipeline stages (instruction fetch, decode, register access, execute, memory, exception, write-back).

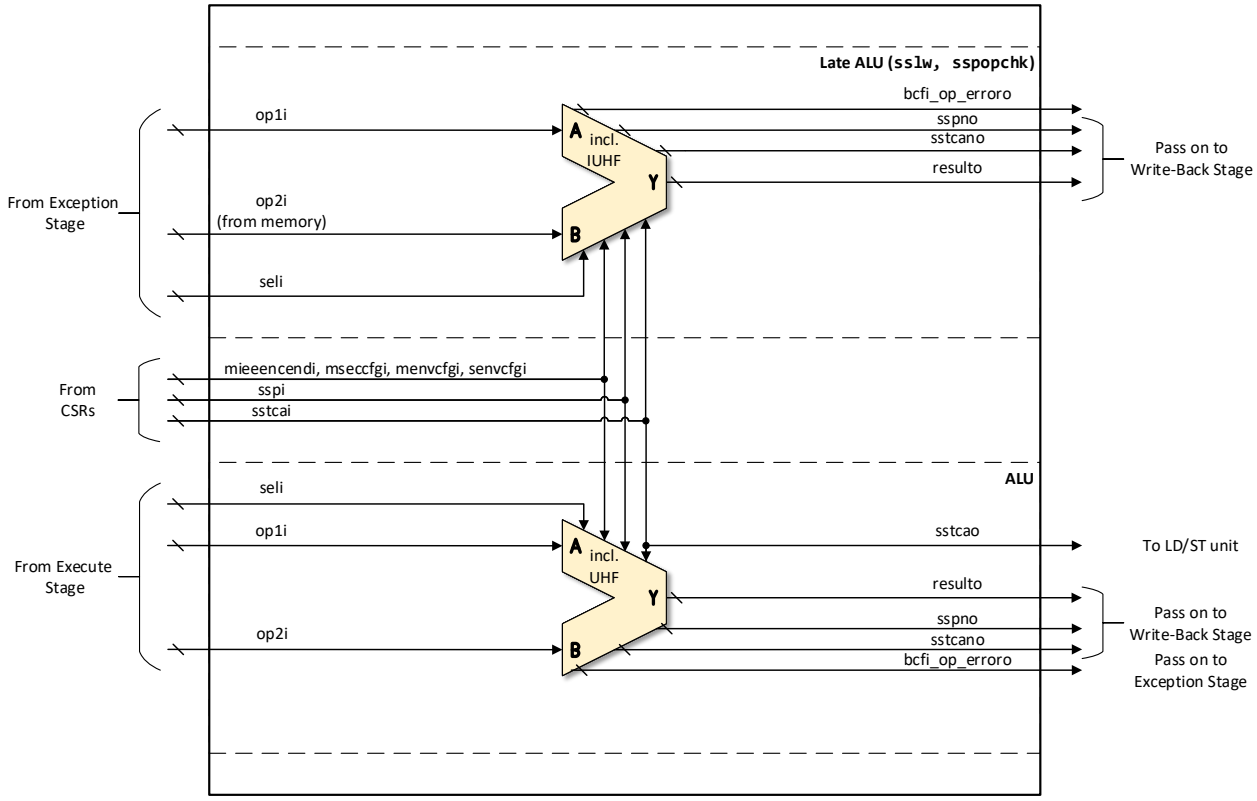


Figure 3.1.2.4-1: Internal architecture of the BCFI module

The BCFI module extends the ISA of the RV32 core with multiple instructions, listed in the tables below. The corresponding operations are added to the normal Arithmetic Logic Unit (ALU) which gets its inputs from the Execute Stage (every instruction except for `sslw` and `sspopchck` - see Table 3.1.2.5-4), or the late ALU, which gets its inputs from the Exception Stage as one of the operands must be loaded from memory (for `sslw` and `sspopchck`). Further inputs come from the CSRs related to the shadow stack operation (`ssp`, `sstca`, `mieeencend`, `mseccfg`, `menvcfg` and `senvcfg`). The regular ALU must contain a module for universal hashing (`UHF`) and the late ALU a module for computing the inverse (`IUHF`) required for the hashing and reconstruction of the return addresses. The ALUs output an updated shadow stack state (`sspno`, `sstcano`) and an output value (`resulto`) which are passed to the next stage. Additionally, an error signal (`bcfi_op_erroro`) is asserted if a security check fails. The error signal will lead to the processor jumping to an appropriate exception handler. For a detailed description of the BCFI instructions and other ISA changes refer to the next section.

3.1.2.5 ISA

The BCFI module extends the ISA of the base RV32 core. The next sections describe modified and new design parameters, CSRs, instructions, and exceptions. In the following, *BLEN* denotes the block size of the memory encryption in bits (see also the description of NXP-AT's IEE module in Section 3.1.1). Note that the referenced `IEE_ENCRYPTION_BASE` design parameter and `mieeencend` CSR are part of NXP-AT's IEE module and described there. The instruction encodings and assembly syntax match the one defined by the official Zicfiss extension [RV-SS-LP-TG2024] allowing compilers supporting this extension being reused for our cryptographically isolated shadow stack. Note that the instructions required for control stack mode (`sslw`, `ssincp`) match version 0.3.1 of the Zicfiss extension [RV-SS-LP-TG2023] which was the current version at the time of development. The ratification of control stack mode has been postponed in

favour of a simpler and easier to ratify first version of the extension, but the control stack is expected to be ratified with the next update.

Design Parameters

Parameter	Default Value	Function
BCFI_ENABLE	1	Determines if the BCFI module is included in the design.

Table 3.1.2.5-1: BCFI module design parameters

CSRs

In the following tables, the privilege column gives the minimum required privilege mode (first letter) to access the CSR and which access types are allowed (last two letters). For example, M-RW means M-mode has read and write access, but lower privilege modes have no access.

CSR	Index	Privilege	Bits	Function
ssp	0x011	U-RW	31-0	Shadow Stack Pointer
sstca[3]	TBD[4]	U-RW	*(BLEN-1)*-0	Topmost Chained Address (topmost hash value)
ssuhfx[3]	TBD[4]	M-RW	*(BLEN-1)-0*	Parameter X of UHF
ssuhfxinv[3]	TBD[4]	M-RW	*(BLEN-1)*-0	Parameter X^{-1} of IUHF

Table 3.1.2.5-2: CSRs added by the BCFI module.

The access conditions for `ssp` and `sstca` further depend on the state of the BFCI module (enabled, disabled) in the current privilege level. If BCFI is disabled, then access to these CSRs raises an `IllegalInstruction` exception.

CSR	Index	Privilege	Field	Bit(s)	Function
mseccfg	0x747	M-RW	SSUP	3	M-mode shadow stack grows in upward direction.
			SSCHK	4	If set, then `ssp < sp` is asserted at instructions accessing the regular stack or the shadow stack in M mode. Only meaningful if SSUP is also set.

[3] May require multiple CSRs depending on *BLEN* but represented as single CSR as simplification.

[4] These indices depend on the free CSR addresses in the processor to which the BCFI module should be added.

menvcfg	0x30A	M-RW	SSE	3	Shadow stack enabled in S mode
			SSUP	8	S-mode shadow stack grows in upward direction.
			SSCHK	9	If this flag and MENVCFG.SSE are set, then ssp < sp is asserted at instructions accessing the regular stack or the shadow stack in S mode. Only meaningful if SSUP is also set.
senvcfg	0x10A	S-RW	SSE	3	Shadow stack enabled in U mode
			SSUP	8	U-mode shadow stack grows in upward direction.
			SSCHK	9	If this flag and SENVCFG.SSE are set, then ssp < sp is asserted at instructions accessing the regular stack or the shadow stack in U mode. Only meaningful if SSUP is also set.

Table 3.1.2.5-3: CSRs modified by the BCFI module.

Note that the shadow stack is always enabled in M mode. SSUP allows to place the regular stack and the shadow stack in the same memory region growing towards each other. Additionally, SSCHK can be used to detect when the stacks overlap in this scenario.

The ssp < sp assertion only works correctly if the executed code complies with the RISC-V (E)ABI [RVI2024] (sp is at least 8-byte aligned, sp is mapped to register x2, sp offsets are only positive). If xSSCHK is enabled, then an implementation can optionally assert that sp (x2) is 8-byte aligned and that offsets of memory operations using sp (x2) are positive. If any of the above-described assertions fails, a SoftwareCheck exception (xCAUSE=18, xTVAL=4) shall be raised.

Instructions

All memory accesses performed by the operations of the BCFI module set the BCFI flag in the tweak propagated to the memory encryption engine to isolate them from all other memory operations. For more information, see the description of NXP-AT's IEE module (Section 3.1.1). Note that shadow stacks must be aligned to the cache line size as the IEE module stores tweaks at a cache line granularity. The following operation description makes use of the symbol "*…"* to denote a slice of memory, i.e., *memory[start … end]* denotes the byte range in memory starting with *start* until (exclusive) *end* (like slices in Python). Note that all BCFI memory accesses must be BLEN/8-aligned, otherwise a StoreAccessFault exception should be raised.

Instruction	**Operation**
sspush rs ***(rs={x1,x5})*** **c.sspush** ***(rs=x1)*** ***Push to shadow stack***	`if (xSSE = 1)` `{` `  addr = xSSUP ? ssp + (BLEN / 8) : ssp - (BLEN / 8)` `  if (is_access_outside_iee_region(addr, BLEN / 8))` `  {` `  /* xCAUSE=18, xTVAL=5 */` `  raise SoftwareCheckException(5)` `  }` `  memory[addr … addr + (BLEN / 8)] = sstca` `  t = UHF(rs, sstca)` `  /* only at retirement */`

	sstca = t ssp = addr }
sslw rd (rd={x1,x5}) ***Load from shadow stack***	if (xSSE = 1) { addr = ssp if (is_access_outside_iee_region(addr, BLEN / 8)) { */* xCAUSE=18, xTVAL=5 */* raise SoftwareCheckException(5) } t = memory[addr … addr + (BLEN / 8)] rd = IUHF(sstca, t) /* only at retirement */ sstca = t } else { */* Zimop default behavior */* rd = 0 }
ssincp ***c.ssincp*** ***Increase shadow stack pointer***	if (xSSE= 1) { ssp = xSSUP ? ssp - (BLEN / 8) : ssp + (BLEN / 8) }
sspopchk rs (rs={x1,x5}) **c.sspopchk (rs=x5)** ***Pop from shadow stack and compare with link register***	if (xSSE = 1) { addr = ssp if (is_access_outside_iee_region(addr, BLEN / 8)) { */* xCAUSE=18, xTVAL=5 */* raise SoftwareCheckException(5) } t1 = memory[addr … addr + (BLEN / 8)] t2 = IUHF(sstca, t1) if (t2 != rs) { */* xCAUSE=18, xTVAL=3 */* raise SoftwareCheckException(3) */* trap handler might restart instruction! */* } /* only at retirement */ sstca = t1 ssp = xSSUP ? ssp - (BLEN / 8) : ssp + (BLEN / 8) }
ssrdp rd ***Read ssp into a register***	if (xSSE = 1) { rd = ssp } else { */* Zimop default behavior */* rd = 0 }
ssamoswap.w rd, rs2, (rs1) ***Atomic swap from shadow stack location***	if (xSSE = 1) { addr = rs1 if (is_access_outside_iee_region(addr, BLEN / 8)) { */* xCAUSE=18, xTVAL=5 */* raise SoftwareCheckException(5)

	} /* perform atomically with sequential consistency */ rd = memory[addr … addr + (BLEN / 8)] memory[addr … addr + (BLEN / 8)] = rs2 } else { /* xCAUSE=2 */ raise IllegalInstructionException }

Table 3.1.2.5-4: Instructions added by the BCFI module.

`ssamoswap` behaves like a regular `amoswap.w`, but it sets the BCFI flag in the memory encryption tweak.

The Table 3.1.2.5-4 refers to `xSSE`, `xSSCHK`, `xSSUP` and the function `is_access_outside_iee_region()` which are defined below:

```
xSSE =
{
  if (in M mode)
     return 1
  /* below M mode */
  else if (menvcfg.SSE == 0)
     return 0
  /* below S mode */
  else if (S mode implemented && in U mode && senvcfg. SSE == 0)
    return 0
  else
    return 1
}

xSSCHK =
{
  if (xSSE == 0)
     return 0
  if (in M mode)
     return mseccfg. SSCHK
  else if (in S mode)
     return menvcfg. SSCHK
  else if (in U mode)
     return (S mode implemented) ? senvcfg.SSCHK : menvcfg. SSCHK
}

xSSUP =
{
  if (in M mode)
     return mseccfg.SSUP
  else if (in S mode)
     return menvcfg.SSUP
  else if (in U mode)
     return (S mode implemented) ? senvcfg.SSUP : menvcfg. SSUP
}

is_access_outside_iee_region(addr, size)
```

```
{
  access_last_offset = size - 1
  access_last = addr + access_last_offset
  if ((addr >= IEE_ENCRYPTION_BASE) && (access_last < mieeencend))
    return false
  return true
}
```

Exceptions

Exception	Code	Description
`SoftwareCheckException`	18	Synchronous exception which is triggered when there are violations of checks and assertions with regards to the integrity of software assets. The exact cause can be determined by examining the `xTVAL` register: • 3: Raised when `sspopchk` detects a tampered return address. • 4: Raised when the regular stack and shadow stack overlap (`ssp >= sp`). • 5: Raised when a shadow stack operation attempts to access memory outside the encrypted region.

Table 3.1.2.5-5: Exceptions causes added by the BCFI module.

3.1.2.6 Sub-Modules

The BCFI module requires submodules implementing a universal hash function (UHF) and its inverse (IUHF). The requirements for these modules are described in the following subsections.

UFH

The UHF module must fulfill the following requirements:

- The UHF module shall allow updating the 32-bit hash state by taking the previous 32-bit hash state and a 32-bit message (i.e., return value) as inputs. The updated hash state shall be available at the output.
- The UHF shall be parameterizable using the `ssuhfx` CSR.
- The UHF shall have a collision resistance close to the theoretical bound given by the birthday paradox.

IUHF

The IUHF module must fulfill the following requirements:

- The IUHF module shall allow to reconstruct the 32-bit message from the corresponding and previous 32-bit hash state.
- The IUHF shall be parameterizable using the `ssuhfxinv` CSR.

3.1.2.7 Debugging Strategy

The RISC-V debug module and NXP-AT's IEE module must be configured in the right way to enable correct debug functionality in the presence of the memory encryption engine required for the BCFI module. For more information, see the specification of NXP-AT's IEE module (Section 3.1.1).

Additionally, toolchain support is beneficial in control stack model to allow the debugger to unwind the stack correctly.

3.1.3 Context-Aware Performance Monitor Counter (CA-PMC) – TRT

Part of Task 3.1 Safety & Security Modules.

3.1.3.1 General Information

The Context Aware Monitoring framework is a set of IPs to enhance the monitoring IPs with context information and standardize the same monitoring IPs deployed in a system on a chip (SoC). The context information is typically defined by a context controller which typically is a core defining the context in which the events monitored are issued. The Context Aware Monitoring framework is composed of 4 different IPs (or IPs extensions): the CA-CORE, the CA-BUS, the CA-PMC (described in this section), and the CA-PMC-IF (Section 3.3.1).

3.1.3.2 Purpose and Scope

The CA-PMC's purpose, as a regular Performance Monitor Counter, is to count events of some kind (e.g., cache misses, cache hits, etc.), but providing means of filtering the counted events on some system defined context.

The CA-PMC implementation is highly dependent on the IP (e.g., cache) the CA-PMC is integrated in, but at minimum, in addition to the counter register it should have a register to store the context that should be used for filtering the events. The IP integrating the CA-PMC should have also the means to receive the context of the event from the source generating the event (e.g., cache read access), so the IP can transfer the event with the associated context to the CA-PMC.

In the context of ISOLDE, we will target the SoC caches as IP integrating the CA-PMC (and CA-PMC-IF to configure the CA-PMC and retrieve data from the CA-PMC).

3.1.3.3 Place in the System

The CA-PMCs are extended Performance Monitoring Counters to be placed in the different IPs of the system, as shown in the example Figure 3.1.3.3-1. They count the different events produced in the IP in dedicated registers. Unlike regular PMCs which count all the events that happen in the IP (e.g., all cache misses in a cache), the CA-PMCs allow the events to be filtered by a context. The actual meaning of the context depends on how the system is configured, but typically the context will refer to the initiator of the request that caused the event on the IP (e.g., the process that caused the cache miss event).

The CA-PMC only performs the event counting logic, configuration of the CA-PMC and reading of its event counting registers is performed through the CA-PMC-IF module (Section 3.3.1).

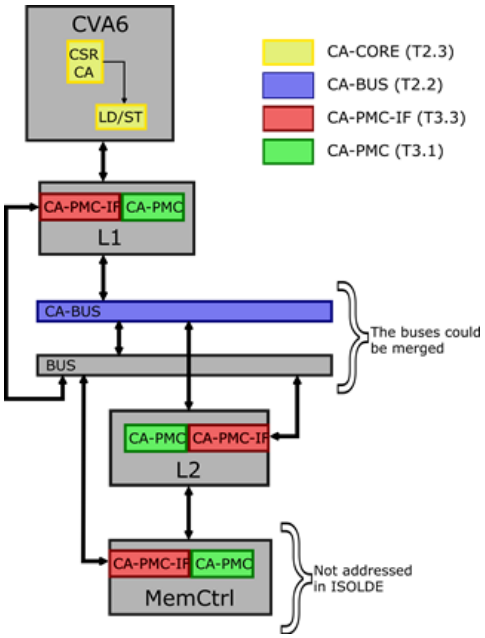


Figure 3.1.3.3-1: CA-PMC - Place in the Context Aware Monitoring infrastructure system

In the project's context, the Instruction and Data L1 Caches CA-PMCs will be targeted. Additional CA-PMCs might be developed be developed if time and effort are available.

3.1.3.4 Block Diagram

Figure 3.1.3.4-1 shows a high-level and preliminary view of the CA-PMC block diagram inside an L1 cache, but it could be any other IP, and how it is connected to the CA-PMC-IF module (Section 3.3.1) and the IP internals. The CA-PMC has multiple counter registers to monitor the events from the IP and can receive signals from the IP being monitored and the CA-PMC-IF module. Associated to each counter register there is a configuration register that defines:

- if the counter is active or not,
- which is the event that the counter register monitors,
- and which is the context to monitor.

A logic in front of each counter register manages the update of the counter register depending on the IP incoming event and associated context. If the incoming event id and context match the configuration register values, the counter register is updated. The configuration register and the counter register provide interfaces for the CA-PMC-IF module for reading and writing them. Based on this basic design enhanced implementations can be developed providing new functionalities like threshold configuration to generate interrupts when a counter reaches a value.

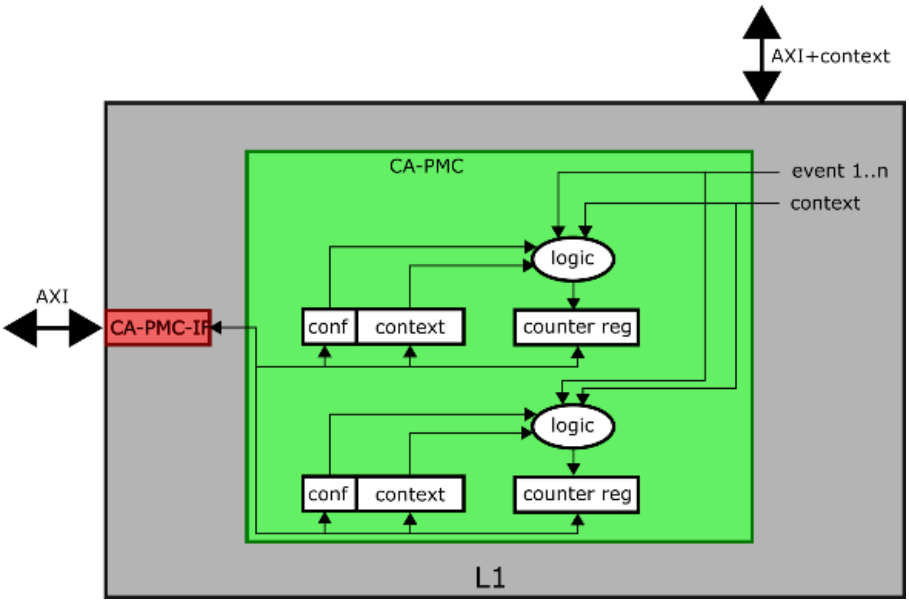


Figure 3.1.3.4-1: CA-PMC - Block diagram

3.1.3.5 Interfaces

The CA-PMC is connected with two different IPs (see Figure 3.1.3.4-1):

- A CA-PMC-IF module (Section 3.3.1) enabling the CA-PMC configuration and the read of the counter registers it contains, and
- the IP being monitored.

CA-PMC and monitored IP

Communication happens from the monitored IP to the CA-PMC. A basic interface defines N bit sized connections, one for each type of event the IP can generate, and a context-sized connection indicating the context of the request that generated the event. Both connections are clock synchronized.

IPs that can generate multiple events during the same cycle will require a queue to provide the events sequentially to the CA-PMC.

CA-PMC and CA-PMC-IF

A register write/read interface controlled by the CA-PMC-IF is required between the CA-PMC-IF and the CA-PMC.

3.1.3.6 Reset Strategy

At system reset the CA-PMC configuration registers should be initialized to not monitor any event. CA-PMC counter registers should be initialized to a default value (zero).

3.1.4 Cryptographically Tagged Memory (CTM) – NXP-AT

Part of Task 3.1 Safety & Security Modules.

3.1.4.1 Purpose and Scope

As explained in Section 3.1.2, memory corruption vulnerabilities are the most used path to gain remote control over digital devices. In Section 3.1.2, we introduced the BCFI module that helps to counter attackers exploiting memory safety issues to modify return addresses spilled on the stack. However, there are also other assets located on the stack, heap, or in global objects that are of interest to attackers, such as:

- Function pointers
- Values indirectly affecting the control flow (e.g., evaluated in conditional branches)
- Sensitive data like cryptographic keys

Fine-grained protection of such assets within the context of one task surpasses the capability of traditional protection measures like memory protection or management units. Memory tagging can be used to minimize this gap. It assigns additional metadata, the color, with memory blocks of a defined size. Every genuine pointer and its associated memory blocks are assigned the same statistically unique color at memory allocation. Therefore, only the designated pointer received from the allocator can be used to access the allocated data. This lock-and-key mechanism prevents out-of-bound accesses (spatial bugs) using another pointer or accesses using a dangling pointer (temporal bugs) as shown in Figure 3.1.4.1-1.

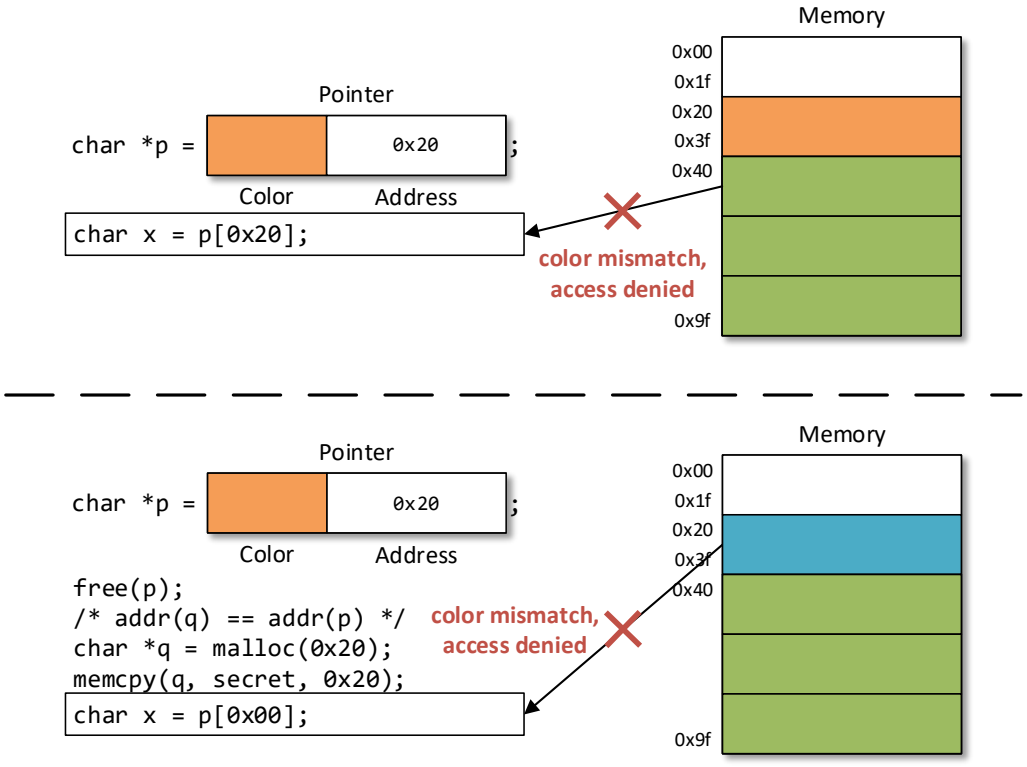


Figure 3.1.4.1-1: CTM - Memory tagging overview (top: out-of-bounds access, bottom: use-after-free)

Note that the colors must be stored additionally to the rest of the data increasing the overall memory usage. Therefore, existing memory tagging implementations, like Armv8.5-A MTE [arm2024], allocated only few bits for the colors, making them unsuitable for security purposes. Cryptographically tagged

memory [Nasahl2021] avoids this additional memory overhead by implicitly linking memory blocks with the genuine color. This implicit association is achieved by including the color in the tweak for memory encryption and decryption. Hence, this technique allows to use more bits for the color without increasing the memory overhead if a memory encryption engine is available. The major behavioral difference to the original scheme is that an attacker can still misuse pointers to access unintended memory blocks[5]. However, it is much harder to perform a successful attack as accesses triggered from a misused pointer will not be useful:

- Suppose misusing a pointer triggers an out-of-bounds read. Then, the decryption results are wrong as the color does not match ensuring the confidentiality of the corresponding data.
- Suppose misusing a pointer triggers an out-of-bounds write. Then, the color of the genuine pointer will not match the one of the misused pointers. Hence, the attacker cannot modify the data in a controlled way.

Until now, cryptographically tagged memory has only been applied to 64-bit processors where the designers used unimplemented bits in the virtual address space to store the colors. The CTM module adds cryptographically tagged memory to 32-bit processors without introducing the requirement of a memory management unit. The smaller address space poses additional challenges as no unused bits in pointers are available, and therefore, we developed a different strategy to include the colors in the pointer.

3.1.4.2 Place in the System

The CTM module is an ISA extension for RV32 cores without a MMU (microcontrollers) as can be seen in Figure 3.1.4.2-1. Further, it requires changes to the cache architecture and depends on a suitable tweakable memory encryption engine. These contributions are not described here, but in the section describing NXP-AT's IEE module (Section 3.1.1).

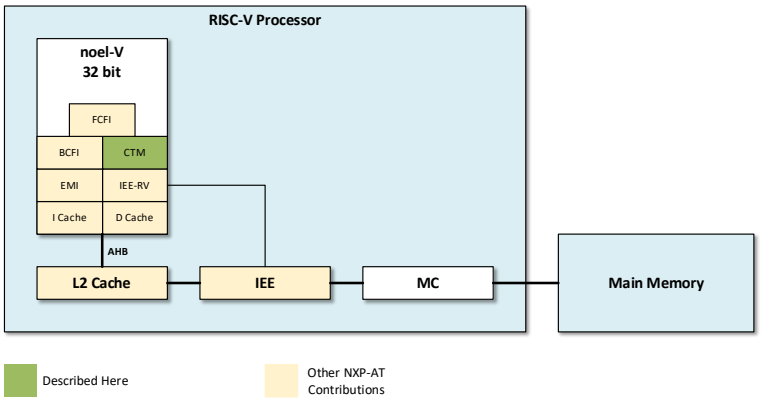


Figure 3.1.4.2-1: Overview of the CTM module in the system

3.1.4.3 Block Diagram

Figure 3.1.4.3-1 shows the internal architecture of the CTM module. The figure also indicates to which pipeline stages and modules inside the RV32 core the CTM module will be connected. The description

[5] In addition, such scheme can easily be turned into one which prevents the attacker from misusing pointers by using authenticated encryption, however, then again, additional storage is needed for storing the tags. Nevertheless, this might be already present.

assumes that the base core has 7 pipeline stages (instruction fetch, decode, register access, execute, memory, exception, write-back).

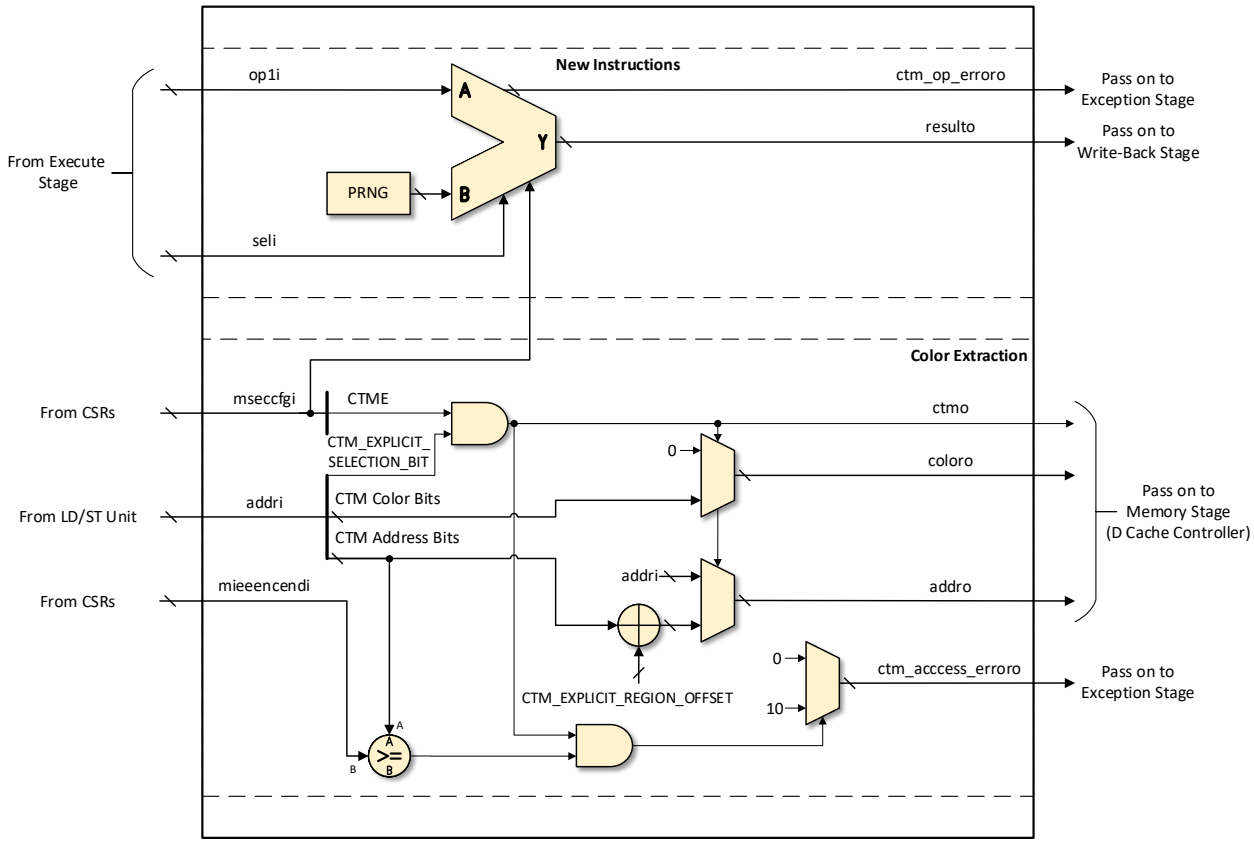


Figure 3.1.4.3-1: Internal architecture of the CTM module

The CTM module extends the ISA of the RV32 core with two instructions (`ctmtag`, `ctmuntag`). `ctmtag` adds a color to a pointer and `ctmuntag` removes a color from a tagged pointer. These new operations are depicted by the ALU symbol in Figure 3.1.4.3-1. The CTM module also includes a Pseudo-Random Number Generator (PRNG) as the colors should be statistically unique. Note that users can only assign colors to memory blocks with a size and alignment matching the cache line size. Hence, allocators must assure that the allocated region has the correct alignment and size, else the instructions will raise an exception. For more detailed information about the behavior of these instructions, see Section 3.1.4.4.

Further, if the CTM module is present, the RV32 core performs additional steps before memory requests are passed to the cache controller. As shown in Figure 3.1.4.3-1, the following steps are performed for CTM-protected addresses (bit at position `CTM_EXPLICIT_SELECTION_BIT` is set):

- The CTM module extracts the color and raw address (without color and bit at position `CTM_EXPLICIT_SELECTION_BIT`) from the address received from the LD/ST unit. It recomputes the address for the memory request as the raw address plus the `CTM_EXPLICIT_REGION_OFFSET` constant. The determined CTM state (`ctmo`), color (`coloro`) and address (`addro`) serve as input for the cache controller.
- The module checks if the raw address lies outside the memory region where memory encryption is active (above or equal to the value in the `mieeencend` CSR). If so, then it reports an error as CTM cannot protect memory blocks that do not go through the memory encryption engine.

For more information about the `CTM_EXPLICIT_SELECTION_BIT` and `CTM_EXPLICIT_REGION_OFFSET` constants, see Section 3.1.4.4.

The cache subsystem may hand over the memory access request to the main memory and its encryption engine. The tweak used by the memory encryption engine will include the forwarded color. After the request is completed, the color will be stored in the cache line alongside the decrypted data. For more information about the changes to the cache subsystem and the memory encryption engine, see NXP-AT's IEE module (Section 3.1.1).

3.1.4.4 ISA

The CTM module extends the ISA of the RV32 core. The next sections describe modified and new design parameters, CSRs, instructions, and exceptions. In the following, *L* denotes the cache line size used in the processor (either 32 or 64 bytes) and *rand()* indicates a value read from the PRNG. Note that the referenced `mieeencend` CSR is part of NXP-AT's IEE module and described there.

Design Parameters

Parameter	Default Value	Function
`CTM_EXPLICIT_SELECTION_BIT`	TBD[6]	Indicates which address bit selects between tagged and untagged pointers. The chosen address bit must not be used in the memory map of the processor.
`CTM_EXPLICIT_REGION_OFFSET`	TBD[6]	This value is ORed to the untagged address (replacing the zeroed color bits) before a memory access is performed. Hence, it can be used to move the start address of the CTM-protected region.
`CTM_TAG_MASK`	TBD[6]	Mask to indicate which address bits of a tagged pointer contain the color bits.
`CTM_ENABLE`	1	Determines if the Cryptographically Tagged Memory module is included in the design.

Table 3.1.4.4-1: CTM module design parameters

CSRs

In the following table, the privilege column gives the minimum required privilege mode (first letter) to access the CSR and which access types are allowed (last two letters). For example, M-RW means M-mode has read and write access, but lower privilege modes have no access.

[6] These parameters depend on the memory map of the system to which the CTM module should be added.

CSR	Index	Privilege	Field	Bit(s)	Function
mseccfg	0x747	M-RW	CTME	5	CTM enable for all privilege modes

Table 3.1.4.4-2: CSRs modified by the CTM module.

Instructions

Instruction	Function
ctmtag rd, rs ***Adds a random color to the value in rs and stores the result in rd.***	`if (mseccfg.CTME == 0) {` `  /* xCAUSE=2 */` `  raise IllegalInstructionException` `}` `if (rs & (1 << CTM_EXPLICIT_SELECTION_BIT) ||` `  rs & (L-1) != 0)` `{` `  /* xCAUSE=18, xTVAL=11 */` `  raise SoftwareCheckException(11)` `}` `if ((rs & CTM_TAG_MASK) != CTM_EXPLICIT_REGION_OFFSET ||` `  rs >= mieeencend)` `{` `  /* xCAUSE=18, xTVAL=10 */` `  raise SoftwareCheckException(10)` `}` `rd = (rs & ~ CTM_TAG_MASK) | (rand() & CTM_TAG_MASK) |` `  (1 << CTM_EXPLICIT_SELECTION_BIT)`
ctmuntag rd, rs ***Removes the color from the value in rs and stores the result in rd.***	`if (mseccfg.CTME == 0)` `{` `  /* xCAUSE=2 */` `  raise IllegalInstructionException` `}` `if (rs & (1 << CTM_EXPLICIT_SELECTION_BIT))` `{` `  addr = rs & ~(1 << CTM_EXPLICIT_SELECTION_BIT)` `  addr &= ~CTM_TAG_MASK` `  addr |= CTM_EXPLICIT_REGION_OFFSET` `}` `else` `{` `  /* xCAUSE=18, xTVAL=12 */` `  raise SoftwareCheckException(12)` `}` `if (addr >= mieeencend)` `{` `  /* xCAUSE=18, xTVAL=10 */` `  raise SoftwareCheckException(10)` `}` `rd = addr`

Table 3.1.4.4-3: Instructions added by the CTM module.

Exceptions

Exception	Code	Description
SoftwareCheckException	18	Synchronous exception which is triggered when there are violations of checks and assertions regarding to the integrity of

		software assets. The exact cause can be determined by examining the `xTVAL` register: • 10: Raised if a to-be-tagged (`ctmtag`) or untagged (`ctmuntag`, memory operations) address points to memory outside the defined CTM region. • 11: Raised by `ctmtag` if the address in `rs` is already tagged or not correctly aligned. • 12: Raised by `ctmuntag` if the address in `rs` is not tagged.

Table 3.1.4.4-4: Exceptions causes added by the CTM module.

3.1.4.5 Sub-Modules

The CTM module requires the availability of a PRNG.

PRNG

The CTM module includes a PRNG for generating the colors needed for the tagging of pointers. The PRNG must fulfill the following requirements:

- The entropy should at least match the (configurable) color size so that the probability of neighboring memory blocks with matching colors is as low as possible.
- Observing the color values of tagged pointers should not allow an attacker to predict future color values.

3.1.4.6 Debugging Strategy

The RISC-V debug module and NXP-AT's IEE module must be configured in the right way to enable correct debug functionality in the presence of the memory encryption engine required for the CTM module. For more information, see the specification of NXP-AT's IEE module (Section 3.1.1).

Additionally, toolchain support is beneficial as the debugger can only read the correct values from tagged memory blocks if the correct color is provided.

3.1.5 Enclave Memory Isolation (EMI) – NXP-AT

Part of Task 3.1 Safety & Security Modules.

3.1.5.1 General Information

On modern systems, often a range of workloads run simultaneously on the same physical general-purpose processor. Running diverse workloads on one processor is cost-efficient as it reduces hardware complexity and reuses the existing infrastructure as much as possible. However, it is often the case that the different workloads use assets with varying sensitivity levels. Users may not trust every workload, and the vendors of the different workloads may not trust each other. Hence, it must be prevented that one workload can leak sensitive information related to another workload. Traditionally, this isolation is guaranteed by enabling processors to execute instructions in different privilege modes and to limit which memory can be accessed by a workload (e.g., memory protection or management units). In such a setting, critical configuration settings, like memory access permissions, can only be accessed by trusted software (usually the operating system) running in higher privilege modes. Hence, the trusted software can isolate workloads by setting suitable memory access permissions before lowering the privilege mode and jumping to the scheduled workload. Switching from one workload to another is done during a so-called *context switch*. A context switch is usually triggered by a timer interrupt, which will cause the processor to enter the highest privilege mode and execute an appropriate handler. During a context switch, the trusted software performs the following steps:

1. Save the current processor state associated with the active workload.
2. Determine which workload shall be run next.
3. Restore the processor state of the workload scheduled to run next. The restored processor state also includes the appropriate memory access permissions. After this step, the execution will continue with the selected workload in the lower privilege mode.

The isolation of different workloads serves two main purposes:

- *Confidentiality*: A malicious workload cannot access any data of another workload.
- *Integrity***:** A malicious workload cannot modify another workload's data without detection.

However, research has shown that this logical isolation may be bypassed in malicious environments. For example, logical isolation is insufficient against a range of physical attacks:

- By physically probing the memory bus between the processor and external memory chip the attacker could leak sensitive data.
- Advanced techniques like laser fault injection could flip bits in memory modifying the behavior of software using these values [Roscian2013]. Note that there is also a software-based attack, called *Rowhammer* [Kim2014], that can achieve bit flips in DRAM.
- Researchers showed that attacking the context switch by injecting faults also breaks logical isolation [Nashimoto2021]. Their work presents a fault attack that skips instructions reconfiguring the memory access permissions for the scheduled workload, allowing the scheduled workload unintended access to data of the previous workload.

Adding a memory encryption engine that encrypts all data before it is stored in memory mitigates the first attack example, raises the bar for the second as injecting controlled modifications is harder, but does not help against the third example without additional countermeasures.

3.1.5.2 Purpose and Scope

As described in the previous section, logical isolation is insufficient to protect sensitive information in a malicious environment. Hence, the purpose of the EMI module is to enable workload-specific memory encryption for RISC-V cores together with NXP-AT's IEE module (Section 3.1.1). Further, the design of the EMI module mitigates the impact of fault attacks including skipping instructions during the context switch.

Using the EMI module, the confidentiality of the data associated with different workloads is protected cryptographically without reliance on logical isolation. Note that NXP-AT's IEE module (Section 3.1.1) only provides encryption without integrity protection to keep the memory overhead as small as possible. Hence, if used without logical isolation, malicious modifications of data associated with another workload are possible and cannot be detected directly. However, controlled modifications are only possible if a suitable value from a matching address is replayed, as the encryption tweaks will differ for every workload. Further, if a tweakable authenticated memory encryption engine is already available this can be incorporated with NXP-AT's IEE module (Section 3.1.1) and modifications can be detected.

Combining the classical logical isolation and the cryptographic isolation provided by the EMI module results in stronger security guarantees (e.g., mitigation of the previously explained attacks). These stronger guarantees benefit use cases requiring strong isolation between workloads like trusted execution environments. *Keystone* [Kohlbrenner2020] is an example of a RISC-V framework that allows the creation and management of trusted execution environments. In the case of Keystone, the untrusted workloads are run in isolated compartments called *enclaves* and interact with the main operation system called *host*. The enclaves run in privilege modes below M-mode and are managed by a trusted M-mode software called *security monitor* (SM). Figure 3.1.5.2-1 shows an overview of the different entities isolated by Keystone and which data they can access.

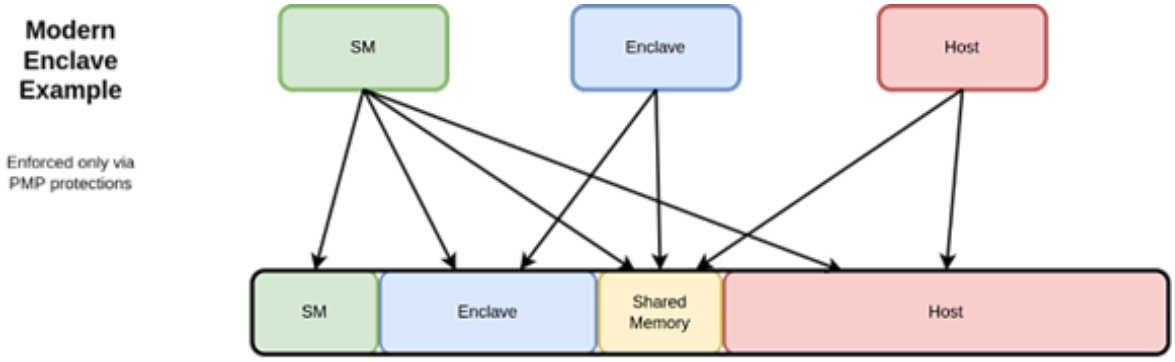


Figure 3.1.5.2-1: EMI - Entities isolated by Keystone.

3.1.5.3 Place in the System

The EMI module is an ISA extension for RV32 cores without a MMU (microcontrollers) as can be seen in Figure 3.1.5.3-1. Further, it requires changes to the cache architecture and depends on a suitable tweakable memory encryption engine. These contributions are not described here, but in the section describing NXP-AT's IEE module (Section 3.1.1).

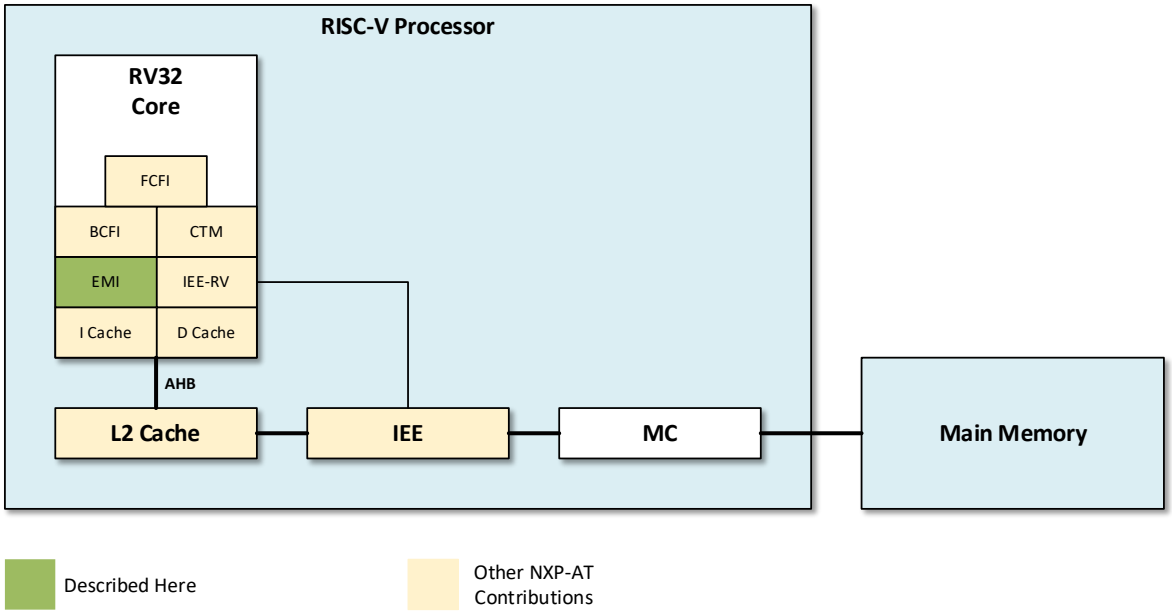


Figure 3.1.5.3-1: Overview of the EMI Module in the system

3.1.5.4 Block Diagram

Figure 3.1.5.4-1 shows the internal architecture of the EMI module. The symbol "||" denotes the concatenation of bits or bit vectors where the most significant bits are always the leftmost. The figure also indicates to which pipeline stages and modules inside the RV32 core the EMI module will be connected. The description assumes that the base core has 7 pipeline stages (instruction fetch, decode, register access, execute, memory, exception, write-back).

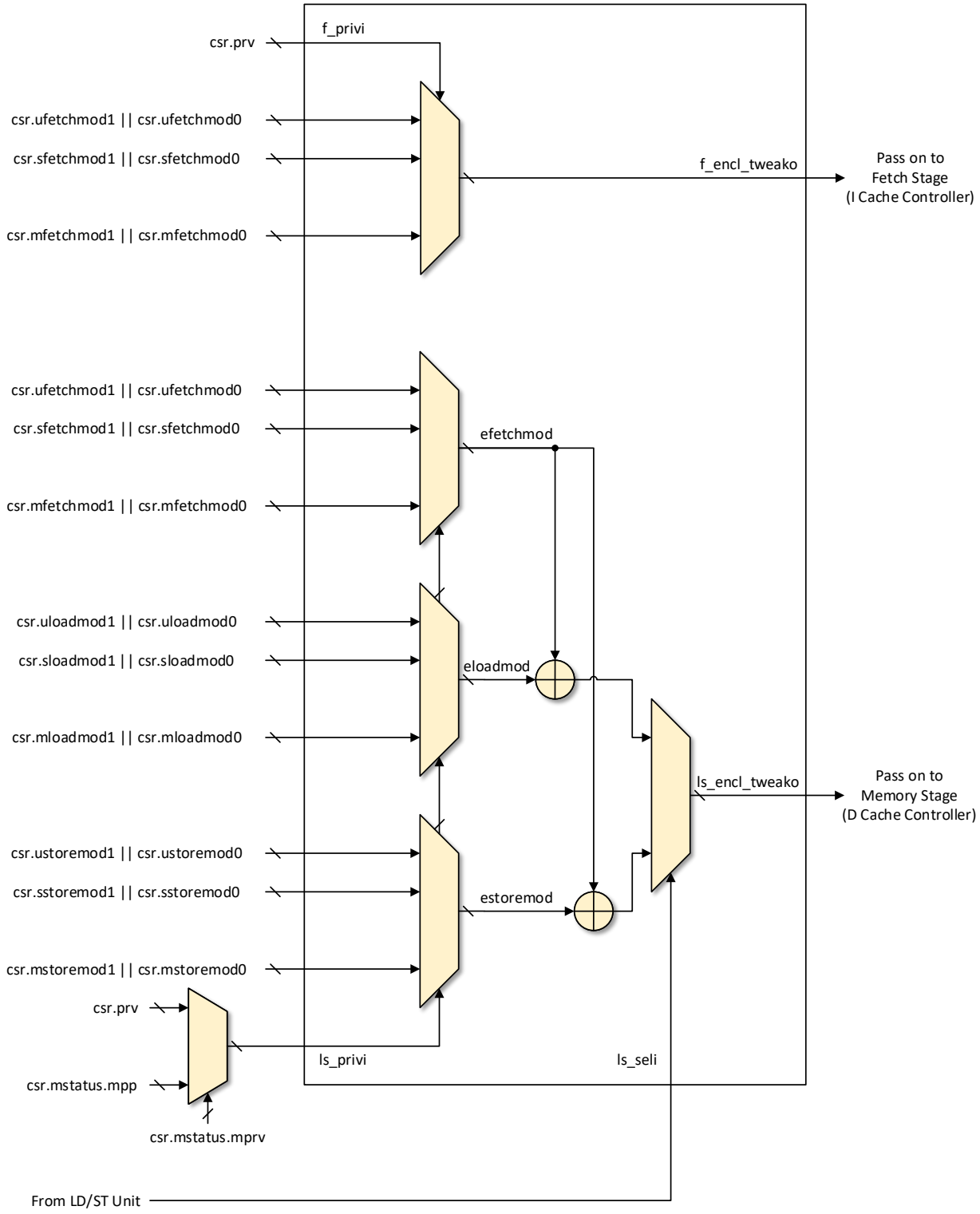


Figure 3.1.5.4-1: Internal architecture of the EMI module

As can be seen in the Figure 3.1.5.4-1, most of the inputs to the EMI module come from the CSRs including the modifiers that are used to build the enclave-specific tweaks used for memory encryption. Every modifier consists of two CSRs to provide sufficient protection against brute-force guessing attacks. Writes to modifier CSRs cause a pipeline flush if the modifier affects the execution of the current privilege mode. For more information about these CSRs, see Section 3.1.5.5. The `prv` CSR represents the privilege mode the processor is currently executing in. This CSR is only used internally and cannot be accessed with instructions. The last input comes from the LD/ST unit and determines whether memory access is a load or store. The effective fetch, load, and store modifiers are selected according to the effective privilege mode. For instruction fetches, the effective privilege mode (`f_privi`) always matches the processor's current privilege mode. However, the privilege mode can be overridden for load and store operations using the `MPRV` and `MPP` fields in the `mstatus` CSR [Waterman2021]. If `MPRV` is set, then the privilege level encoded in the `MPP` field is used for load and store operations instead of the processor's current privilege mode. We also comply with this behavior by using the resulting effective memory access privilege mode (`ls_privi`) to select the effective modifiers for load (`efetchmod, eloadmod`) and store (`efetchmod, estoremod`) accesses.

The output of the EMI module is an enclave-specific encryption tweak for fetch (`f_encl_tweako`; routed to the I Cache controller) and load/store operations (`ls_encl_tweako`; routed to the D Cache controller). The enclave tweak for fetch operations matches the effective fetch modifier. On the other hand, the enclave tweak for load or store operation is computed by combining the effective fetch modifier with the effective load or store modifier using an XOR operation. The fetch modifiers can only be configured by the trusted M-mode software, like the Physical Memory Protection (PMP) unit settings, while the other modifiers can also be written by lower-privileged software. Generating the enclave-specific tweaks according to this scheme has several advantages:

- By allowing the enclaves to modify their load and store modifiers, they can reconfigure the memory encryption to access shared memory regions. In such a case, the enclave needs to know the load and store modifiers of itself and all shared memory regions.
- The leakage of modifiers stored in a victim enclave does not enable another entity to access its memory or shared memory regions, as the enclave tweak computation for loads and stores also includes the effective fetch modifier. The effective fetch modifiers are only known to the trusted M-mode software and are statistically unique per enclave. Hence, without knowing the effective fetch modifiers, no load and store modifiers granting access to the victim enclave or its shared memory regions can be computed.
- The computation of the load and store enclave tweak as a combination of the effective fetch modifier with the effective load or store modifier also leads to architectural resilience against fault attacks aiming to skip instructions during context switches:
    - Suppose the switching of the load or store modifier is skipped. In that case, the combination of genuine fetch and previous load or store modifier will not match the needed value for accessing the memory of the previous enclave. Hence, an attacker can neither leak data from the prior enclave nor inject controlled modifications.
    - The only way to get an enclave tweak suitable for leaking or modifying the previous enclave's data would be to skip the switching of the fetch and load or store modifier. However, then the fetch modifier will not match the value required to decrypt the instructions belonging to the scheduled (attacker) enclave. Hence, the instructions in memory will be decrypted wrongly, and the execution will likely quickly end in an illegal instruction fault.

The final enclave tweak is stored as part of the cache lines[7], passed to the memory controller by the cache subsystem and used by the memory encryption engine to cryptographically isolate the enclaves from each other. Note that this also implies that the protected memory regions must at least be cache line size aligned. For more information about the changes to the cache subsystem and the memory encryption engine see NXP-AT's IEE module (Section 3.1.1). Figure 3.1.5.4-2 demonstrates the entity isolation provided by Keystone enhanced with EMI using the same example as in Figure 3.1.5.2-1. For example, considering the SM, we can see that in addition to the existing logical isolation by the PMP (as shown in Figure 3.1.5.2-1), the correct encryption tweak must be set for accessing memory regions (e.g., „SM tweak" for accessing the memory region associated with the security monitor or „Host tweak" for accessing the memory region associated with the host). If the set encryption tweak does not match the one associated with the memory region, then reads return random content, and controlled writes are not feasible.

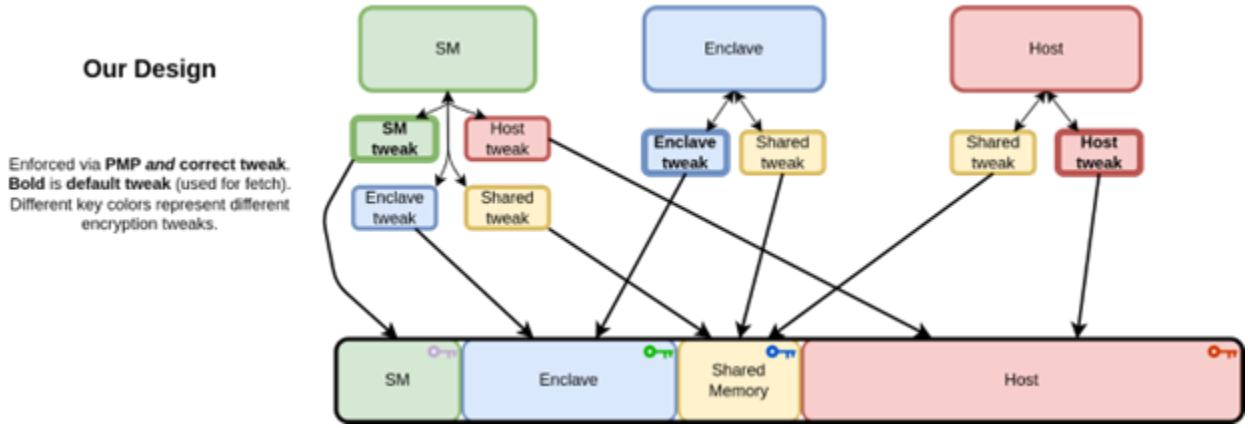


Figure 3.1.5.4-2: Entities isolated with Keystone enhanced with EMI.

3.1.5.5 ISA

The EMI module extends the ISA of the base RV32 core. The next sections describe modified and new design parameters, CSRs, instructions, and exceptions.

Design Parameters

Parameter	Default Value	Function
`EMI_ENABLE`	1	Determines if the Enclave Memory Isolation module is included in the design.

Table 3.1.5.5-1: EMI module design parameters

CSRs

[7] In practice, the cache lines do not need to store the entire tweaks which would result in a large overhead, but rather would store a compressed version in the cache lines themselves and resolve those compressed versions via a lookup table before forwarding them to memory. This in turn requires an appropriate replacement strategy for the tweaks in the lookup table and an invalidation of certain cache lines in case a tweak is not present in the table.

In the following table, the privilege column gives the minimum required privilege mode (first letter) to access the CSR and which access types are allowed (last two letters). For example, M-RW means M-mode has read and write access, but lower privilege modes have no access.

CSR	Index	Privilege	Bits	Function
mloadmod0	TBD[8]	M-RW	31-0	LSW of load modifier for M-mode
mstoremod0	TBD[8]	M-RW	31-0	LSW of store modifier for M-mode
mfetchmod0	TBD[8]	M-RW	31-0	LSW of fetch modifier for M-mode
mloadmod1	TBD[8]	M-RW	31-0	MSW of load modifier for M-mode
mstoremod1	TBD[8]	M-RW	31-0	MSW of store modifier for M-mode
mfetchmod1	TBD[8]	M-RW	31-0	MSW of fetch modifier for M-mode
sloadmod0	TBD[8]	S-W[9]	31-0	Alias to u[load,store,fetch]mod[0,1]
sstoremod0	TBD[8]	S-W[9]	31-0	
sfetchmod0	TBD[8]	M-RW	31-0	
sloadmod1	TBD[8]	S-W[9]	31-0	
sstoremod1	TBD[8]	S-W[9]	31-0	
sfetchmod1	TBD[8]	M-RW	31-0	
uloadmod0	TBD[8]	U-W[9]	31-0	LSW of load modifier for U-mode
ustoremod0	TBD[8]	S-W U-W[9]	31-0	LSW of store modifier for U-mode
ufetchmod0	TBD[8]	M-RW	31-0	LSW of fetch modifier for U-mode
uloadmod1	TBD[8]	S-W U-W[9]	31-0	MSW of load modifier for U-mode

[8] These indices depend on the free CSR addresses in the processor to which the EMI module should be added.

[9] Those CSRs return zero when they are read in U or S-mode preventing unintended leakage of the modifiers.

ustoremod1	TBD[8]	S-W U-W[9]	31-0	MSW of store modifier for U-mode
ufetchmod1	TBD[8]	M-RW	31-0	MSW of fetch modifier for U-mode

Table 3.1.5.5-2: CSRs added by the EMI module.

Table 3.1.5.5-2 shows that the S-mode CSRs are only aliases for the corresponding U-mode CSRs in the proposed design. This aliasing simplifies the design for our envisioned use case of enclaves running in privilege modes below M. In such a setting, differentiating between S- and U-mode is not required as the isolation shall be enforced between different enclaves. The advantage of this simplification is a reduced implementation overhead, as no physical registers for the S-mode CSRs are required. We kept the U-mode CSRs to allow the workload to switch its load and store modifiers in U-mode, enabling quick access to shared memory sections. This design choice matches the PMP, which is configured by M-mode software, and afterward the resulting memory access permissions apply to all privilege levels below M.

3.1.5.6 Debugging Strategy

The RISC-V debug module and NXP-AT's IEE module must be configured in the right way to enable correct debug functionality in the presence of the memory encryption engine required for the EMI module. For more information, see the specification of NXP-AT's IEE module (Section 3.1.1).

Additionally, if the confidentiality of the enclaves shall be ensured, the devices must disable external debugging before enclaves are provisioned (can be ensured by an appropriate lifecycle). Otherwise, debugging can leak any enclave's information as the debug access cannot be limited to specific enclaves using the standard debug module. If debugging access is still required, either a self-hosted debug implementation with an agent in the trusted software that can enforce the restriction of access needed or an adapted debug module is required.

3.1.6 Forward-Edge Control Flow Integrity (FCFI) – NXP-AT

Part of Task 3.1 Safety & Security Modules.

3.1.6.1 General Information

Systems operating in potentially malicious environments are subject to logical and physical attacks. Fault injection attacks based on optical, electromagnetic, clock, or voltage glitches are a form of active physical attacks. These attacks can modify or skip instructions, altering the control flow and bypassing security checks. Research demonstrated various successful attacks exploiting instruction skips introduced by fault attacks. Among those are bypassing signature verification to load malicious firmware [Buhren2021] or skipping the reconfiguration of memory protection units to gain access to protected data [Nashimoto2021].

Note that we cannot share detailed information in this public deliverable as no patent has yet been filed for the FCFI scheme. Hence, only high-level information is presented, and the other sections have been removed.

3.1.6.2 Purpose and Scope

The FCFI module ensures the integrity of the instruction stream by calculating a running checksum over the executed instructions and regularly comparing the current checksum value with pre-computed reference values. If a mismatch is detected, then a software integrity violation is raised. Hence, the FCFI module allows the detection of attacks that modify the instruction stream. The FCFI module can also detect modifications of forward-edge control flow transfers (indirect jumps and calls). However, it cannot protect backward-edge control flow transfers (function returns). For this purpose, it can be combined with NXP-AT's BCFI module (Section 3.1.2).

3.1.6.3 Place in the System

The FCFI module is an ISA extension for RV32 cores without a MMU (microcontrollers) as can be seen in Figure 3.1.6.3-1.

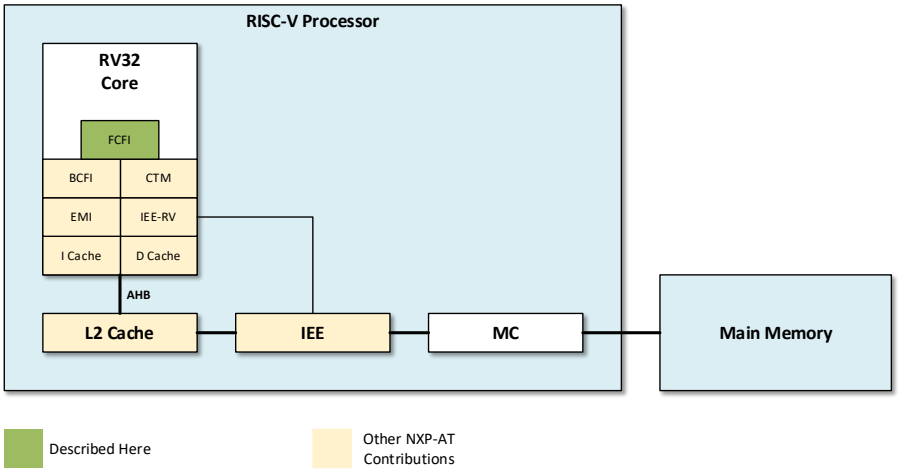


Figure 3.1.6.3-1: Overview of the FCFI module in the system

3.1.6.4 Sub-Modules

The FCFI module requires a submodule for computing and updating a checksum.

3.1.6.5 Debugging Strategy

If the RV32 core is in debug mode, then the FCFI module is suspended to allow executing instructions in the debug ROM and program buffer without integrity violations. The FCFI module is resumed after leaving debug mode.

3.1.7 Memory Subsystem Support for Bytecode VMs – HM

Part of Task 3.1 Safety & Security Modules.

3.1.7.1 General Information

The memory subsystem support for the acceleration of bytecode VMs concerns the data memory path of the microarchitecture. To map stack-based machines onto RISC-V register machines, hardware-based stacks offload the stack push and pop operations. Besides that, the integration of memory tagging operations is under investigation as part of the module.

3.1.7.2 Purpose and Scope

The module accelerates the data memory interaction of the bytecode VM implementation. It accelerates both interpreter implementations and ahead-of-time compiled code. As stack accesses have a significant impact on the execution, a hardware-assisted data stack can accelerate the execution of both interpreters and ahead-of-time compiled code. The subsystem integrates with the main core pipeline and can be accessed from other accelerators, such as bytecode execution units.

3.1.7.3 Place in the System

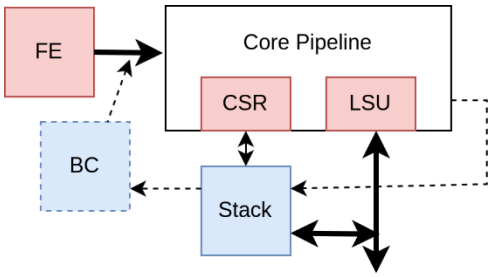


Figure 3.1.7.3-1: Memory Subsystem Support for Bytecode VMs – Place in the system (FE = Fetch, CSR = Control and Status Register Interface, LSU = Load Store Unit, BC = Bytecode Translator)

The module (“Stack”) in Figure 3.1.7.3-1 integrates with the core pipeline primarily via CSR registers. The prime use case is to load and store the top of the stack via a dedicated CSR. The module has a configurable size of the hardware stack that contains the top elements. It spills between the hardware stack and the memory autonomously from the pipeline (ideally without backpressure). The width of the spilling is configurable usually through the cache block size or through the memory width.

A second port to the stack allows other modules to also interact with the stack. In the case of a hardware-assisted bytecode execution (dashed), the bytecode generator can pop values from stack and generate instructions that write back to the module (which requires deep integration into the pipeline).

3.1.7.4 Block Diagram

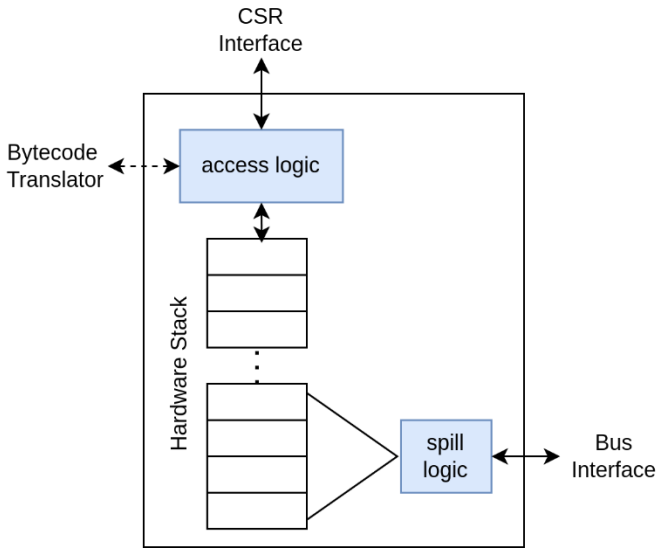


Figure 3.1.7.4-1: Memory Subsystem Support for Bytecode VMs – Block diagram

The block diagram 3.1.7.4-1 shows the internal structure of the module. The spill logic and the access logic are state machines that interface the outside world, the stack itself is a shift register, controlled by both interfaces.

3.1.7.5 Interfaces

The module has three interfaces.

CSR Interface

There are two CSR registers:

- `CSR_BCVM_STACKADDR`: The base address of the stack in memory. It is used to configure the stack address of the currently running module.
- `CSR_BCVM_TOP`: Push and pop operands from hardware stack.

Data Memory Interface

Access to the data memory interface, which can be multiplexed on the memory interface. An Advanced eXtensible Interface 4 lite (AXI4lite) interface is supported for convenience.

Core Integration (optional)

The push and pop access can also be done via two simple handshake channels (`ready`, `valid`, `data`).

3.1.8 Safety-Related Traffic Injector (SafeTI) – BSC

Part of Task 3.1 Safety & Security Modules.

3.1.8.1 General Information

The SafeTI is a flexible and programmable traffic injection hardware module to enable exhaustive timing verification and validation of powerful Multiprocessor System-on-Chips (MPSoCs) for safety-critical systems. In particular, BSC's latest version comes along with an increased number of features and an improved architecture that have been contributed as part of the work in ISOLDE.

3.1.8.2 Purpose and Scope

SafeTI is designed to inject programmable traffic in on-chip interconnects. SafeTI allows programming arbitrary traffic patterns where multiple parameters can be configured, such as read/write requests, data size sent/received, burst length, inter-request delays, repetitions per request, sequence of requests, etc.

Traffic pattern programming is devised to keep the memory footprint low to reduce the internal storage needed and speed SafeTI programmability up. Moreover, traffic pattern descriptors have been devised, enabling future extensions.

SafeTI is implemented as a pipelined module to enable high injection rates without unnecessary delays between consecutive requests.

SafeTI is designed to ease its portability across different communication interfaces like AMBA AHB, AMBA AXI and others. We provide its realization for an AMBA AHB interface.

SafeTI is integrated in an FPGA-based MPSoC from Frontgrade Gaisler AB, based on RISC-V NOEL-V cores.

3.1.8.3 Place in the System

The SafeTI is an AMBA AHB and AXI compliant module for traffic injection. It is intended to be connected to these two types of interfaces, and it is particularly useful if those interfaces have either multiple managers or are connected to subordinates receiving requests from multiple managers. For instance, its best placement is as part of the interface connecting the cores and/or accelerators with the shared caches or memory controllers, as illustrated in the schematic in Figure 3.1.8.3-1. This way, the predefined traffic can be injected to test a variety of timing and functional behavior controllably.

SafeTI's programming port is compliant with the AMBA Advanced Peripheral Bus (APB), although it will be extended to AMBA AXI in the future.

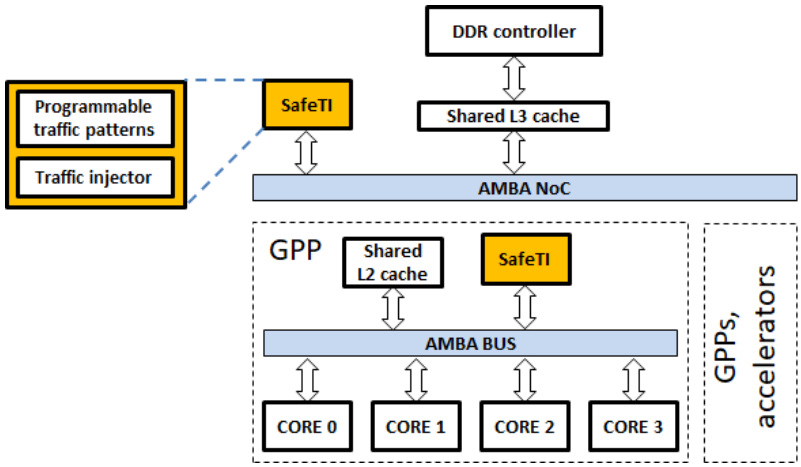


Figure 3.1.8.3-1: SafeTI - Place in the system

3.1.8.4 Block Diagram

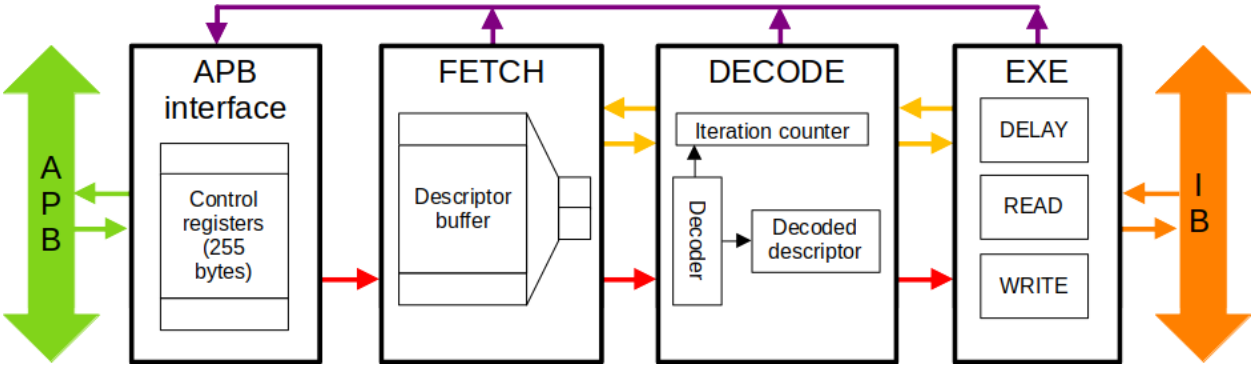


Figure 3.1.8.4-1: SafeTI - Block diagram

The SafeTI has a set of control registers programmed through an APB interface. Those control registers are the ones allowing to program the descriptor buffer, which stores microprogrammed sequences of traffic patterns to be injected by the SafeTI into the injection interface (IB in the Figure 3.1.8.4-1), namely an AXI or AHB interface.

The injection pipeline of the SafeTI works as follows: once the next descriptor is fetched (they are fetched from the descriptor buffer analogously to instructions from memory in a computing core), it is decoded. A descriptor pointer indicates the next descriptor to fetch. The iteration counter in the descriptor indicates whether the next descriptor needs to be fetched next, or whether the current descriptor needs to be used again (e.g., to inject repeated traffic). Using the information of the decoded descriptor, the traffic injection stage generates the traffic to inject (read or write, with a given data transaction size, whether in burst mode or not, etc.). Note that, if the descriptor is a delay descriptor, no traffic is injected until the indicated delay elapses.

3.1.8.5 ISA

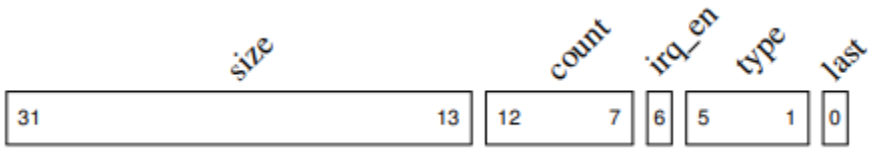


Figure 3.1.8.5-1: SafeTI as microprogrammed and memory mapped module using its own descriptor format.

The SafeTI is a microprogrammed and memory mapped module using its own descriptor format, which we illustrate in Figure 3.1.8.5-1.

Each descriptor type features a different word length and field encoding to accommodate the programmable parameters required by the action to be carried out. However, every descriptor shares the first descriptor word format specified in the figure, providing a compatibility layer in the descriptor format for lighter implementation. Changes to the first descriptor word fields are considered in future descriptor type expansions. Fields `size` and `count` could be modified for new descriptor types due to being action specific, whereas fields like `irq_en`, `type` and `last` are considered immutable, to maintain the compatibility layer.

The `size` field encodes the number of bytes to access for READ and WRITE descriptor types or the number of clock cycles needed to wait for the DELAY descriptors. The `count` field encodes the number of times the descriptor's execution must repeat. Thus, the same operation is executed *(`count` + 1)* times. The `irq_en` bit allows the SafeTI to send an interruption through the APB interface upon descriptor completion. Finally, the `last` bit is used to finalize the injection pattern at a specific descriptor completion (which disables the traffic injector if QUEUE mode is disabled) or restarts the execution from the first descriptor.

Descriptor types READ, WRITE, READ_FIX and WRITE_FIX include a second 32-bit word used as a starting address where to perform the access operation. Should an invalid address be programmed, the traffic injector behavior depends on the network response to complete the access with an error (e.g., lack of permission, non-existing, etc.) and resumes traffic injection.

3.1.8.6 Interfaces

AMBA AHB/AXI interface

The AHB or AXI interface is a manager interface used to inject traffic. It is fully compliant with the specification of the corresponding protocol. Note that, in general, a SafeTI instance supports only one of those interfaces and injects traffic accordingly.

AMBA APB interface

Hardware interface

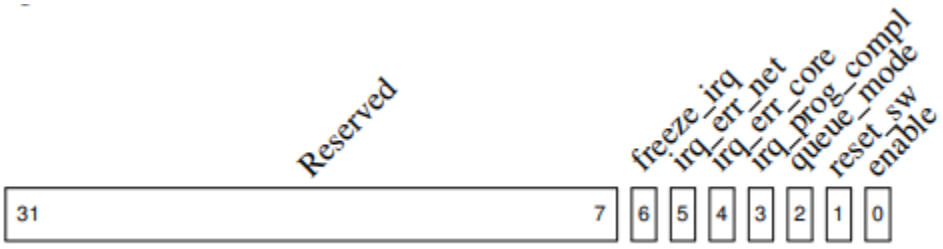


Figure 3.1.8.6-1: SafeTI - Hardware interface

The AMBA APB subordinate interface is used to program the control registers of the SafeTI, and to store descriptors in the internal descriptor buffer. An address space of 256 bytes is reserved for the APB interface, with such addresses being set at integration time.

The APB interface only supports single 32-bit accesses for setting the configuration register (0x00), shown in Figure 3.1.8.6-1, and descriptor word input feed register (0xFC) for programming the injection pattern.

The SafeTI programming process consists of word by word writing each descriptor, in execution order. Descriptors to be written are obtained through the APB descriptor feed register stored in the descriptor buffer, which is part of the FETCH stage. Once the desired injection patterns have been programmed, the injector may be configured and then initialized.

The `reset_sw` bit is asserted when a new injection pattern needs to be loaded. The current pattern is wiped out, and all circuits are reset except those related to transactions in process. This is necessary to allow for the correct termination of ongoing transactions. On the other hand, the hardware reset puts all circuits in a default state without exceptions. Yet, note that the hardware reset is a SafeTI signal not visible to the software layers.

SafeTI module features several interruption flags that are propagated through the APB interface. These include interruptions raised due to a network error, generated when the injection network answers with an error, due to an internal error caused by an unsupported encoding, or due to injection pattern completion. Furthermore, descriptor completion can also trigger an interruption, programmed on the first descriptor word as explained before.

SafeTI also features an automatically disabling mechanism, which triggers an interruption by asserting the `freeze_irq` flag to notify that it has been disabled. SafeTI is disabled by means of a hardware breakpoint of the traffic pattern execution. The conditions that can trigger the interruption are configured by asserting the `irq_err_net`, `irq_err_core` and `irq_prog_compl` for network error, SafeTI error, or injection pattern completion respectively.

SafeTI can be set in QUEUE mode by asserting the `queue_mode` flag so that the injection pattern execution loops to the first descriptor after completion. The `freeze_irq` flag overrides the QUEUE mode, meaning that upon the right conditions the traffic injector is disabled, even if SafeTI is configured to work in QUEUE mode.

Software Interface

The control register of the SafeTI, as well as the descriptor word input feed register used for SafeTI configuration must be modified only by software components with appropriate privileges. To realize this, the SafeTI registers are mapped in specific physical addresses upon integration in the platform, and the hypervisor (FENTISS' XtratuM in the particular case of the SafeTI integration in ISOLDE) is in charge of managing privileges to allow only specific partitions updating and reading of the SafeTI's registers.

The preferred configuration consists of allowing only a single partition to modify the SafeTI's registers, whereas the other partitions cannot access them. XtratuM guarantees this behavior leveraging the MMU existing in the NOEL-V cores. This MMU also implements the RISC-V ISA hypervisor extension.

Overall, the XtratuM hypervisor provides memory space isolation for the SafeTI's registers, hence achieving freedom from interference, in line with safety standards guidelines for items with integrity requirements.

3.1.8.7 Clocking Strategy

SafeTI is designed to share the same clock signaling used for the injection interface, whose input port is labeled as `clk`. The module does not allow yet for different clocking regions between the programming (APB) and injection (AHB or AXI) interfaces.

3.1.8.8 Reset Strategy

The SafeTI integrates two reset methods, the hardware active low reset signal through the input port `rstn`, and the aforementioned software reset flag `reset_sw` through the APB programming interface. The hardware reset completely wipes all data from both SafeTI and injection interface, resetting the module to a blank state and interrupting any on-going transaction. The software reset clears APB registers, sets descriptor buffer and stage registers to their initial state, and lets the injection interface operate independently to complete any on-going transactions.

3.1.8.9 Verification Strategy

The SafeTI verification strategy incorporates a custom testbench for the simulation environment, generating expected and unexpected communication from the injection interface with adjustable degrees of tolerance (e.g., answer at different clock cycle). Additionally, SafeTI includes a number of counters (e.g., number of transactions requested) to provide an increase in observability for debugging on FPGA environment.

3.1.9 Safety and Security Control Unit – IFX

Part of Task 3.1 Safety & Security Modules.

3.1.9.1 Purpose and Scope

The safety controller shall collect the kind and number of on-chip detected errors, pre-process them, and trigger follow-up actions. Error detection and follow-up actions are out of scope of the safety controller; the errors are reported, and follow-up actions are triggered by the safety controller.

3.1.9.2 Place in the System

The safety controller is intended to be SW programmable, thus it shall support a bus interface; at the moment this is AHB. Further, the safety controller is connected to IP-blocks as shown in Figure 3.1.9.2-1 or blocks onside the IPs' implementation. Via this connection, detected and corrected errors are reported separately to the safety controller. Thus, also a corrected and an additionally occurring but not corrected error can be reported. Further, the Safety Controller is connected to an interrupt mechanism, which is here an external device.

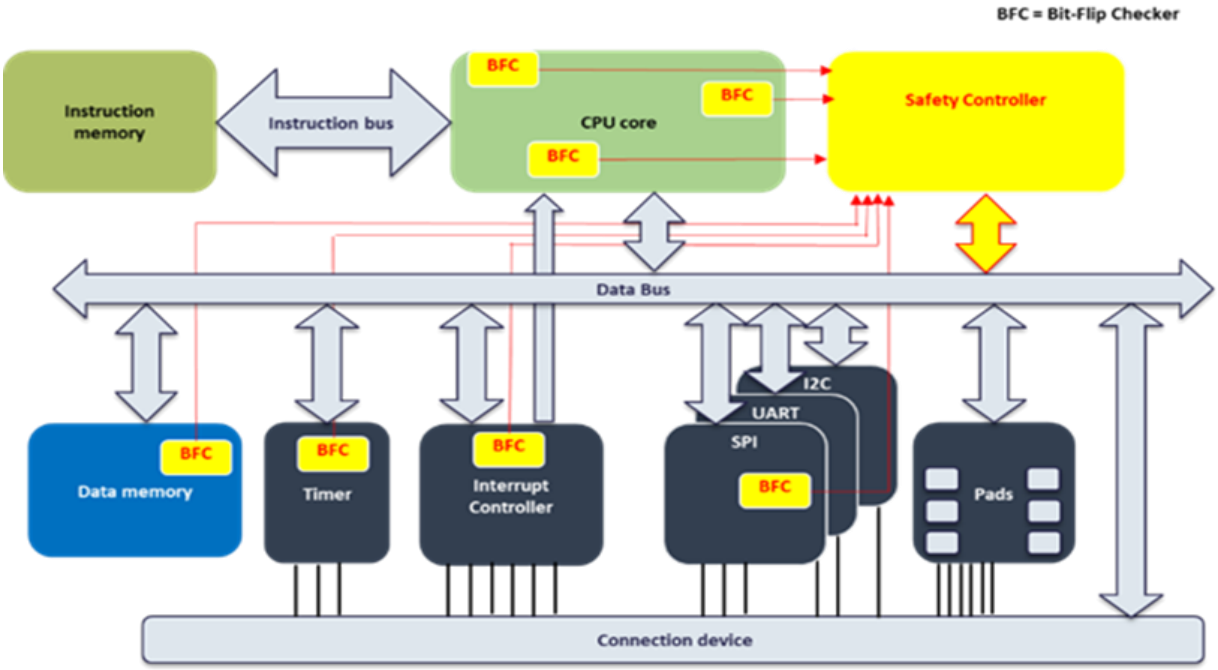


Figure 3.1.9.2-1: Safety Controller in a SoC

3.1.9.3 Block Diagram

The safety controller is depicted in Figure 3.1.9.3-1. It possesses signal interfaces to wires reporting on errors. These inputs are pre-processed in two error analyzers. One for not corrected errors (error analyzer) and one for corrected errors (corrected error analyzer).

The error analyzer contains channels for each signal wired to the safety controller. The analyzed errors are forwarded to the action request unit, which manages the action request. All three subcomponents are controlled via values written in bit fields. They report the status by reading the bitfields.

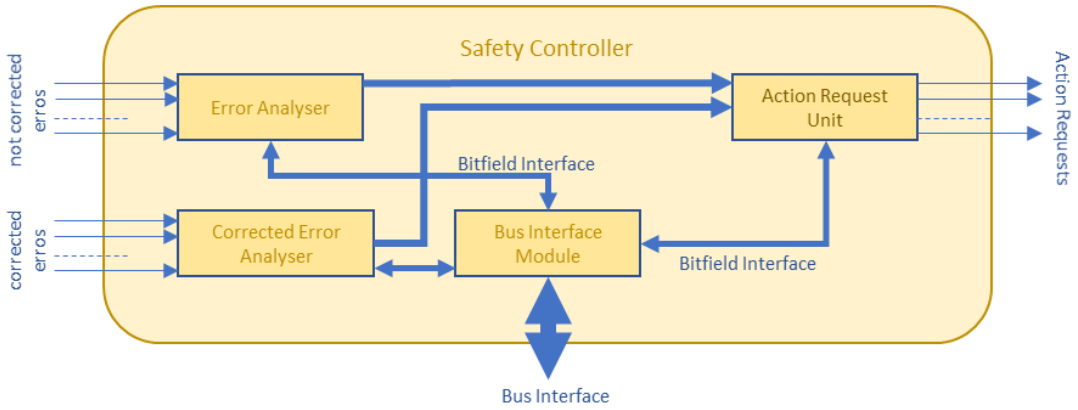


Figure 3.1.9.3-1: Safety Controller

3.1.9.4 ISA

There are no special instructions planned. In case of tight coupling, the safety control unit may be SW accessed via CSRs. The address of the CSRs is not defined as this is an unsupported option.

The software-based error actions use interrupt signals. More precise alignments may need to enhance the CLIC related definitions in the RISC-V specification.

3.1.9.5 Interfaces

The interfaces consist of error detection signals, safety action trigger signals, a bus interface and the clock/reset interface.

AMBA AHB/ABP Interface

The bus interface is AHB or APB.

Error Detection Signals

Error detection signals are level sensitive. The report is high active. If more than one error shall be reported at a time, the number of errors is binary encoded in a vector of error detection signals. In both cases a low value on a single wire or the number zero reports “no-error”.

Safety Action Trigger

The safety action trigger is assumed to be positive edge sensitive. Level and handshake-based signals / signal pairs are an alternative but are not supported.

Clock / Reset Interface

The clock is assumed to trigger with rising edge and the reset is assumed to be asynchronous low sensitive. Further, there should be no clock edge when the reset is active.

Clock and reset are planned as separate signals but may be grouped in an interface.

3.1.9.6 Sub-Modules

The submodules are the error analyzer, which is instantiated twice, the action request unit, and the bus interface module.

(Corrected) Error Analyzer

Error analyzers count the arriving errors and trigger the action request unit only if a threshold is reached. Optionally, every error can be forwarded to the action request unit.

The error count, the error threshold and a reset of those values is controlled by bitfields.

Action Request Unit

The action request unit merges error requests and raises error requests to the outside. It also handles the protocol to the unit(s) that handle the action requests. The action request handler is controlled via bitfields.

Bus Interface Module

The bus interface enables the software by memory mapped IO to access the bitfields in the units.

3.1.9.7 Software based Error Handling

It is intended to use a RISC-V CPU core for handling detected errors. Other solutions are open as well. For the intended case, action trigger signals from the safety controller are connected to an interrupt controller or directly the exception unit.

RISC-V Trap Handling

Exceptions and interrupts are crucial concepts in the RISC-V architecture, providing mechanisms to handle unexpected events or changes in the system's state. In this section, we will cover what exceptions and interrupts are, briefly describe how they are handled in RISC-V and discuss how exceptions and interrupts can be used to ensure system reliability.

Exceptions and interrupts are events altering the normal program execution flow. The first ones are related to synchronous events, while the second ones are asynchronous. In other words, exceptions are tied to a specific instruction, while interrupts are not. Illegal instructions, misaligned addresses, and memory access faults are typical exceptions, while interrupts can be triggered by peripheral devices, timers, and other cores. The mechanism to handle exceptions and interrupts is called a "*trap*." When an exception or interrupt occurs, the processor takes a "trap" to a predefined location in the memory, known as the "trap handler." To increase readability, we'll from now on use the word "traps" to refer to both exceptions and interrupts. The trap handler is responsible for saving the current state of the processor, handling the trap, and restoring the processor state before resuming normal execution.

The following key registers are used to handle traps:

- `mcause`
  Encodes the specific cause of the trap (load access fault, timer interrupt, external interrupt, etc.) so that the trap handler can take appropriate measures.
- `mepc`
  Stores the address of the instruction that caused the trap for exceptions, and the address of the instruction that would have been executed next for interrupts. It helps returning to the regular program execution once the trap has been handled.

- **`mtvec`**
  Contains the address of the trap handler where the CPU needs to jump. It is typically written during the system boot.
- **`mtval`**
  Holds additional information related to the trap to handle it better. For instance, this value could be the faulting address in case of a load access fault.

When a trap occurs, the processor saves the current program counter (PC) in the `mepc` register and sets the `mcause` register accordingly. The `mtval` register is updated if additional information is required. The processor then jumps to the trap handler address stored in `mtvec`, processes the trap, and resumes normal execution by setting the PC to the `mepc` value.

Some traps are already defined by RISC-V, such as the ones mentioned above. These traps define typical unexpected behaviors that can happen during the execution of a program and handling them is particularly important in safety-critical systems. Furthermore, some free space is left to define custom traps and can therefore be related to on-chip faults detected by various mechanisms (DMR, TMR, ECC, etc.). In conclusion, traps appear as a robust and flexible mechanism for ensuring system reliability and fault tolerance. The hardware unit responsible for setting the registers and disrupting the regular execution of the program is called the “Exception Unit.”

Exception Unit as Safety Controller

As discussed in the previous section, traps are suitable mechanisms for safety-critical applications. Safety mechanisms can be connected to whether trigger a trap directly or to increment a counter that will trigger a trap when reaching a predefined value. Self-tests can also be executed inside a dedicated trap handler, and the Exception Unit will resume the program execution when done. The Exception Unit can therefore be seen as a Safety Controller. Now that we explained the concept, this part will focus on explaining the capabilities of our Exception Unit.

The design of the Exception Unit follows a model-driven approach where a single implementation leads to different configurations, depending on the user inputs. In this regard, each trap can be enabled or disabled separately, and if a trap requires custom registers, they’ll be added to the design automatically without the user manually specifying them. The number of privilege levels (from one to three) can also be selected. The whole Exception Unit architecture will be adapted accordingly, alongside the CPU.

When a trap occurs during the execution of a trap handler, it is called a “nested trap.” Each trap has a priority associated with it which impacts the Exception Unit behavior in case of nested traps. If the second trap has a lower priority, it is stacked inside the Exception Unit and will be handled once the current trap handler finishes. If it has a higher priority, it will disrupt the execution of the current trap handler to execute its own. In order not to erase the information stored in the registers to handle the first trap, new registers are created. Since we cannot create an infinite number of registers, there is a limit to the number of nested interrupts that our design can handle. This limit is set by the user as part of our metamodel.

The `mtvec` register has two bitfields: `mtvec.BASE`, where the address of the trap handler is stored, and `mtvec.MODE` that defines special behavior for interrupts. Figure 3.1.9.7-1 shows how hardware and software collaborate to pass control to the correct handler when `mtvec.MODE = 0`. When a trap is triggered, the Exception Unit will write to the `mcause` register and propagate the `mtvec.BASE` value to the PC. The program jumps to a general trap handler that will read the cause of the trap and jump to a specific trap handler accordingly. When `mtvec.MODE = 1`, exceptions still jump to the address contained in `mtvec.BASE` and will ultimately execute a general exception handler that is responsible for jumping to the specific handler. However, interrupts don’t jump to `mtvec.BASE` but to a location depending on their code (cause of the interrupt). At this location, there is directly a jump instruction to the specific handler location without requiring the software to read `mcause`, as shown in Figure 3.1.9.7-2. This lowers the latency for handling interrupts as the core now only needs to perform two jumps (first decided by hardware and second by software) to execute the specific trap handler. Exceptions are also sped up, as the software in the general exception handler needs to check fewer values for `mcause` since the interrupts have been taken away.

In any case, there is still a latency before executing the trap handler, which could be a problem for safety-critical applications where the Worst-Case Execution Time (WCET) is of utmost importance. For this reason, we developed a fast address resolution, as shown in Figure 3.1.9.7-3. Each trap is now associated with a register. When a trap occurs, the PC is set to the address of the associated register to handle the specific trap directly, reducing the latency to a minimum. The user can decide not to create a register for a specific trap. If so, the PC will be set to `mtvec.BASE` per default when this trap is triggered, and the software will be responsible for jumping to the specific trap from there. The user can also select a single register to be associated with multiple traps. This high flexibility makes this Exception Unit suitable for multiple applications with different requirements.

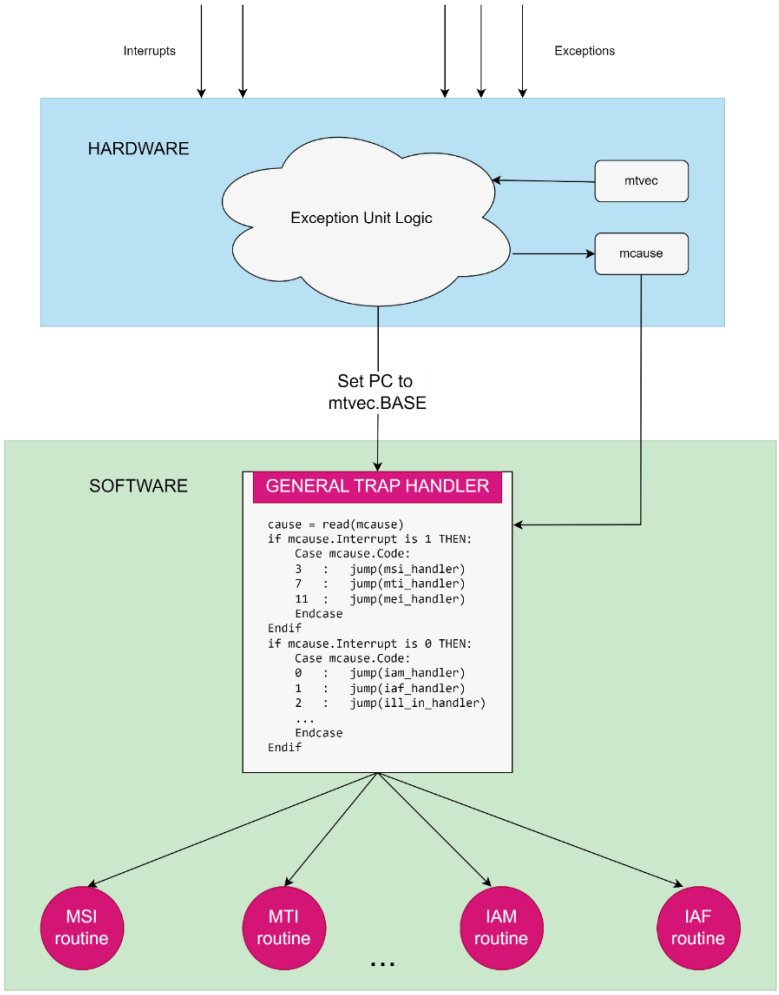


Figure 3.1.9.7-1: Safety Control - Normal address resolution with `mtvec.MODE = 0`

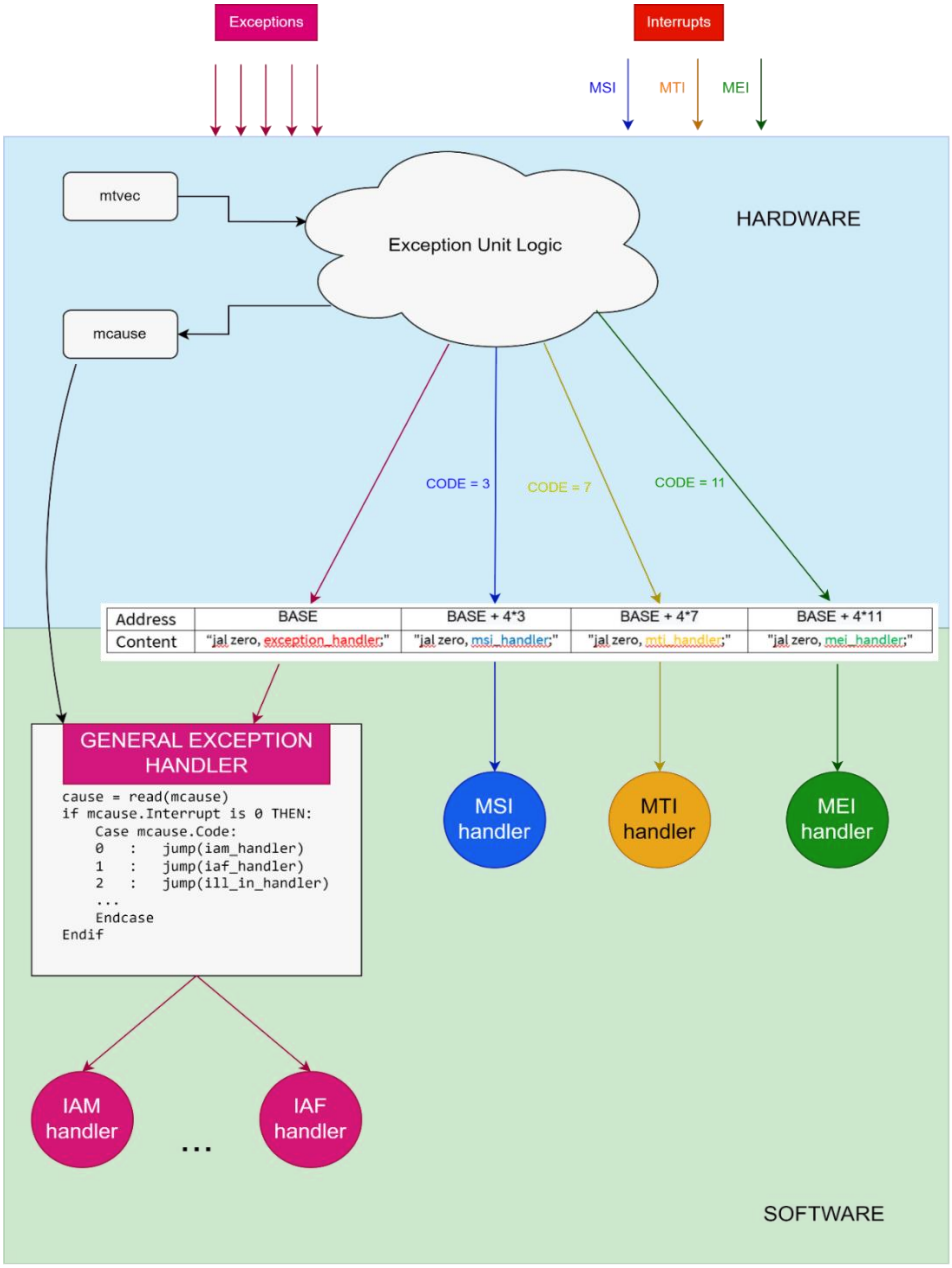


Figure 3.1.9.7-2: Safety Control - Normal address resolution with `mtvec.MODE = 1`

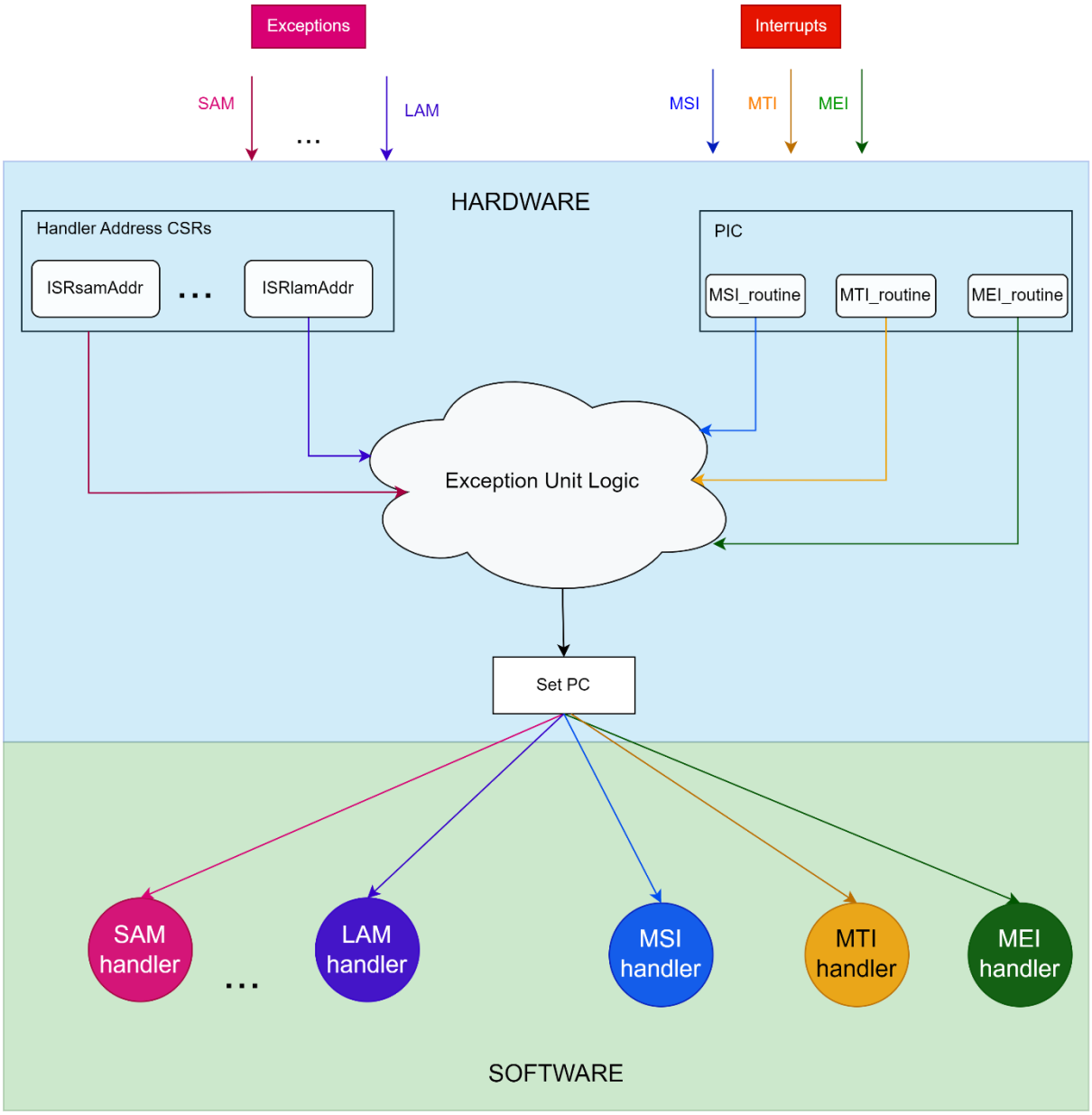


Figure 3.1.9.7-3: Safety Control - Fast address resolution

3.1.10 Safety Island - Interface Definition – UZL

Part of Task 3.1 Safety & Security Modules.

3.1.10.1 General Information

A safety island is in charge of monitoring the Processing System, detecting and managing overall critical behaviour, and provides a function to execute selected high-criticality software. The safety island detects safety issues and either flags the issue or, more advanced, also handles the issue. The safety island is built from several units comprising computing, monitoring, and control.

Externally, the safety island has similar interfaces to those of a processing system, such as those to plug it to address/data interconnects (e.g., an AMBA AXI interface) as well as to the interrupt controller handling interrupts of several units. Hence, the Multi-Processor SoC -- where the safety island is deployed -- remains mostly unchanged, and only minor modifications in some units' interfaces are required to add safety island components.

3.1.10.2 Purpose and Scope

In the last two years, there were developments regarding safety island interfacing inside and outside the ISOLDE projects. For example, BSC's SafeSU Unit uses the AMBA AXI interface to connect to the processing system, also OFFIS' Time Contract Monitoring Co-Processor (TCCP) module references on BSC's strategy. ETHZ proposes a System called Carfield, which again makes use of the AMBA AXI interface and some additional interrupt signals.

During the proposal and starting period of ISOLDE, partners already decided to use the AMBA AXI interface and focus their development on this interface as a connection between the safety island / safety modules and the processing system. Based on this development, UZL will refocus and move efforts into the development of the accelerator system.

3.1.10.3 Place in the System

The safety island consists of different modules that are connected to the processing system as illustrated in Figure 3.1.10.3-1. There is also an independent system of safe cores, which is independent from the main processing system.

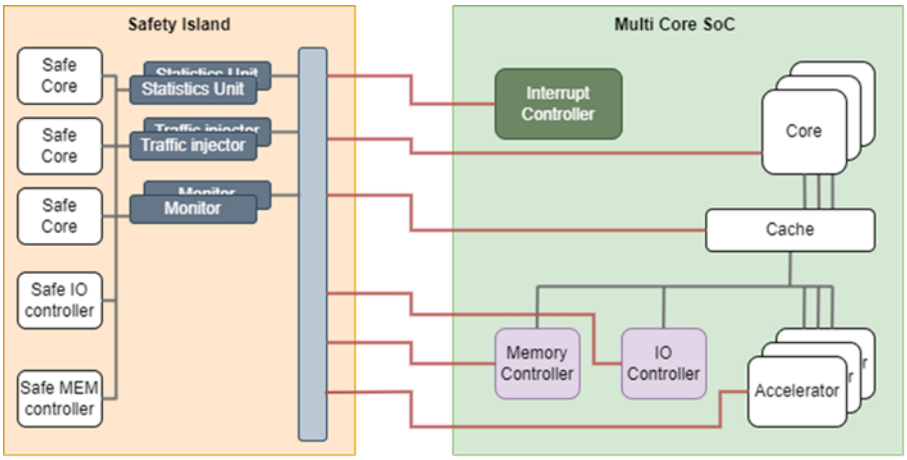


Figure 3.1.10.3-1: Safety Island block design & MPSoC

3.1.10.4 Interfaces

Following the partner concepts, the AMBA AXI interface will be used to connect to the processing system. Furthermore, modules will have certain independent interrupt signals that are handled by the safety islands' interrupt controller.

3.1.10.5 Reset Strategy

The Safety Island is independent from the processing system. Hence, it will be alive during system's reset process. To achieve this behaviour, the reset signal for the processing system must be delayed until the safety island is ready to operate. This can be realized by either the safety island-controlled reset or by the reset logic which will be ANDed with the safety island state "running".

3.1.10.6 Debugging Strategy

When the CVA6 core is in debug-mode, the safety island should be informed analogue, so it is capable of differentiating between debugging mode and misbehavior.

3.1.11 Root-of-Trust Unit (RoT) – UNIBO

Part of Task 3.1 Safety & Security Modules.

3.1.11.1 General Information

Silicon Root-of-Trust (RoT) units represent the state-of-the-art in terms of trusted computing and system integrity, as they establish an isolated silicon region with security features for data and code protection. They protect memory from tampering and include cryptographic acceleration units and physical countermeasures (e.g., voltage/temperature monitors) to detect security threats.

3.1.11.2 Purpose and Scope

The Root-of-Trust provided by UNIBO within ISOLDE is based on lowRISC's OpenTitan, the first open-source RISC-V based RoT design. It includes acceleration units for the Secure Hash Algorithm (SHA) enabling cryptographic hashing (SHA-256 and SHA-3), message authentication (Hash-based Message Authentication Code - HMAC, KECCAK Message Authentication Code - KMAC) and symmetric encryption (Advanced Encryption Standard - AES). UNIBO's RoT is meant to be a ready-to-integrate silicon IP able to act as a RoT.

3.1.11.3 Place in the System

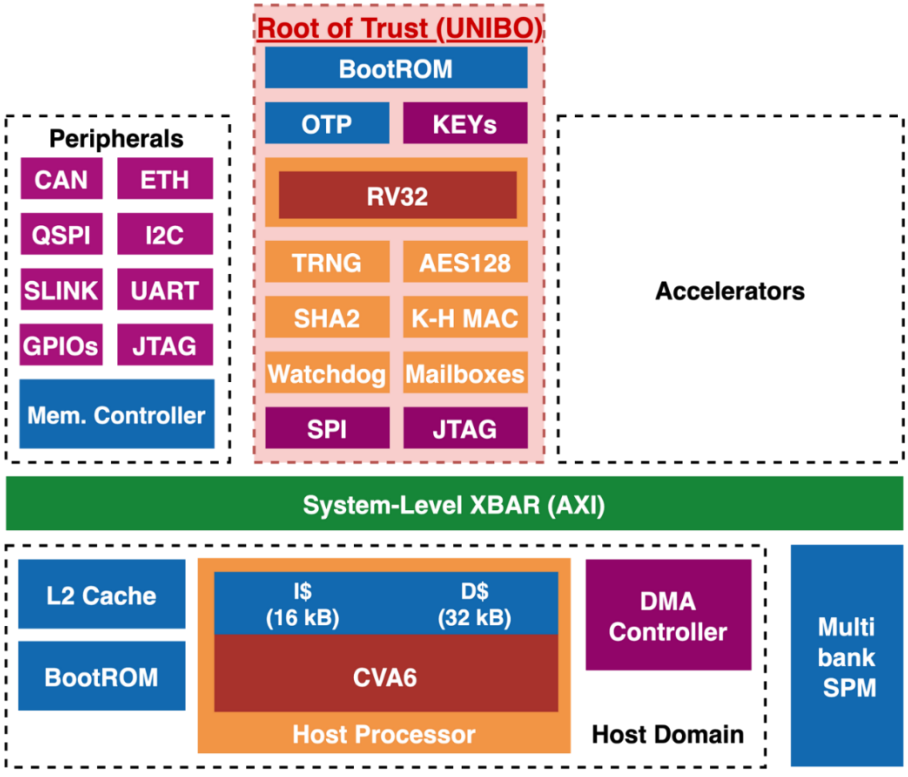


Figure 3.1.11.3-1: RootOfTrust - Place in the system

As shown in Figure 3.1.11.3-1, UNIBO's RoT is meant to be integrated to a top-level system AXI4 crossbar.

3.1.11.4 Block Diagram

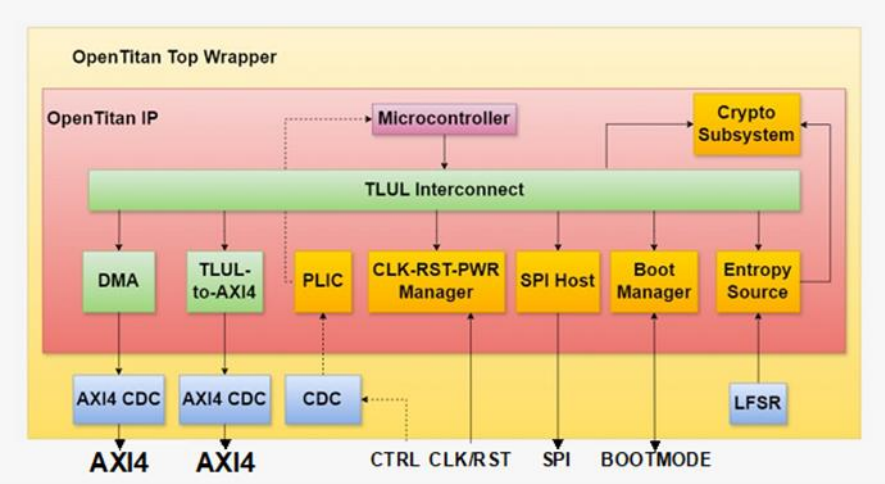


Figure 3.1.11.4-1: RootOfTrust - Block diagram

Figure 3.1.11.4-1 sketches the architecture of the Root-of-Trust (RoT) unit. The central unit of the RoT is OpenTitan[10]; it contains a microcontroller based on the IBEX architecture and is centered on a system interconnect based on the TileLink Uncached Lightweight (TLUL) variant, along with several crypto acceleration subsystem, a boot manager, and a source of entropy. OpenTitan is wrapped into TLUL/AXI4 bridges to provide connectivity with the rest of the system.

3.1.11.5 Power Management Strategy

Dynamic clock gating of the internal computing elements depending on the utilization in the executed kernel.

[10] https://opentitan.org/documentation/index.html

3.1.12 Root-of-Trust Unit Design and Interface with RISC-V Host Processor (TitanCFI) – UNIBO

Part of Task 3.1 Safety & Security Modules.

3.1.12.1 General Information

TitanCFI is a module aimed at extending the CVA6 core with a stage able to filter CFI instructions and forward them into a private mailbox. It foresees the presence of an instance of the OpenTitan Root-of-Trust integrated in the System-on-chip. The computational element in the OpenTitan (namely the IBEX processor) reads the instruction stream and enforces the CFI policy.

3.1.12.2 Purpose and Scope

The proposed module relies on exploiting the OpenTitan RoT, that is already present on the platform to enable Secure Boot and Remote Attestation, as a CFI co-processor. The motivation behind this choice is to harness the RV32IMAC Ibex core within OpenTitan to execute custom CFI policies in software. This approach avoids the area overhead associated with integrating a separate security monitor and maximizes the utilization of the RoT, which typically remains unused after the platform is initially set up. Moreover, our solution takes advantage of the security features provided by the RoT, including access to private tamper-proof storage and cryptography accelerators, to provide additional security guarantees with respect to the other state-of-the-art (SoA) solutions.

3.1.12.3 Place in the System

The module is composed by a host domain and a RoT domain part. The host domain part is located in the CVA6 core and is aimed to extract information about the committed instructions. This host domain part is connected to the RoT domain via a HW mailbox using the System Control and Management Interface (SCMI) protocol. The RoT domain contains a custom OpenTitan firmware executing the CFI policy.

3.1.12.4 Block Diagram

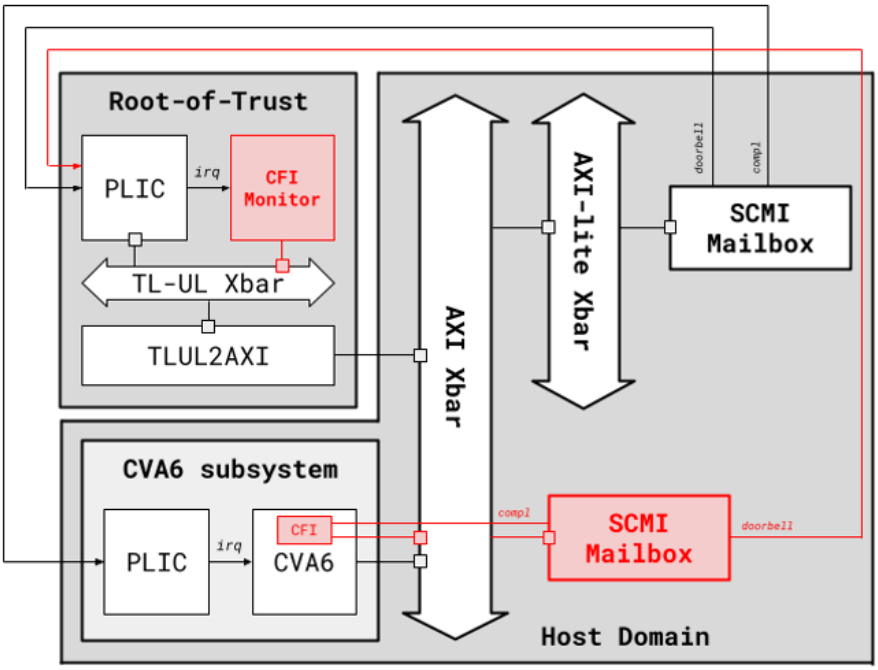


Figure 3.1.12.4-1: TitanCFI - Block diagram

We designed the CFI monitor following the scheme in the Figure 3.1.12.4-1. The monitor is software configurable to extract the wanted instructions. Also, it is possible to configure the instruction monitoring queue. Having a queue of 1 instruction allows an immediate reaction in case of control-flow diversion detection. However, this implies an overhead as the CVA6 core will have to wait for the monitor (i.e., the OpenTitan firmware) to complete the analysis. From the other side, increasing the queue reduces this overhead at the cost of a postponed reaction. The design trade-off of these solutions will be evaluated during the project using a set of benchmarks and considering target aerospace applications.

3.1.12.5 Interfaces

The interface between the CFI monitor inside the CVA6 core and OpenTitan exploits a HW mailbox using the SCMI protocol. OpenTitan is integrated in the SoC and it can access the memory through a custom bridge between the internal TileLink interconnect and an AXI plug exposed externally. Communications between the host domain and the RoT are mediated by a SCMI compliant mailbox. The mailbox consists of a set of general-purpose memory mapped registers meant for data sharing. Additionally, it features two registers, named Doorbell and Completion, which are meant to send an interrupt to the IBEX security microcontroller and to the CVA6 host core. CFI metadata extracted from the retired instructions are stored in a CFI Mailbox, where they can be read from the RoT. The design of the CFI Mailbox is analogous to the SCMI- like mailbox already present in the reference SoC. We parametrize the general-purpose registers to be large enough to store the CFI metadata required to represent a single control flow instruction. When a new metadata is ready to be read, the enhanced CVA6 commit stage sets the doorbell register in order to trigger an interrupt in the RoT. Unlike a regular SCMI-like mailbox, the completion register is not connected to the host domain interrupt controller, but directly to the commit stage of the CVA6 core. To indicate that a previously retired instruction has been checked, the result of the CFI enforcement policy can be read from the mailbox, signaling that the RoT is ready to read the next commit log.

3.1.12.6 Clocking Strategy

The CVA6 core extensions which take care of filtering the instructions retired by the core, forwarding them to the OpenTitan Root-of-Trust, are completely synchronous with the CVA6 core, and they are not expected to need special treatments with respect to the core pipeline.

At the same time, the CFI Mailbox, which stores the CFI metadata until OpenTitan reads it, is synchronous with the interconnect where it is mapped.

3.1.12.7 Reset Strategy

The sequential elements present in the CVA6 core extensions, and the CFI mailbox are expected to be synchronously cleared during reset.

3.1.12.8 Debugging Strategy

When the CVA6 core is in debug mode, none of the instructions retired by the core should be considered for the sake of Control-Flow Integrity enforcement, and no additional instructions should be logged into the CFI mailbox.

3.1.13 High-Performance Cache Analysis – SYSGO

Part of Task 3.1 Safety & Security Modules.

3.1.13.1 General Information

We analyze the high-performance cache provided by CEA in the TRISTAN project for the ISOLDE demonstrator. The output is a short analysis comparing the CEA cache with the default cache.

3.1.13.2 Purpose and Scope

Advance CVA6 ecosystem by showing usability of advanced caches. Our general focus is on safety and security, and caching is one important part of this, e.g., concerning future implementation of cache partitioning.

3.1.13.3 Place in the System

Caches are between CPU and memory and serve to speed up memory access. We expect this to be relevant in an application setting.

3.1.13.4 Block Diagram

The block diagram of the cache itself has been published by TRISTAN partner CEA, see Figure 3.1.13.4-1. We use a demonstrator setup where the application runs on the CPU, that takes data from memory via the cache, see Figure 3.1.13.4-2.

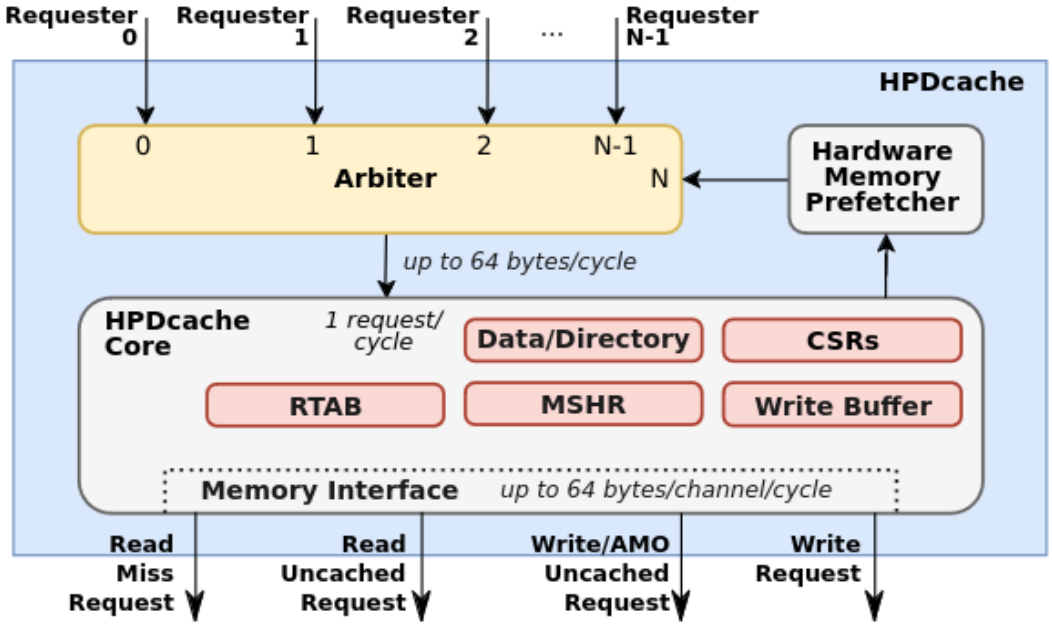


Figure 3.1.13.4-1: High-performance cache block diagram, by CEA [Fuguet 2023]

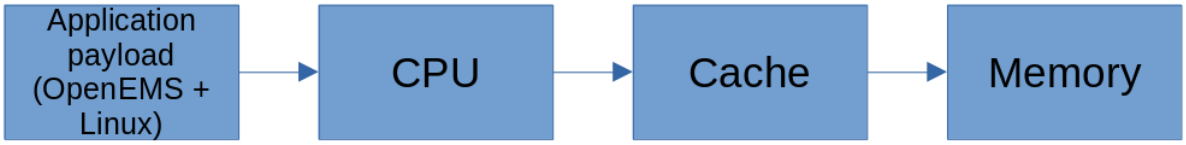


Figure 3.1.13.4-2: High-performance cache analysis block diagram

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

Task 3.2, M3-M33, Leader: UZL

Task 3.2 focuses on accelerator infrastructure such as arithmetic units and fast memories. The infrastructure to the core plays an important part in enabling an accelerator's performance. The developed modules include scratchpad memory (integrated in the Vector-SIMD accelerator developed in Task 3.4), a floating-point accelerator, as well as floating-point tensor processing units with custom float- and fixed-point operators. These accelerators are suitable for intense floating-point applications targeting specific trade-offs between accuracy and performance requirements.

IP	Lead Beneficiary	Type	Domain	Dependencies	Licensing
FPMIX	POLIMI	RISC-V Core Extension	Arithmetic Unit	RISC-V core with F extension	Open source (bfloat16) Proprietary (others)
FPU	UZL	RISC-V Core Extension	Arithmetic Unit	CVA6	Open source
Scratchpad	IMT	Core	Accelerator	None	Restrictive open source (GPL-3.0)

Table 3.2-1: Overview of contributions in Task 3.2

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

Part of Task 3.2 Accelerator infrastructure, memories, arithmetic units, interfaces and virtualization.

3.2.1.1 General Information

Customizable floating-point arithmetic unit that can implement operations with various amounts of precision bits in their floating-point arithmetic formats for the operands and result.

3.2.1.2 Purpose and Scope

The floating-point unit (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. For example, workloads such as artificial-intelligence ones in which a loss of accuracy can be accepted, implementing Floating-Point (FP) formats with smaller precisions can be leveraged to reduce the area occupation and power consumption of computing hardware. To this end, the formats of FP operations in FPMIX can be configured at design time, also by leveraging precision tuning approaches such as [Cherubin2020].

3.2.1.3 Place in the System

The FPMIX floating-point unit can be integrated as the functional unit of a RISC-V CPU core replacing any existing FPU.

FPMIX will be evaluated for the integration in the root of trust of the space demonstrator use case, if floating-point computations are needed in the root of trust.

3.2.1.4 Block Diagram

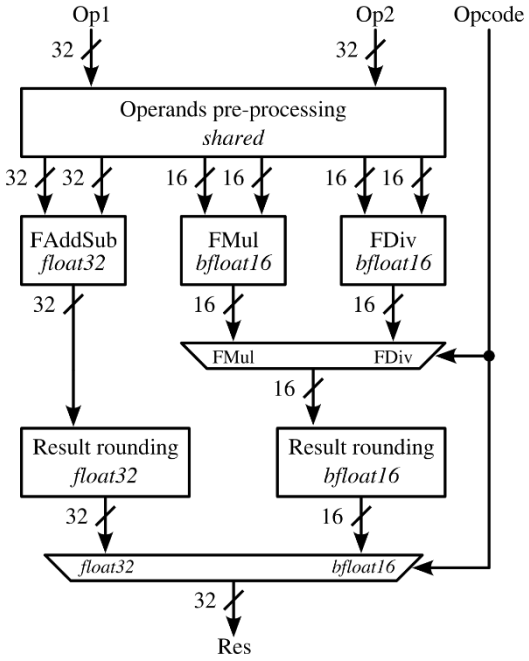


Figure 3.2.1.4-1: FPMIX - Block diagram

The FPU implements floating-point operations whose precision (number of mantissa bits) can be configured at design time. The precision for each type of operations 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.

For example, the block diagram depicted in Figure 3.2.1.4-1 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 employed by at least an operation in the FPU.

Instantiating FP operations 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.

3.2.1.5 ISA

The FPU implements arithmetic operations corresponding to those defined for the single-precision (float32) format by the IEEE-754 standard, and it is therefore compatible with the instructions defined by the F single-precision floating-point extension of the RISC-V ISA.

All the floating-point formats supported by the FPU share indeed the dynamic range of IEEE-754 float32 and have at most the number of mantissa bits of float32.

As mentioned in Section 3.2.1.4, internal conversions to and from smaller-precision formats are performed by truncation and extension, respectively, thus no additional conversion instructions are required.

3.2.2 Floating-Point Unit for RISC-V (FPU) – UZL

Part of Task 3.2 Accelerator infrastructure, memories, arithmetic units, interfaces and virtualization.

3.2.2.1 General Information

Floating-Point Units are specialized arithmetic units for calculating floating-point arithmetic. In modern systems, they are highly integrated into the processor pipeline and support different arithmetic specifications such as IEEE 754 single precision (32bit) 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.

3.2.2.2 Purpose and Scope

The RISC-V ecosystem is highly adaptable and configurable. It is hence desirable that the RISC-V's Floating-Point Unit is configurable and adaptable for different use cases as well. During the current runtime of the ISOLDE project, UZL researched and tested some of the already existing and Open-Source Floating-Point Units of the RISC-V ecosystem. One (promising) example is the OpenHW Group's Floating-Point Unit CVFPU which is capable of IEEE 754-2008 single-, double-, quad- and half-precision specification. With the development of a FPU with support for a wide range of floating-point arithmetic and compatible with different processor systems, UZL intends to focus on integrating selected and domain specific FPUs into the SoC system, like, for example, for the automotive demonstrator (owned by Continental).

3.2.2.3 Place in the System

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's block diagram (Figure 3.2.2.3-1), the FPU (depicted inside the red box) is integrated into the processor pipeline with direct access to required operands.

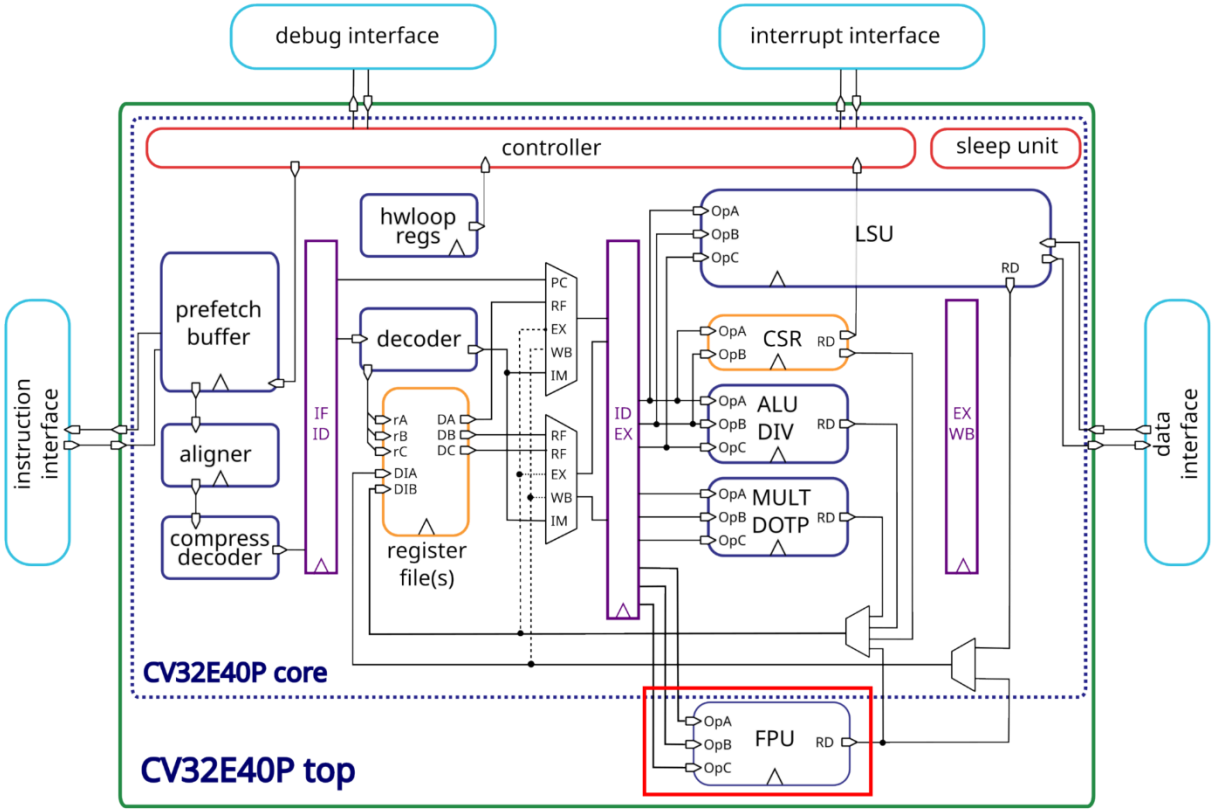


Figure 3.2.2.3-1: Official CV32E40P block diagram including FPU (red box)

3.2.2.4 Block Diagram

The FPU core is interfaced to the RISC-V core. It is integrated to the pipeline of the processor and has access to the operands. The block diagram is depicted in Section 3.2.2.3.

3.2.2.5 ISA

No ISA modifications are required. Floating Point is already part of the ISA Specification: “F” (Single Precision), “D” (Double Precision), and “Q” (Quad Precision).

3.2.2.6 Interfaces

The FPU is interfaced into the pipeline with direct access to the Register File for operand access and write stage.

3.2.3 Scratchpad – IMT

Part of Task 3.2 Accelerator infrastructure, memories, arithmetic units, interfaces and virtualization.

3.2.3.1 General Information

In memory bound applications, memory accesses may represent an important bottleneck. The memory receives requests from many components and only can handle them sequentially. For a vector processor, sequential data access may have performance penalties. One possible solution is to employ multi-bank memories which employ two techniques: data duplication and data mapping.

Assuming each memory bank allows one memory access per clock cycle, conflict-free parallel access means that by using N memory modules we can access N distinct data items in each clock cycle, one from each memory module. If two or more data items are required from the same memory bank, this creates a conflict, and the accesses will be serialized.

The data duplication technique creates copies of the data stored in a memory module in two or more memory modules. That allows parallel access to these memory banks, allowing conflict-free parallel access to data in a simple and straight forward way. The important disadvantage is the hardware cost of duplicating the memory modules while not increasing the capacity we need to duplicate the memory module. Another possible disadvantage is related to coherency: writing data may lead to data coherency problems if synchronization is not handled correctly.

The second technique is data mapping: distributing the data in multiple memory, so it can be accessed in parallel. When using data mapping, no additional memory modules are needed. However, the challenge is to optimally distribute data to these memory banks to allow conflict-free access.

Our proposed Scratchpad Memory is based on PolyMem [Ciobanu2018] and uses the Memory Access Schemes originally used in the Polymorphic Register File [Ciobanu2013], [Kuzmanov2006].

3.2.3.2 Purpose and Scope

A Scratchpad Memory is a personalized mapped memory that guarantees parallel conflict-free access for a limited selection of access patterns which are known at design time. It may improve performance for memory bound applications for which the memory access patterns are supported by the Scratchpad Memory.

Our proposed Scratchpad Memory acts as a fast memory module that allows multi-lane, conflict-free access to multiple data simultaneously for selected access patterns. Our design supports the following access patterns: rectangle, row, column, transposed rectangle and main and second diagonal. Depending on the memory access scheme implemented, our Scratchpad Memory allows conflict-free access to one or more access patterns.

This Scratchpad Memory is envisioned to have a configurable capacity (e.g., a few MB), and the data transfers to/from the Scratchpad Memory are managed manually by the programmer.

3.2.3.3 Place in the System

The Scratchpad Memory could be placed in two places in the system: inside a system component or as a standalone component. If inside the component, the Scratchpad give to user a simple and basic interface for control and data. As a standalone component, the system has an AXI Lite interface for configuration, an AXI Memory-Mapped (AXI-MM) interface to access main memory and multiple AXI Stream interface to send and get data to another component.

Scratchpad Memory in a component

One use case scenario for Scratchpad is to be used inside an accelerator. The designers could use this Scratchpad Memory as a buffer, to store some data temporally to finish the job faster while reducing the communication with the slower main memory. In that scenario, all the control signals are driven by the accelerator and the Scratchpad handles multiple data items per request. An intuitive diagram is presented in Figure 3.2.3.3-1.

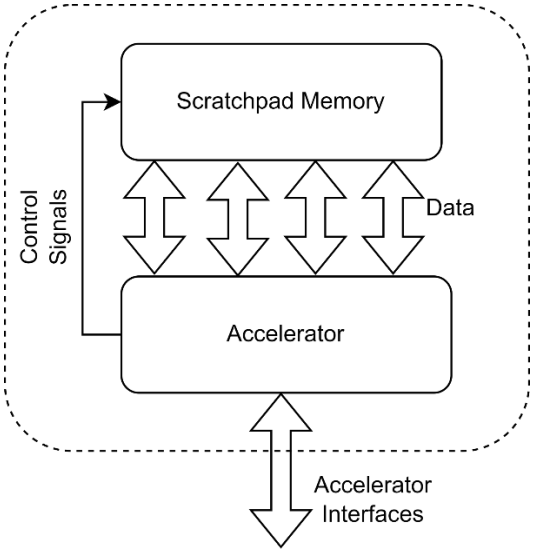


Figure 3.2.3.3-1: Connecting a Scratchpad Memory to an accelerator.

The default Scratchpad interfaces include 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. This multi-port feature makes the Scratchpad Memory suitable for an arithmetic accelerator. The full interfaces are presented in Figure 3.2.3.3-2.

The simple control signals are 2D coordinates for writing and reading, memory organization 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 elements is called the number of lanes and is a parameter set at design time.

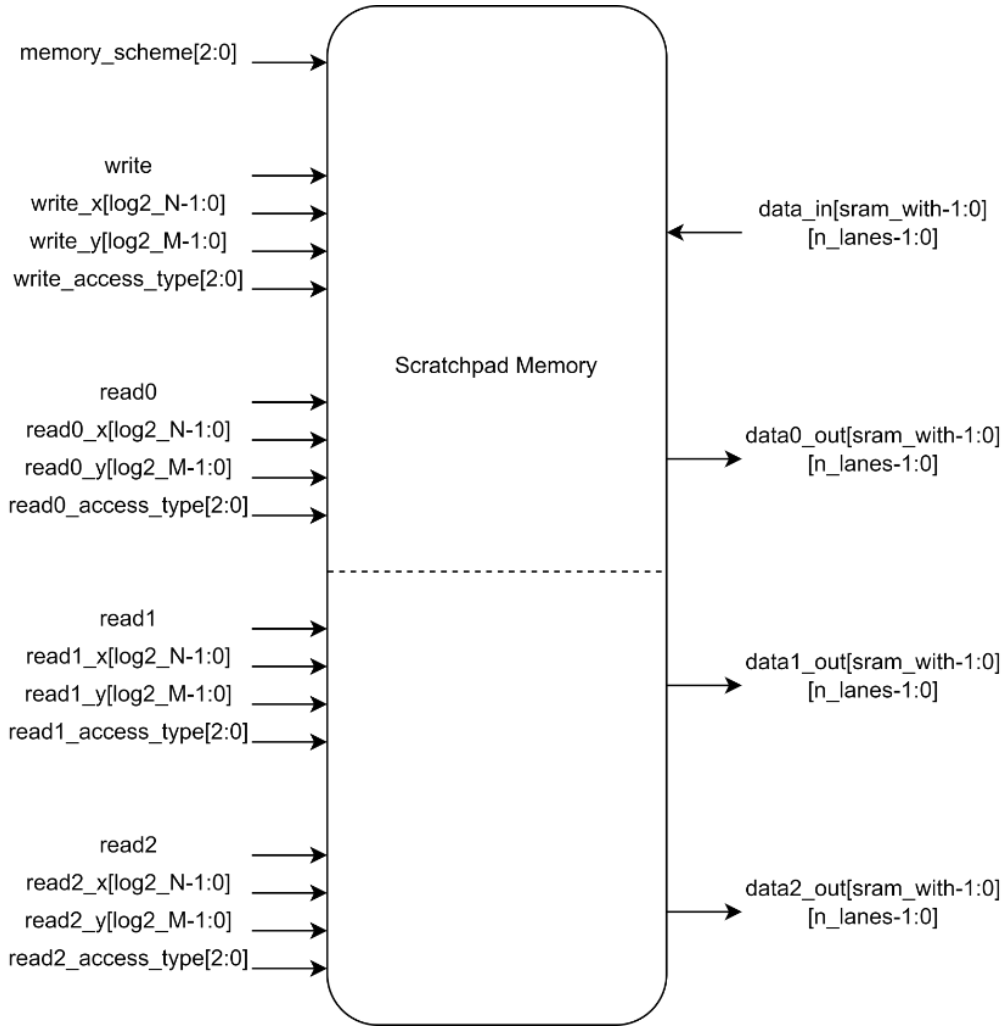


Figure 3.2.3.3-2: Interfaces for Scratchpad Memory [Ciobanu2013]. The upper side is the basic interface, and the lower one is the interfaces extended by duplication.

Scratchpad Memory as a component

Another use case scenario is to use the Scratchpad as an independent component. In that way, the user has a wrapper over read/write operations and may unload the accelerator from some memory related tasks. The Scratchpad Memory components handle memory access and data synchronization.

The Scratchpad Memory as a standalone component is presented in Figure 3.2.3.3-3. When the Scratchpad is a standalone component, the control signals are driven by a configuration register bank. This component is configured by the system via an AXI Lite interface. This configuration includes a memory interface that is connected to the main AXI interconnect to access the main memory. Furthermore, to allow high-speed communications with other components, the interfaces are AXI Stream. In that configuration the Scratchpad is configured from the outside.

With a dedicated configuration interface from the AXI family, both the CPU and the accelerator can configure the Scratchpad, allowing the user to preload data in the Scratchpad Memory and potentially to improve performance.

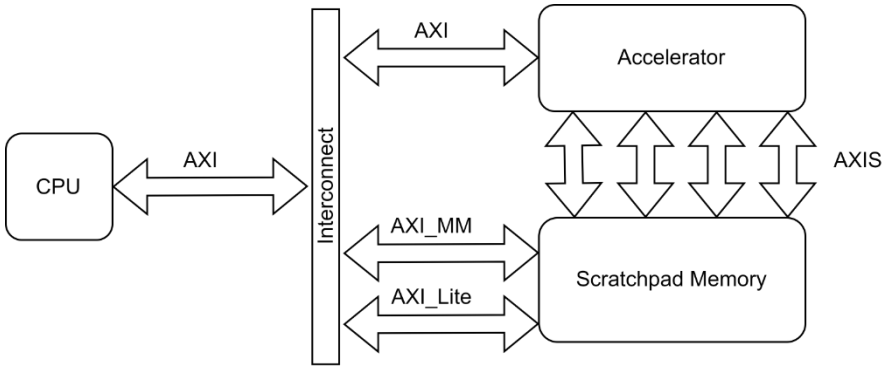


Figure 3.2.3.3-3: Scratchpad Memory as a component

3.2.3.4 Block Diagram

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 *[Ciobanu2013]* are Rectangle Only (ReO) [Kuzmanov2006], Rectangle Row (ReRo), Rectangle Column (ReCo), Row Column (RoCo) and Rectangle Transposed (ReTr). The Rectangle Only scheme only 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 internal structure of the Scratchpad Memory is presented in Figure 3.2.3.4-1. 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 module. The data is read from the memory banks and finally a Data Shuffle rearranges the data to be passed to the user.

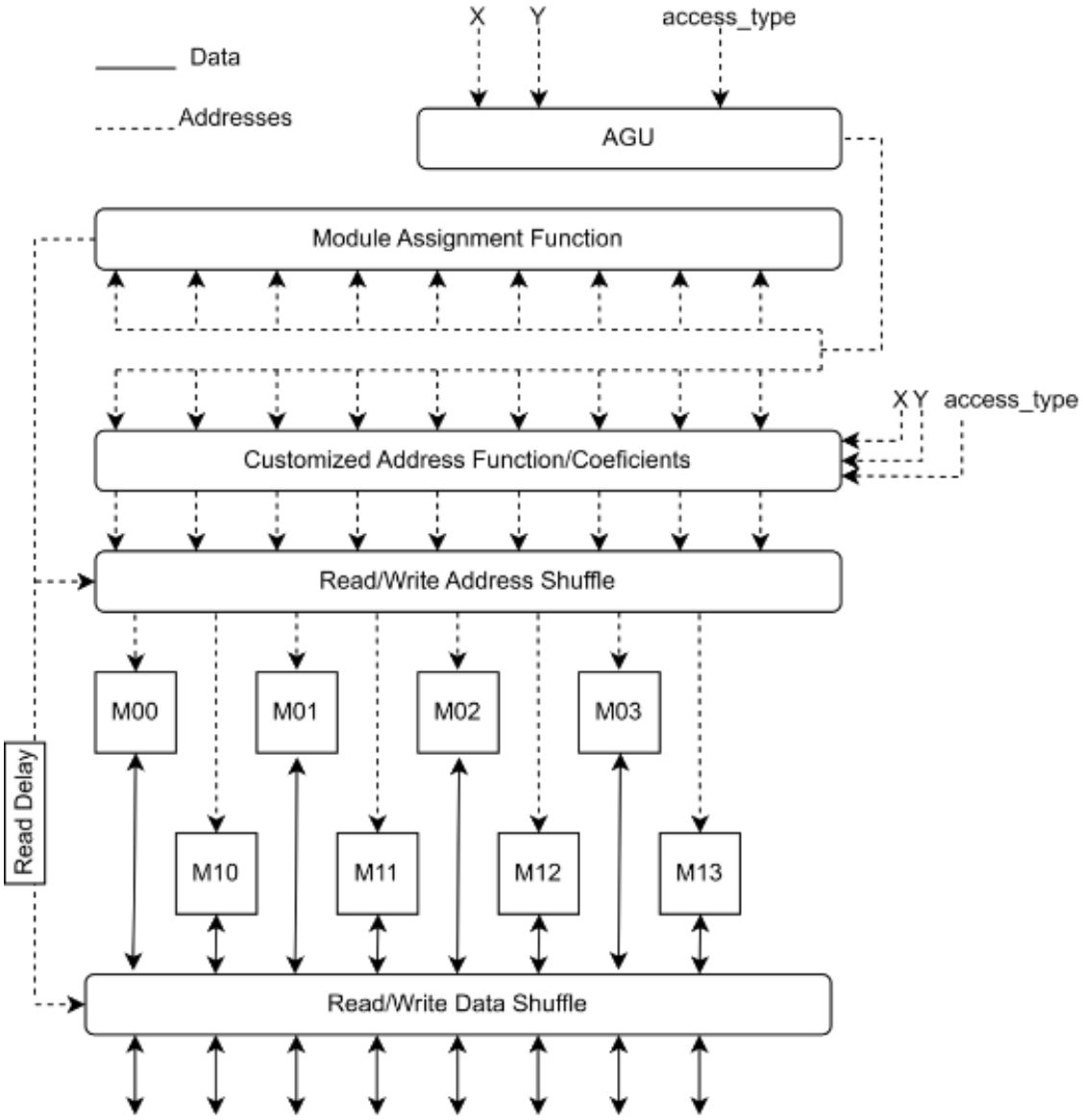


Figure 3.2.3.4-1: Internal organization of Scratchpad Memory, based on [Ciobanu2013]

Figure *3.2.3.4*-2 illustrates the scenario when the Scratchpad is employed as a standalone component, then additional modules are required. This standalone Scratchpad has a configuration unit, allowing for the Scratchpad Memory to be configured. A DMA engine handles communication with the main memory,

handling data reads and writes to/from the main memory. First-In-First-Out (FIFO) units are used to handle fast streams of data to other components.

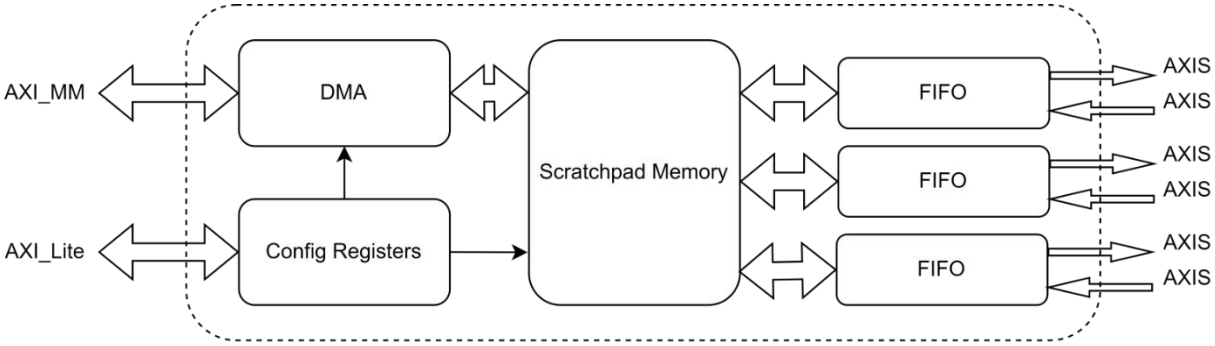


Figure 3.2.3.4-2: Internal architecture for Scratchpad as a component

3.2.3.5 Clocking Strategy

The Scratchpad may be employed in two modes: inside an accelerator or as a standalone component. If placed inside an accelerator, the Scratchpad requires one clock domain for memory banks. As a standalone component, the Scratchpad module has two clock domains: one clock domain for the configuration and memory interface and the second one for the AXI Stream (AXIS) interfaces. The synchronization between the memory clock domain and the stream interfaces is handled by employing asynchronous FIFOs.

3.2.3.6 Reset Strategy

When the Scratchpad Memory is integrated inside an accelerator it may require a reset signal for the memory banks. This depends on the target technology, as some Block Rams (BRAMs) have a reset signal that may be used to clear the memory data.

When the Scratchpad Memory is used as a standalone component and the reset signal is active, then all the states machines are returned to initial state, all configuration registers return to default values and all FIFOs are flushed.

3.3 Monitoring Infrastructure

Task 3.3 M3-M33, Task Leader: POLIMI

Task 3.3 focuses on the development of components and methodologies that provide monitoring support for multiple purposes, ranging from performance monitoring in a safety-specific context to power and energy monitoring, via on-line hardware-software monitoring infrastructures that enable therefore the optimization of the overall system both at design time and at run time.

On the one hand, a multicore statistics unit (BSC) is integrated as part of the safety island, while context-aware performance monitoring counters are extended with context filtering capabilities to further strengthen the monitoring of the safety island (TRT) and a configurable and programmable co-processor dedicated to monitoring time contracts (OFFIS) can observe application-specific hardware and software timing properties. On the other hand, a dedicated methodology can deliver an on-line power monitoring infrastructure (POLIMI) while considering the accuracy, area overhead, and side-channel information leakage metrics as constraints in the power model identification phase.

IP	Lead Beneficiary	Type	Dependencies	Licensing
CA-PMC-IF	TRT	Core	CA-PMC, CA-BUS (WP2)	TBD
RTPM	POLIMI	Core	Monitored IP	Proprietary
SafeSU	BSC	Core	None	Permissive open source (MIT)
TCCP	OFFIS	Core	Safety Island, SafeSU	Permissive open source (Apache-2.0)

Table 3.3-1: Overview of contributions in Task 3.3

3.3.1 Context-Aware PMC Interface (CA-PMC-IF) – TRT

Part of Task 3.3 Monitoring infrastructure.

3.3.1.1 General Information

The Context Aware Monitoring framework is a set of IPs to enhance the monitoring IPs with context information and standardize the same monitoring IPs deployed in an SoC. The context information is typically defined by a context controller which typically is a core defining the context in which the events monitored are issued. The Context Aware Monitoring framework is composed of 4 different IPs (or IP extensions): the CA-CORE, the CA-BUS, the CA-PMC (Section 3.1.3), and the CA-PMC-IF.

3.3.1.2 Purpose and Scope

The CA-PMC-IF module's purpose is to provide a means for the system (e.g., main core, supervision core) to program the CA-PMC module the CA-PMC-IF is associated with and retrieve the counters from the CA-PMC.

In the context of ISOLDE, we will target the SoC caches as IP integrating the CA-PMC (and CA-PMC-IF to configure the CA-PMC and retrieve data from the CA-PMC).

3.3.1.3 Place in the System

The CA-PMCs are extended Performance Monitoring Counters to be placed in the different IPs of the system, as shown in the example in Figure 3.3.1.3-1. The CA-PMC-IF module is the module responsible for making the CA-PMC visible to the rest of the system.

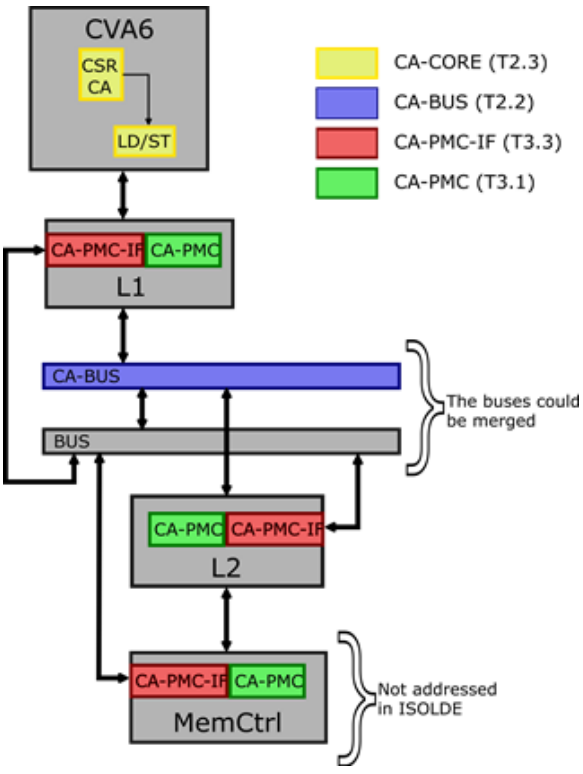


Figure 3.3.1.3-1: CA-PMC-IF - Place in the system

In the project's context, the Instruction and Data L1 Caches CA-PMC-IFs will be targeted, but it should be reusable in other IPs.

3.3.1.4 Block Diagram

The CA-PMC-IF definition is currently being actively discussed and a final architecture is not yet available. Figure 3.3.1.4-1 (taken from the CA-PMC description) shows a high-level block diagram displaying how the CA-PMC-IF is connected to the CA-PMC and an AXI bus.

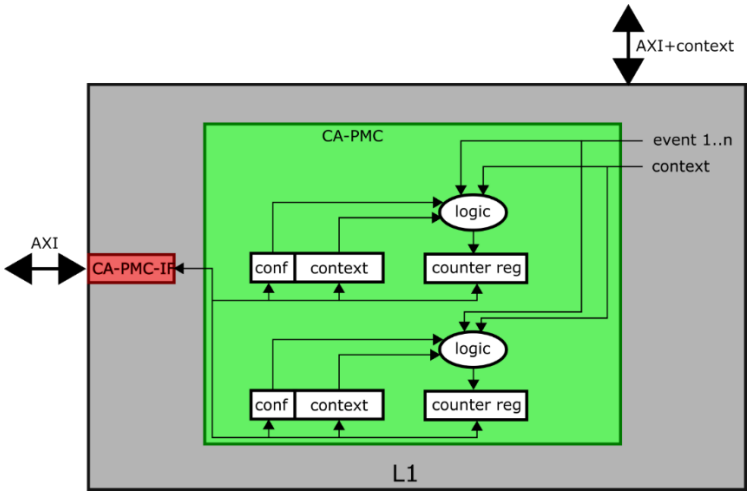


Figure 3.3.1.4-1: CA-PMC-IF - Block diagram

3.3.1.5 Interfaces

The CA-PMC-IF is connected with two different IPs:

- The CA-PMC module, and
- to an AXI bus to which it provides a memory mapped interface.

AXI bus interface

The CA-PMC-IF provides an AXI bus interface, providing a memory mapped access to the CA-PMC registers the CA-PMC-IF is connected with. Details on the exposed memory map are being defined.

Dedicated interface between CA-PMC-IF and CA-PMC

Register write/read interface controlled by the CA-PMC-IF is required between the CA-PMC-IF and the CA-PMC. It provides the CA-PMC-IF the capability to read and write the CA-PMC configuration and counter registers.

3.3.2 Run-Time Power Monitoring Instrumentation (RTPM) – POLIMI

Part of Task 3.3 Monitoring infrastructure.

3.3.2.1 General Information

The automatic generation of the run-time power monitoring infrastructure delivers periodic power estimates from the switching activity of a few select signals in the monitored components.

3.3.2.2 Purpose and Scope

The effectiveness of run-time optimization techniques that aim to improve the energy efficiency of a target computing platform is strongly tied to the quality of the measurements or estimates of power consumption provided by a run-time power monitoring infrastructure. The latter can perform indirect estimation of the dynamic power consumption of the target computing platform by analyzing its run-time statistics such as the switching activity of microarchitectural signals, monitored through dedicated hardware counters.

3.3.2.3 Place in the System

The monitoring infrastructure consists of a set of switching activity counters attached to corresponding inputs and output signals of the accelerators of which it is required to monitor dynamic power consumption. The power estimate value obtained by aggregating the counter values is exposed through a hardware register.

3.3.2.4 Block Diagram

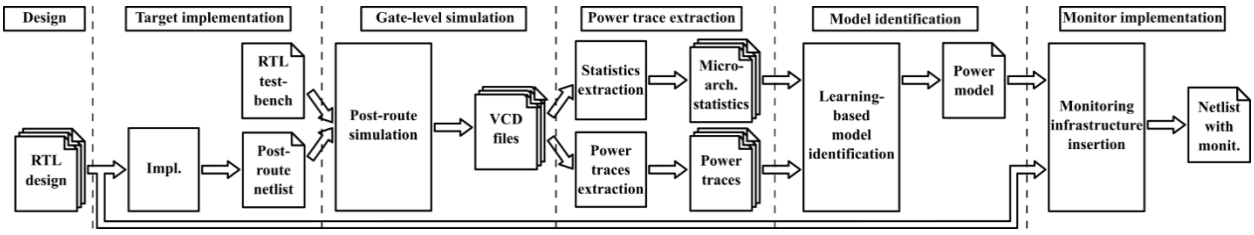


Figure 3.3.2.4-1: RTPM - Block diagram

A run-time power monitoring infrastructure is inserted in a generic RTL design as shown in Figure 3.3.2.4-1. First, the design's internal switching activity is correlated to its power consumption. This is done in order to identify a first-order linear power model.

A selected subset of signals whose switching activity is selected as the input to such identified model is wrapped with hardware counters that monitor their switching activity. The collected values are periodically gathered and used to compute an estimate of the power consumption, which is exposed externally to be usable by a run-time management framework.

The accuracy, the area overhead, and the side-channel information leakage of the monitoring infrastructure can notably be considered during the model identification phase. This allows to provide not only accurate estimates but also to satisfy the area and security requirements and constraints of the overall system.

3.3.3 Safety-Related Statistics Unit (SafeSU) – BSC

Part of Task 3.3 Monitoring infrastructure.

3.3.3.1 General Information

The SafeSU is a modular and scalable Performance Monitor Unit (PMU) that can be connected to any on-chip interconnect and allows multicore interference observability and controllability.

3.3.3.2 Purpose and Scope

The SafeSU builds on a number of components, namely, the Contention-Cycle Stack (CCS), the Request Duration Counter (RDC) and the Maximum-Contention Control Unit (MCCU).

- The CCS offers observability features by providing multicore time-interference breakdown.
- The RDC provides end users with an observability channel to monitor high-watermark latencies per event and core, as needed for interference bounding (e.g., during worst-case execution time estimation).
- The MCCU offers controllability capabilities with interference quota monitoring and enforcement, alerting the user when allocated quotas are exceeded.

3.3.3.3 Place in the System

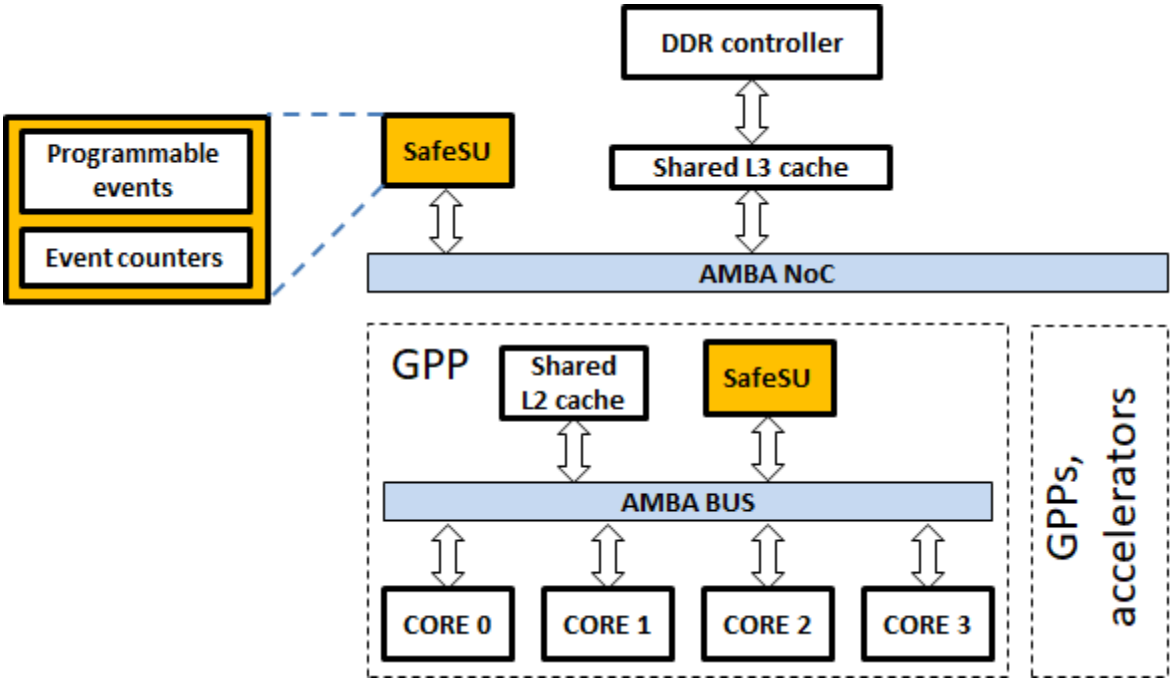


Figure 3.3.3.3-1: SafeSU - Place in the system

The monitoring interface of the SafeSU depicted in Figure 3.3.3.3-1, is AMBA AHB and AXI compliant. The SafeSU is intended to be connected to those types of interfaces, and it is particularly useful if those interfaces have either multiple managers or are connected to subordinates receiving requests from multiple managers. For instance, its best location is normally connected to the interface used by the cores and/or accelerators to access shared caches or memory controllers so that ongoing traffic can be monitored, and

eventually compared to predefined quotas to ensure that no manager abuses the use of relevant shared resources.

SafeSU's programming port is compliant with AMBA APB, although it will be extended to AMBA AXI in the future.

3.3.3.4 Block Diagram

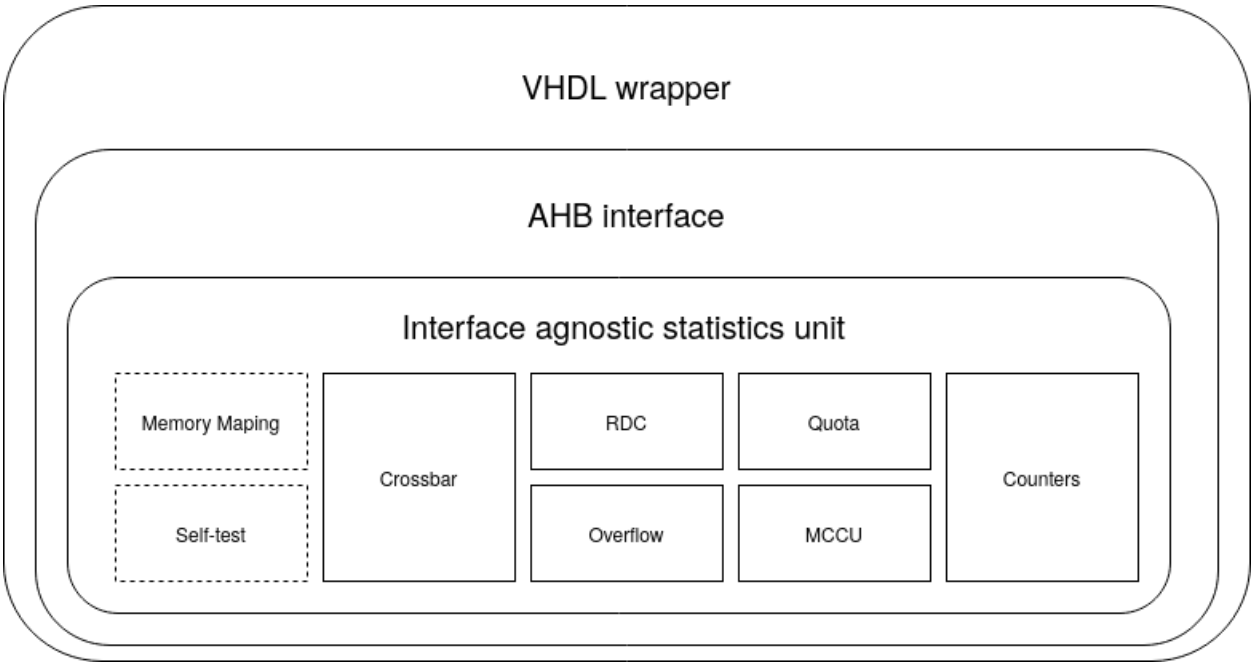


Figure 3.3.3.4-1: SafeSU - Block diagram

The main components of the SafeSU are the following:

- Self-test: configures the counters' inputs to a fixed value bypassing the crossbar and ignoring the SoC inputs. This mode allows for tests of the software and the unit under known conditions.
- Crossbar: routes any input event to any counter.
- Counters: A group of simple counters with settable initial values and general control register.
- Overflow: Detects counters' overflow. It can raise interrupts upon overflow with its dedicated interruption vector and per counter interrupt enable.
- Quota: Deprecated as replaced by MCCU (it may be excluded in a future release).
- MCCU (Maximum Contention Control Unit): Contention control measures for each core for the particular event type that has been programmed to be monitored. It can raise an interrupt if a contention threshold is exceeded. It accepts real contention signals or estimation through weights.
- RDC (Request Duration Counters): Provides measures of the pulse length of a given input signal (watermark). It can be used to determine maximum latency and cycles of uninterrupted contentions. Each of the counters can trigger an interrupt at a user-defined threshold.

3.3.3.5 Interfaces

AMBA AHB/AXI interface

The AHB or AXI interface is a subordinate interface used to snoop traffic. It is fully compliant with the specification of the corresponding protocol. Note that, in general, a SafeSU instance supports only one of those interfaces.

AMBA APB interface

The AMBA APB subordinate interface is used to program the control registers of the SafeSU. The control registers are as follows:

Main configuration and self-test

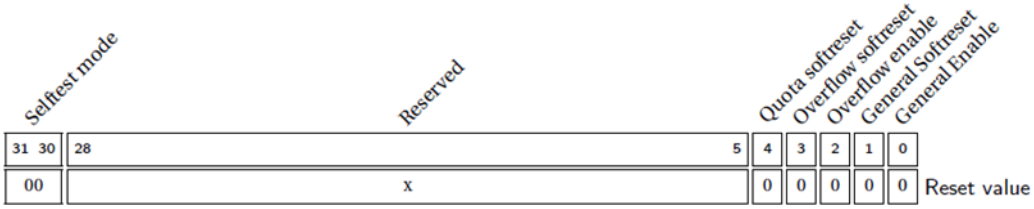


Figure 3.3.3.5-1: SafeSU - Base Configuration Register (0x000)

Reset and enable of overflow, quota, and regular counters' operations can be performed with the Base Configuration Register shown in Figure 3.3.3.5-1. All signals are active high.

Self-test mode allows bypassing the input events from the crossbar and instead using a specific input pattern where signals are constant. This mode can be used for debugging. After the addition of the crossbar and debug inputs, there is a certain overlap. The same results can be achieved with the correct crossbar configuration. Nevertheless, it has been included in this release for compatibility.

These are the self-test modes for each configuration value of the field `Selftest mode` part of the register shown in Figure 3.3.3.5-1:

- 0b00: Events depend on the crossbar. Self-test is disabled.
- 0b01: All signals are set to 1.
- 0b10: All signals are set to 0.
- 0b11: Signal 0 is set to 1. The remaining signals are set to 0.

Crossbar

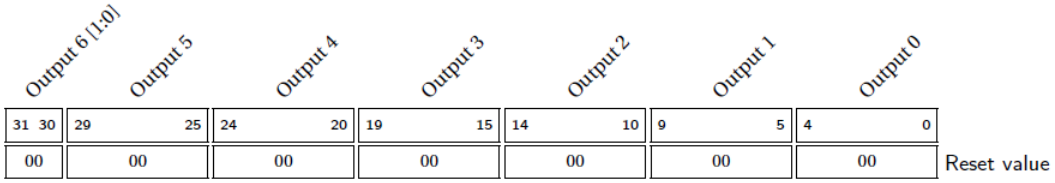


Figure 3.3.3.5-2: SafeSU - Crossbar Configuration Register 0 (0x0AC)

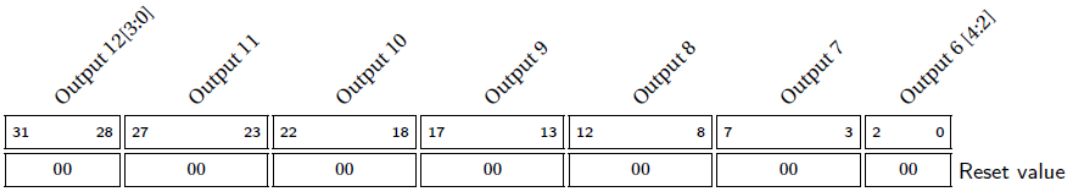


Figure 3.3.3.5-3: SafeSU - Crossbar Configuration Register 1 (0x0B0)

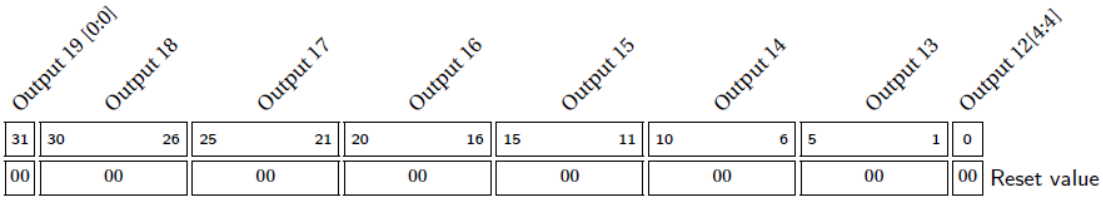


Figure 3.3.3.5-4: SafeSU - Crossbar Configuration Register 2 (0x0B4)

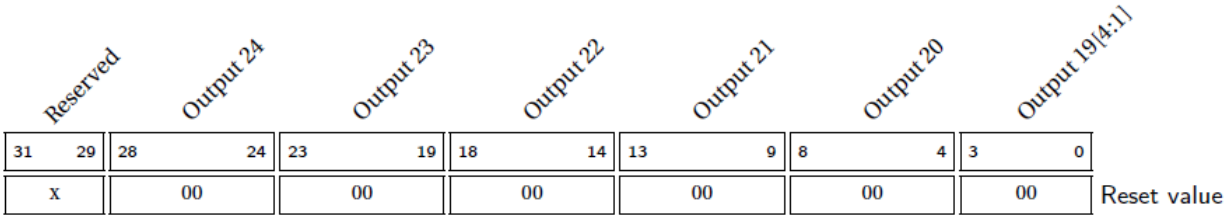


Figure 3.3.3.5-5: SafeSU - Crossbar Configuration Register 3 (0x0B8)

This feature allows routing any of the input signals of the SafeSU into any of the 24 counters of the SafeSU (see Table 3.3.3.5-1). Each one of the counters has a 5-bit configuration value. These values are stored in the registers shown in Figures 3.3.3.5-2, 3.3.3.5-3, 3.3.3.5-4 and 3.3.3.5-5. All the configuration values are consecutive. Thus, some values may have configuration bits in two consecutive memory addresses. Examples of this are Output 6, 12, 19 in our current configuration. As a consequence, the previous outputs may require two writes to configure the desired input signal.

Configuration fields match one to one with the internal counters. So, the field `Output 0` matches with counter 0, `Output 1` with counter 1 and so on.

As a usage example, suppose the user wants to route the signal `pmu_events(0).icnt(0)` to the internal counter 0. The field `Output 0` of the register in Figure 3.3.3.5-2 shall match the index of the signal in the table of inputs. In this case, the index is 2. After this configuration, the event count will be recorded in counter 0. The addresses for counter values range between 0x04 and 0x60.

Counters

The unit in the default configuration contains 24 counters, 32-bit each. The memory address where each counter's value can be accessed ranges between 0x04 and 0x60, as said before. Counter values can be read or written, thus allowing to set the initial value of the counters.

Enable and reset are managed by the base configuration register from Figure 3.3.3.5-1.

Counters can overflow. In such a case, the count will wrap around to 0 and keep counting. The next section (Overflow) describes how to enable the overflow detection interrupts.

Output	Counters	Overflow	MCCU	RDC
0	Yes	Yes	Core 0	Yes
1	Yes	Yes	Core 0	Yes
2	Yes	Yes	Core 1	Yes
3	Yes	Yes	Core 1	Yes
4	Yes	Yes	Core 2	Yes
5	Yes	Yes	Core 2	Yes
6	Yes	Yes	Core 3	Yes
7	Yes	Yes	Core 3	Yes
8	Yes	Yes	No	No
9	Yes	Yes	No	No
10	Yes	Yes	No	No
11	Yes	Yes	No	No
12	Yes	Yes	No	No
13	Yes	Yes	No	No
14	Yes	Yes	No	No
15	Yes	Yes	No	No
16	Yes	Yes	No	No
17	Yes	Yes	No	No
18	Yes	Yes	No	No
19	Yes	Yes	No	No
20	Yes	Yes	No	No
21	Yes	Yes	No	No
22	Yes	Yes	No	No
23	Yes	Yes	No	No

Table 3.3.3.5-1: Crossbar outputs and SafeSU capabilities

Overflow

The user can enable overflow detection for each of the counters in the previous section (Counters). Enables are active high and individual for each counter, as indicated in the `Overflow Interrupt Enable Mask` register depicted in Figure 3.3.3.5-6. If a counter with overflow detection active wraps over the maximum value, the corresponding bit of the `Overflow Interrupt Vector` register depicted in Figure 3.3.3.5-7 will become 1, and AHB interrupt number 6 will become active.

The default AHB interrupt mapping can be modified within the file *ahb_wrapper.vhd*.

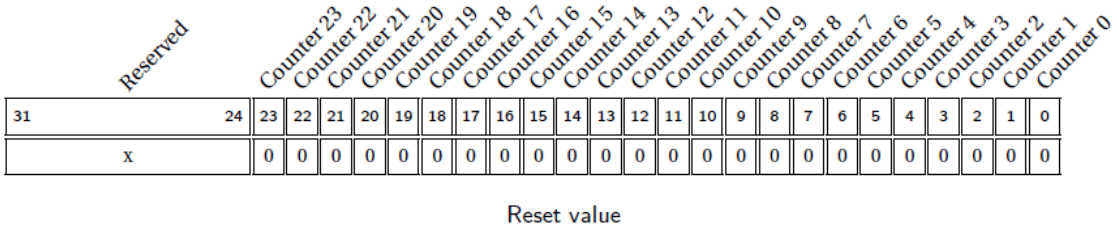


Figure 3.3.3.5-6: SafeSU - Overflow Interrupt Enable Mask (0x064)

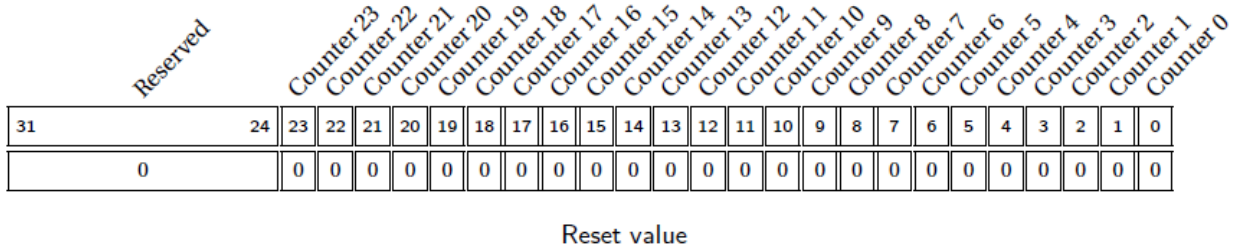


Figure 3.3.3.5-7: SafeSU - Overflow Interrupt Vector (0x068)

Quota

This feature has been replaced by the MCCU and will disappear in future releases. Usage is not recommended.

MCCU

The MCCU allows monitoring for a subset of the input events and tracking the approximate contention that they will cause. Currently, events assigned to counters 0 to 7 can be used as inputs of the MCCU. Thanks to the crossbar, any of the 32 SoC signals can be used by the MCCU.

Figure 3.3.3.5-8 shows the internal elements required to monitor the quota consumption of one core, given that there are four input events. When the events become active, they pass the value assigned in the weight register depicted in Figure 3.3.3.5-10 for the given signal to a series of adders. The addition is subtracted from the corresponding quota register, mapped to addresses 0x088 to 0x094. If the remaining quota is smaller than the cycle contention, an interrupt is triggered.

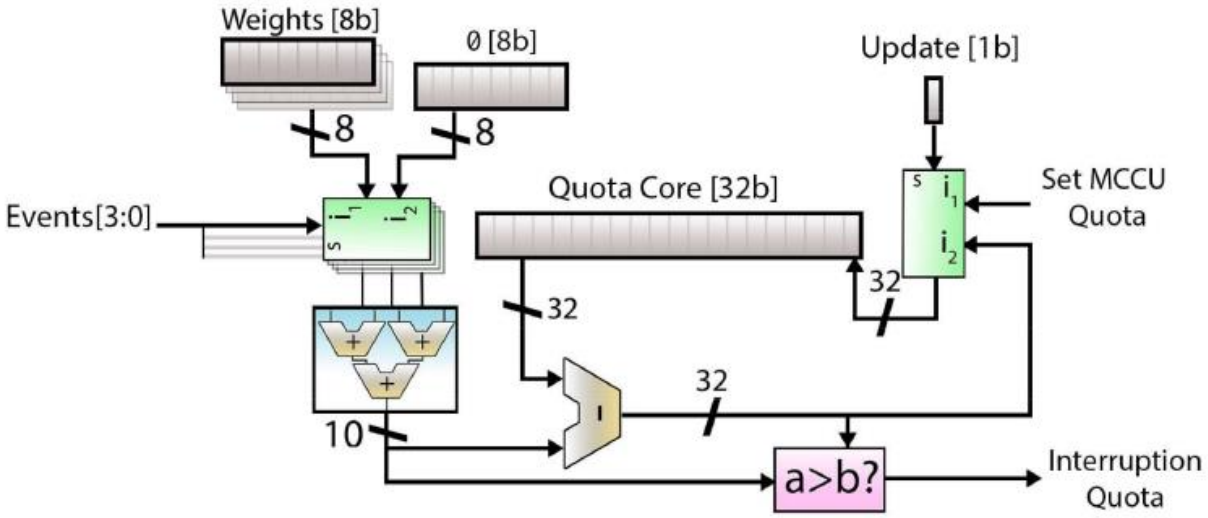


Figure 3.3.3.5-8: SafeSU - Block diagram of the MCCU mechanism for one core

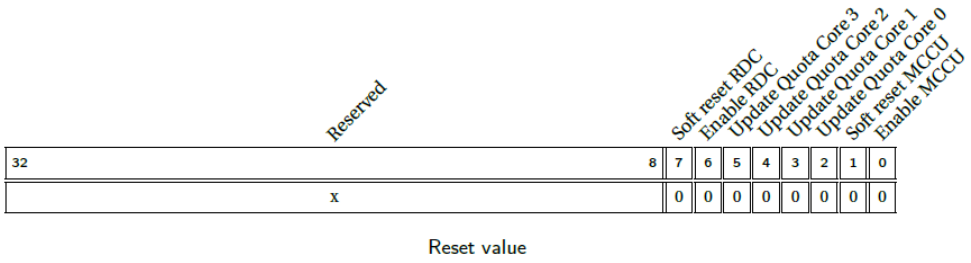


Figure 3.3.3.5-9: SafeSU - MCCU Main Configuration (0x074)

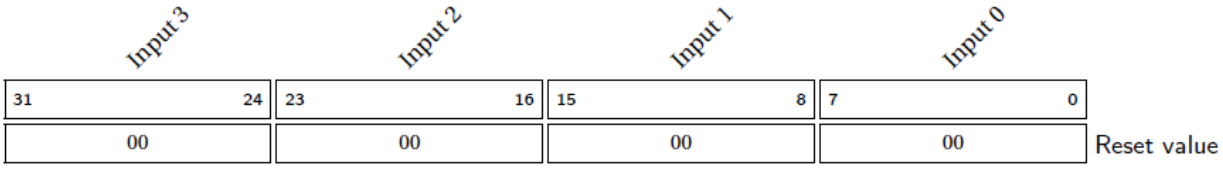


Figure 3.3.3.5-10: SafeSU - MCCU Event Weights Register 0 (shared with RDC; 0x098)

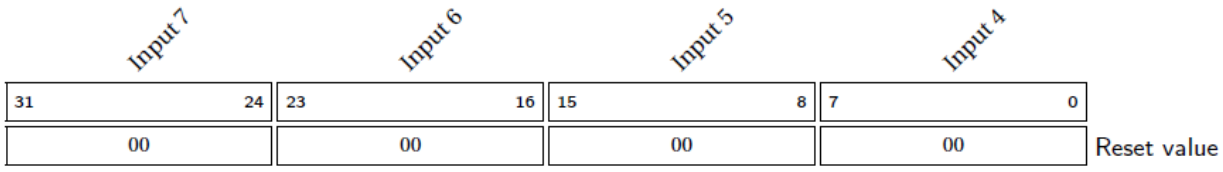


Figure 3.3.3.5-11: SafeSU - MCCU Event Weights Register 1 (shared with RDC; 0x09c)

In the current release, the MCCU can be reset and activated with the respective fields of the `MCCU Main Configuration` register depicted in Figure 3.3.3.5-9. The fields labelled as `Update Quota Core x` are

used to update the available quota of each core (addresses 0x088 to 0x094). While `Update Quota Core x` is high, the content of the corresponding quota register (addresses 0x088 to 0x094) is assigned to the available quota, as configured in registers 0x078 to 0x084. Once released (low), the available quota can start to decrease if the MCCU is active. The current quota can be read while the unit is active.

In the current release, each core can monitor two input events. The MCCU module is parametric. More events can be provided in future releases. Table 3.3.3.5-1 listing the outputs shows the available features for each crossbar output. Under the column MCCU, you can see towards which core quota the event will be computed. The unit provides one interrupt for each of the monitored cores. Quota exhaustion for cores 3, 2, 1, and 0 is mapped to AHB interrupts 10, 9, 8, and 7, respectively.

Weights for each monitored event are registered in the `MCCU Event Weights Register x` registers depicted in Figures 3.3.3.5-10 and 3.3.3.5-11. Currently, each weight is an 8-bit field. Each input of the MCCU maps directly to the outputs of the crossbar. Thus, the weight for the MCCU input 0 corresponds to the signal in crossbar output 0.

RDC

The Request Duration Counter or RDC depicted in Figure 3.3.3.5-12 is comprised of a set of 8-bit counters and comparators that allow monitoring the length of a CCS signal, recording the number of clock cycles of the longest pulse and comparing this number with the defined weight.

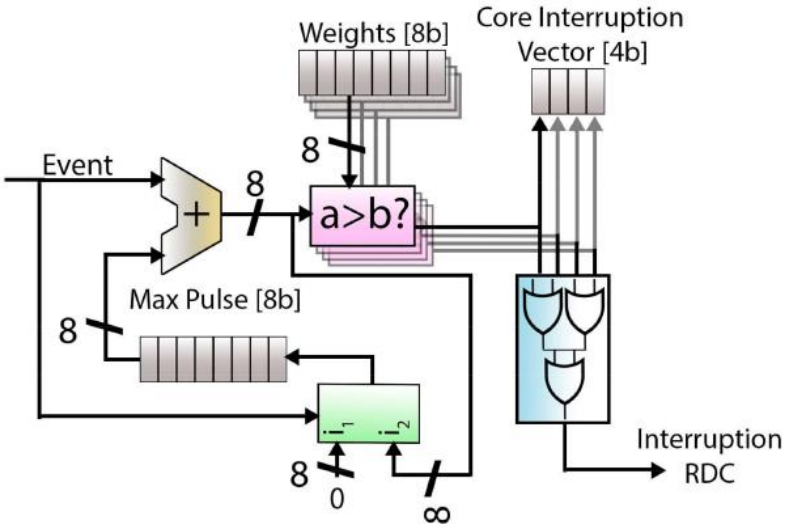


Figure 3.3.3.5-12: SafeSU - Block diagram of the RDC mechanism

The current release provides monitoring for crossbar outputs 0 to 7. The weights for each signal are shared with the MCCU and are stored in the `RDC Event Weights Register x` registers depicted in Figure 3.3.3.5-14. Weights are 8-bit fields. Counters have overflow protection, preventing the count from wrapping over the maximum value. The maximum value for each event (watermarks), is stored in the `RDC Watermark Register x` registers depicted in Figure 3.3.3.5-15.

The RDC shares the main configuration register with the MCCU (Figure 3.3.3.5-9). Through this register, the unit can be reset and enabled through the corresponding fields. Such fields are active high signals.

The unit does provide access to the internal interrupt vector (Figure 3.3.3.5-13), but such information is redundant and may be removed in future releases. Given the current watermarks and assigned weights, the events responsible for the interrupt can be identified. The RDC interrupt has been routed to AHB interrupt 11.

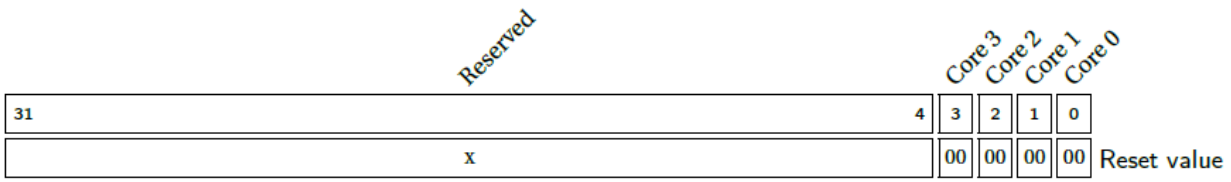


Figure 3.3.3.5-13: SafeSU - RDC Interrupt Vector (0x0A0)

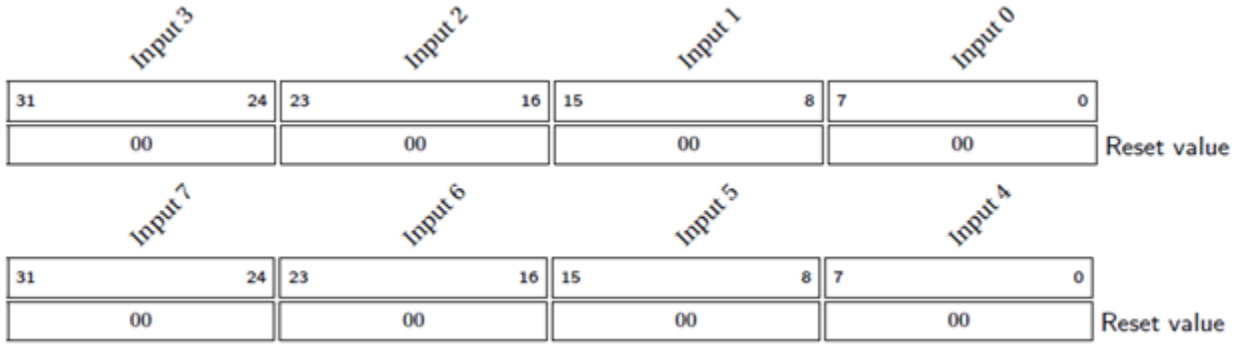


Figure 3.3.3.6-14: SafeSU - RDC Event Weights Registers 0 and 1 (shared with MCCU; 0x098, 0x09C)

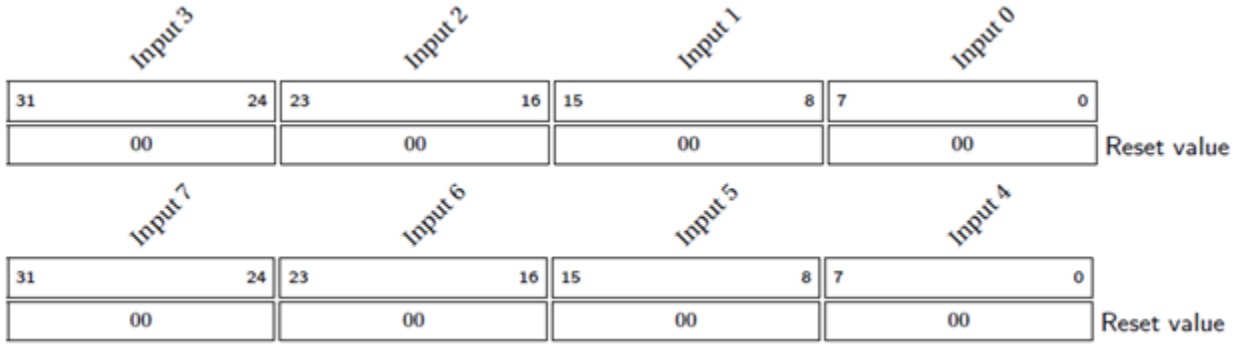


Figure 3.3.3.6-15: SafeSU - RDC Watermark Registers 0 and 1 (0x0A4, 0x0A8)

Software interface

The control registers of the SafeSU, as well as the counters by the SafeSU monitoring the events must be accessed (modified and/or read) only by software components with appropriate privileges. To realize this, the SafeSU registers are mapped in specific physical addresses upon integration in the platform. The hypervisor (FENTISS' XtratuM in the particular case of the SafeSU integration in ISOLDE) is in charge of managing privileges, allowing only specific partitions to be updated, in accordance with SafeSU's registers.

The preferred configuration consists of allowing only a single partition to modify SafeSU's configuration registers and read SafeSU's counters, whereas the other partitions would not be allowed to access those registers. XtratuM guarantees this behavior building on the MMU existing in the NOEL-V cores, which also realizes the RISC-V ISA hypervisor extension.

Overall, the XtratuM hypervisor provides space isolation for the SafeSU's registers, hence achieving freedom from interference. This is in line with safety standards guidelines for items with integrity requirements.

3.3.3.6 Clocking Strategy

SafeSU is designed to share the same clock signaling used for the AHB or AXI interface where it is connected and whose input port is labeled as CLK. The module does not allow for different clocking regions between the programming (APB) and injection (AHB or AXI) interfaces.

3.3.3.7 Verification Strategy

The SafeSU verification strategy incorporates a custom testbench for the simulation environment, generating expected and unexpected input data from the observed AHB or AXI interface. Additionally, as described before, the SafeSU includes a self-test mode that allows to bypass the input events from the crossbar and instead use a specific input pattern where signals are constant.

3.3.4 Time Contract Monitoring Co-Processor (TCCP) – OFFIS

Part of Task 3.3 Monitoring infrastructure.

3.3.4.1 General Information

This module is a modular/composable time contract monitoring co-processor in Safety Island. The stand-alone monitors are developed in VE-VIDES [VE-VIDES], a German funded project but also builds upon earlier work [Tran2020] developed in EU funded projects, e.g., Productive4.0. This co-processor is designed to support a formal Contract-Base Design (CBD) language [Sangiovanni-Vincentelli2012]. Furthermore, it must accept the basic timing properties: Aging, Event Occurrence, and Reaction. Finally, TCCP should be able to monitor different properties simultaneously. The functional and performance requirements of TCCP are presented in Deliverable D1.2 (Section 3.3.2).

3.3.4.2 Purpose and Scope

TCCP monitors the execution of timing properties received from safety island infrastructure and validates them based on the given specifications in a contract-based language. This co-processor is used for safety and security. The result will be presented as violated or not violated as interrupt or Memory-mapped I/O (MMIO).

3.3.4.3 Place in the System

This module is safety-island co-located. It is not decided yet if this module is connected directly to the infrastructure of the Safety-Island or will be integrated into the SafeSU unit (Section 3.3.3), developed by BSC in Task 3.3 “Monitoring Infrastructure”. The idea of integration with SafeSU is to share the observation modules, such as the RDC developed in SafeSU.

3.3.4.4 Block Diagram

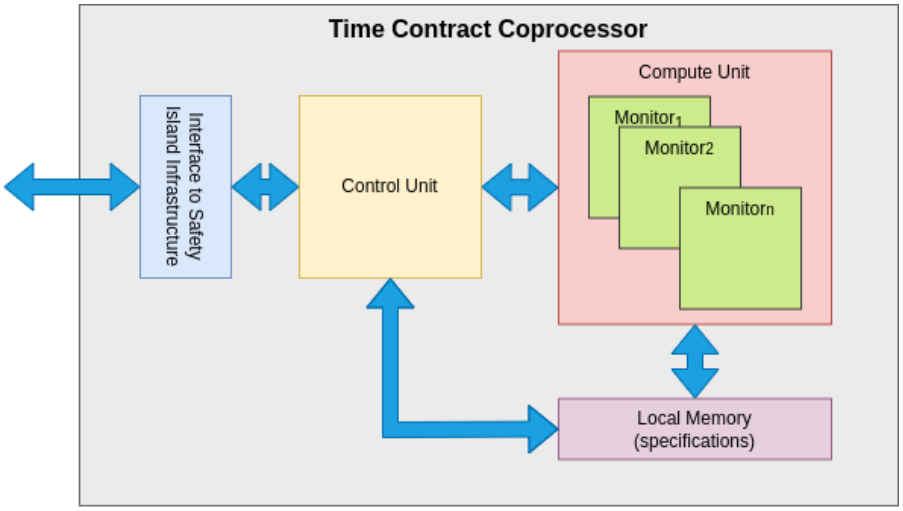


Figure 3.3.4.4-1: TCCP - Block diagram

In Figure 3.3.4.4-1 the abstract schematic of the co-processor is presented. There is an interface to the safety-island infrastructure inside the co-processor, this could be realized using Trace Ingress Port, for example. This schematic shows the interaction of the individual components. The interface unit has a buffer to guarantee the execution of all monitoring requests. The Control unit is responsible for assigning the correct monitor to each request and load the specifications related to each request into the monitors.

Monitors (Aging, Event Occurrence, Reaction) are in the compute unit. There is more than one monitor for each monitoring property to be able to track different requests simultaneously. A local memory is required for storing and loading the monitoring specifications during the initialization and runtime configuration.

3.3.4.5 Interfaces

OFFIS considers connecting the TCCP to the Safety Island infrastructure interface. At the current stage of the design, OFFIS considers using the interface(s) used by SafeSU unit (Section 3.3.3) developed by BSC.

3.3.4.6 Verification Strategy

Our verification strategy incorporates custom testbench for the VHDL simulation environment to validate the functional components in the co-processor. For the system-level debugging, test programs will show expected timing violations that should be detected by the TCCP.

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

Task 3.4, M3-M33, Task Leader: FotoNation

The RISC-V architecture is developed with the perspective of extensibility in mind. Adding support for accelerators enhancing the basic RISC-V instruction set becomes thus straightforward. This task section gathers the modules designed to accelerate intensive mathematical computation operations. These mainly include the multiply and accumulate operations performed in matrix operations, with direct application to neural networks. Here we define the architecture of accelerators that target large-volume numerical computations like the Tensor Processing Unit (TPU, UNIBO), the Vector Processing Unit (VPU, ETHZ), Extension Platform (EXP, TUI) and Parallel Computing Accelerator (PCA, POLITO), or optimize the more specific computations involved by convolutional neural networks – the AI/ML Accelerator (AMA, FotoNation) and the Event-based CNN Accelerator (ECNNA, SAL). The technical solutions range from loosely coupling between the accelerator and the host RISC-V, where the integration is performed using AXI and/or AHB interfaces (AMA and ECNNA), to tightly coupling using the CV-X interface (VPU, PCA), or using both approaches (in TPU) or approaching a custom coupling (used by EXP). Most of the cores developed support both 8bit and 16bit floating point representations (in AMA, TPU and VPU), but also variable range from 8bit to 64bit (for VPU). Almost all of them use some AXI-based memory mapping scheme.

IP	Lead Beneficiary	Type	Dependencies	Licensing
AMA	FotoNation	Core	None	Proprietary
ECNNA	SAL	Core	None	Proprietary
PCA	POLITO	Core	CVA6	Permissive open source (SHL)
TPU	UNIBO	Core	CVA6, CV-X-IF, hwpe-stream, hwpe-ctrl, HCI	Permissive open source (Apache)
VPU	ETHZ	Core	CVA6, CV-FPU	Permissive open source (SHL)
Vector-SIMD Accelerator	IMT	Core	CVA6, NOEL-V, CV-X-IF, Scratchpad	Restrictive open source (GPL-3.0)
EXP	TUI	Core	CVA6, CV-X-IF, one or more AMBA interfaces	Permissive open source (SHL)

Table 3.4-1: Overview of contributions in Task 3.4

3.4.1 AI/ML Accelerator (AMA) – FotoNation

Part of Task 3.4 SIMD/Vector, AI accelerator and tensor processor unit design.

3.4.1.1 General Information

The embracement of artificial intelligence and machine learning based applications is currently on a rising trend. There is an increasing need to provide capable solutions on diverse scales. Whereas on large scale AI applications have shown spectacular results, on the small-scale applications are limited by power consumption and integration area capabilities. Machine learning based applications are typically computing intensive applications; hence the need to apply machine learning operators to vast volumes of data in an effective manner.

One solution to this problem is to design power efficient Machine Learning (ML) accelerators. Fortunately, the ML operators are themselves highly parallelizable, due to the tensor-based nature of the data. Based on the Open Neural Network eXchange (ONNX) format, neural networks that define what operations and the order in which they should be applied to the data can be easily defined in a high-level language. It is then the responsibility of the engineer to come up with optimized hardware able to process those operations in a highly efficient and timely manner, and to develop a specific compiler that translates the high-level ONNX description into low-level instructions for this specific hardware.

3.4.1.2 Purpose and Scope

A hardware accelerator designed for computing ML operators should be efficient in both ways: regarding the power consumed and considering the duration of the processing. In applications such as real-time sensing, which use camera-based systems to perform image processing on the edge, both aspects are critical.

The AI/ML accelerator design exploits the inherent parallelism associated with operators typically encountered in deep convolutional networks. It accelerates operations such as convolution, matrix multiplication, average pooling and element-wise operations like add or multiply. Due to the parallel nature of these operations, duplicated circuits working in parallel are employed to generate the results. This is performed efficiently from both time and power consumption.

3.4.1.3 Place in the System

Figure 3.4.1.3-1 shows the place of the AI/ML accelerator in the system. It can be noticed that all interfaces are standard AXI4 interfaces, meaning that it can be easily integrated in any system with a standard AXI interface.

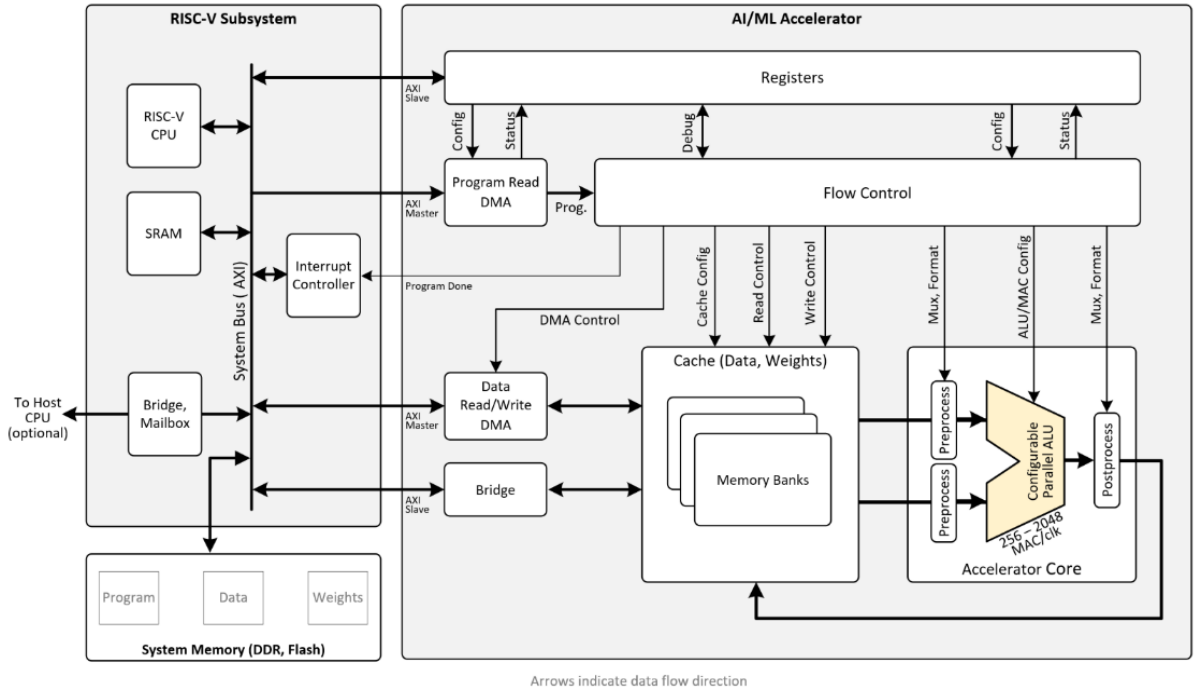


Figure 3.4.1.3-1: AI-ML - Accelerator block diagram and associated system architecture

The system requirements for integrating the AI/ML Accelerator are:

- AXI4 system bus
- CPU
  - Any CPU with AXI bus interfaces.
  - The processor is needed to control the accelerator: configure registers, start the processing, monitor the program running on the accelerator.
- Interrupt Controller
  - At least one interrupt line is needed for the accelerator program done interrupt.
- System Memory
  - Memory (local SRAM, ROM, external DDR, or flash) is needed for storing the CPU routines, accelerator program, parameters, and input/output data.

3.4.1.4 Block Diagram

A block diagram of the AI/ML accelerator is also provided in Figure 3.4.1.3-1. It shows all the main component modules and interfaces. The computing architecture is centered around the Accelerator Core that processes the operands stored into the Memory Banks. The parallelism is achieved by storing operands in parallel memories, thus achieving a processing speed in the range of 256 – 2048 multiply–accumulate (MAC) operations per clock cycle.

The AI/ML Accelerator has the following main features:

- Accelerates most common neural networks layers/operations such as: Convolution, Pooling, Element Wise Add and Mul, Matrix Multiplication;
- Each supported operation is defined by an instruction. Instructions can be aggregated into complex programs that describe the computation of complete neural networks;
- Can operate autonomously or close together with an RISC-V CPU;
- Can be used for both Neural Networks inference and training;

Easy to integrate in any system. All interfaces are standard AXI4 interfaces, 128bit wide.

RISC-V Subsystem Main Features

The RISC-V subsystem jointly works with the accelerator in the following way:

- Configures, starts, and monitors the accelerator module;
- Keeps track of accelerator program execution;
- Provides flow control for complex programs;
- Accesses the accelerator cache to perform operations that are not supported by the accelerator core;
- Assist in debugging the accelerator;
- Communicates to host processor (the host processor can be in fact the RISC-V processor);
- Provides interface to DDR and flash external memories.

Processing Flow

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

- Compile the AI/ML model and load the resulting program and parameters in the system memory;
- Prepare the input maps/data in the system memory;
- Power-up the AI/ML power island – if it is not already on;
- Start the AI/ML clock – if it is not already on;
- Configure the AI/ML registers;
- Set the enable configuration bit for the AI/ML Accelerator;
- Configure the Program DMA and start 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.

3.4.1.5 ISA

The instruction set architecture defines the high-level operations supported by the AI/ML accelerator. They leverage its capability of operating independently of the host processor. As the ISA is still under development, in the following the supported operations are represented as programming API.

```
/**
  * Loads a data map from (external – DDR, Flash) system memory into
  * accelerator’s cache (internal) memory
  */
void load(
    void *   axi_addr,    // system memory address
    uint32_t addr,        // cache memory start address
    uint32_t width,       // map’s row width (in multiples of bytes
                          //     given by data format)
    uint32_t height,      // map’s number of rows
    uint32_t line_stride, // row stride of the map in system memory
    uint32_t format       // data format:
                          //     0: FP16, 1: FP8,
                          //     2: int8, 3: unit8
);
```

```
/**
  * Transfers a data map from cache (internal) memory to system
  * memory.
  */
void save(
    void *   axi_addr,    // system memory address
    uint32_t addr,        // cache memory start address
    uint32_t width,       // map's row width (in multiples of bytes
                          //     given by data format)
    uint32_t height,      // map's number of rows
    uint32_t line_stride, // row stride of the map in system memory
    uint32_t format       // data format:
                          //     0: FP16, 1: FP8
);

/**
  * Applies a 2D convolution on input composed of several channels.
  */
void conv2d(
    uint32_t in_data_addr,  // cache memory input map address
    uint32_t weights_addr,  // cache memory address for weights
    uint32_t bias_addr,     // cache memory address for biases
    uint32_t out_data_addr, // cache memory output map address
    uint32_t in_channels,   // number of input channels
    uint32_t out_channels,  // number of output channels
    uint32_t data_width,    // input map's row width (in bytes)
    uint32_t data_height,   // input map's number of rows
    uint32_t kern_width,    // convolving kernel width
    uint32_t kern_height,   // convolving kernel height
    uint32_t in_stride,     // row stride of the input map
    uint32_t out_stride,    // row stride of the output map
    uint32_t padding        // padding added to all four sizes
                            //     of the input

);

/**
  * Applies a 1D convolution on input composed of several channels.
  */
void conv1d(
    uint32_t in_data_addr,  // cache memory input map address
    uint32_t weights_addr,  // cache memory address for weights
    uint32_t bias_addr,     // cache memory address for biases
    uint32_t out_data_addr, // cache memory output map address
    uint32_t in_channels,   // number of input channels
    uint32_t out_channels,  // number of output channels
    uint32_t data_width,    // input map's row width (in bytes)
    uint32_t kern_width,    // convolving kernel width
    uint32_t in_stride,     // row stride of the input map
    uint32_t out_stride,    // row stride of the output map
    uint32_t padding        // padding added to both sizes
                            //     of the input
```

```
);

/**
  * Applies a 2D max pooling on input composed of several channels.
  */
void max_pool2d(
    uint32_t in_data_addr,  // cache memory input map address
    uint32_t out_data_addr, // cache memory output map address
    uint32_t no_channels,   // number of channels
    uint32_t data_width,    // input map's row width (in bytes)
    uint32_t data_height,   // input map's number of rows
    uint32_t kern_width,    // kernel width
    uint32_t kern_height,   // kernel height
    uint32_t in_stride,     // row stride of the input map
    uint32_t out_stride,    // row stride of the output map
    uint32_t padding        // padding added to both sizes
                            //     of the input
);


/**
  * Applies an activation function over the input map.
  */
void activation(
    uint32_t in_data_addr,  // cache memory input map address
    uint32_t out_data_addr, // cache memory output map address
    uint32_t no_channels,   // number of channels
    uint32_t data_width,    // input map's row width (in bytes)
    uint32_t data_height,   // input map's number of rows
    uint32_t in_stride,     // row stride of the input map
    uint32_t out_stride,    // row stride of the output map
    uint32_t function       // activation function type:
                            //    0: ReLU, 1: Sigmoid,
                            //    2: SiLU, 3: LeakyReLU
);


/**
  * Applies a linear transform to the input data. Both weights size
  * and in_data size equals to no_channels x data_width x data_height.
  * out_data will be a scalar.
  */
void fully_connected(
    uint32_t in_data_addr,  // cache memory input map address
    uint32_t out_data_addr, // cache memory output map address
    uint32_t weights_addr,  // cache memory address for weights
    uint32_t bias_addr,     // cache memory address for biases
    uint32_t no_channels,   // number of channels
    uint32_t data_width,    // input map's row width (in bytes)
    uint32_t data_height,   // input map's number of rows
    uint32_t in_stride      // row stride of the input map
);
```

```
 /**
   * Applies element-wise addition on input maps.
   */
void eltwise_add(
    uint32_t in1_data_addr, // cache memory map address for input1
    uint32_t in2_data_addr, // cache memory map address for input2
    uint32_t out_data_addr, // cache memory output map address
    uint32_t no_channels,   // number of channels
    uint32_t data_width,    // input map’s row width (in bytes)
    uint32_t data_height,   // input map’s number of rows
    uint32_t in1_stride,    // row stride of input 1 map
    uint32_t in2_stride,    // row stride of input 2 map
    uint32_t out_stride     // row stride of the output map
);

/**
  * Applies element-wise multiplication on input maps.
  */
void eltwise_mul(
    uint32_t in1_data_addr, // cache memory map address for input1
    uint32_t in2_data_addr, // cache memory map address for input2
    uint32_t out_data_addr, // cache memory output map address
    uint32_t no_channels,   // number of channels
    uint32_t data_width,    // input map’s row width (in bytes)
    uint32_t data_height,   // input map’s number of rows
    uint32_t in1_stride,    // row stride of input 1 map
    uint32_t in2_stride,    // row stride of input 2 map
    uint32_t out_stride     // row stride of the output map
);

/**
  * Computes the matrix product of two tensors.
  */
void mat_mul(
    uint32_t in1_data_addr, // cache memory map address for input1
    uint32_t in2_data_addr, // cache memory map address for input2
    uint32_t out_data_addr, // cache memory output map address
    uint32_t in1_width,     // input 1 map’s row width (in bytes)
    uint32_t in1_height,    // input 1 map’s number of rows
    uint32_t in2_width,     // input 2 map’s row width
    uint32_t in1_stride,    // row stride of input 1 map
    uint32_t in2_stride,    // row stride of input 2 map
    uint32_t out_stride     // row stride of the output map
);
```

3.4.1.6 Interfaces

All interfaces are AXI4. Additionally, at least one interrupt line (program done) is needed for integration.

Two address zones are needed in the system CPU address space:

- Configuration and Status registers;
- Address space for the AXI to Cache bridge.

The cache data banks store 16 x 16-bit values (also called channels) or 32 x 8-bit values at each address. The mapping between the cache addresses and the CPU/AXI address space is detailed in Figure 3.4.1.6-1. Only AXI accesses aligned at addresses multiple of 16 bytes (128-bit) are allowed.

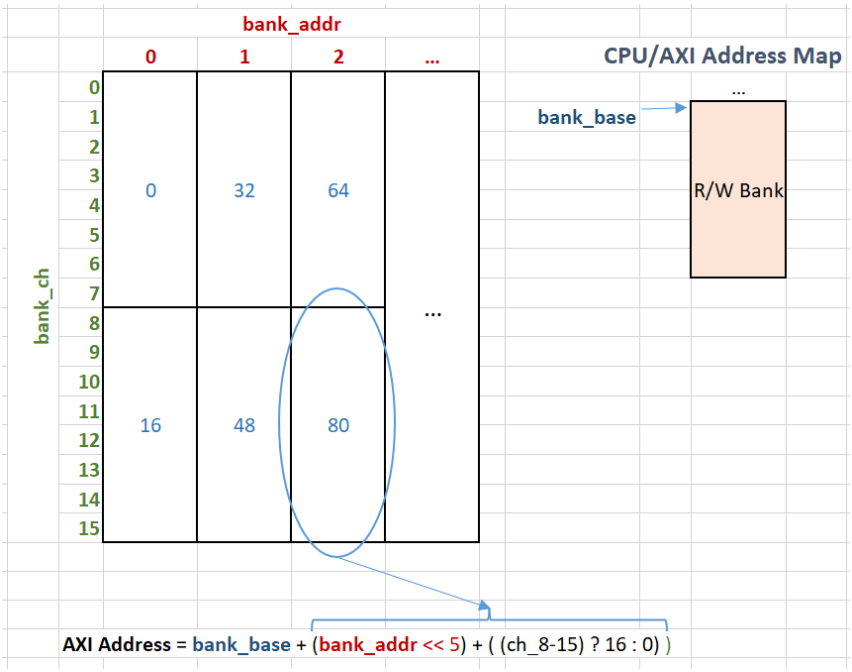


Figure 3.4.1.6-1: AI-ML Accelerator - Address mapping scheme

3.4.1.7 Sub-Modules

AI/ML Sub-modules

The AI/ML Accelerator contains the following main sub-modules:

- Cache:
  - Used for both data (activation maps) and parameters (weights);
  - Parametrized size;
  - Multiple values can be read and written in each clock cycle, ensuring that the accelerator core is not starved of data.
- Accelerator Core:
  - Contains a Configurable Parallel ALU;
  - The parallel ALU processes data from the cache and writes the results back to the cache;
  - Operates on 8-bit and 16-bit data. Has higher precision accumulators;
  - Parametrizable number of MACs: 256, 512, 1024, 2048;
  - Can use all available MACs each clock cycle;
  - Supported operations:
    - 1D, 2D Convolution
    - Pooling
    - Activation Function
    - Fully Connected
    - Element Wise
    - Matrix Multiplication
  - Preprocess / Postprocess modules prepare/arrange the data ensuring that the accelerator core is not data starved or backpressured: multiplexing, aligning, format conversion.

- Registers:
  - Contains all configuration and status registers;
  - Gives the CPU full control of the accelerator core when complex interaction between the core and the CPU is needed, or for debugging.
- AXI to Cache Bridge
  - AXI4 Lite Slave interface;
  - Allows the CPU to directly access the cache;
  - It has a 128-bit-wide data interface, making it possible to be used by a CPU with Single Instruction Multiple Data (SIMD) support, providing access to 8 x 16-bit values or 16 x 8-bit values per clock cycle.
- Program Read DMA:
  - AXI4 Master interface;
  - Used to read programs, ensuring that the accelerator core is never starved of instructions.
- Flow control
  - Decodes the incoming instructions from the Program Read DMA or from the CPU;
  - Controls the data flow through the processing pipeline, from cache read, through ALU processing, to writing back to the cache.
- DMA Read/Write DMA
  - AXI4 Master interface;
  - Used for high-bandwidth data transfer between the cache and memories in the system (DDR, Flash, local SRAM);
  - Operate at the maximum available AXI bandwidth;
  - Used to read and write data (activation maps) and parameters (weights).

CPU Sub-System Sub-Modules

The following modules are required as part of a CPU (RISC-V) subsystem:

- RISC-V CPU
  - Controls the RISC-V subsystem and the AL/ML Accelerator;
  - Supports floating point operations;
  - Support for SIMD processing for efficient processing of data in the accelerator cache;
- Interrupt Controller
  - Manages the subsystem interrupts, including the accelerator interrupts;
- SRAM
  - Local SRAM for the CPU use and for the accelerator data and program;
- Bridge/mailbox
  - Optional modules for connecting and communicating with a host processor.

3.4.1.8 Clocking Strategy

The AI/ML Accelerator has two clock domains. Both clocks can be gated externally when the module is not in use.

- AI/ML Accelerator clock
  - Used by all submodules;
  - The clock to each sub-module is gated when the sub-modules are not active;
  - The clock to each SRAM of the cache memories is gated anytime when there are no memory transactions;
- AXI interface clocks
  - Only used by the AXI4 interface modules.

Depending on the system/implementation (FPGA, ASIC technology) the two clocks can be asynchronous or can have the same frequency (in this case no synchronization logic between the two clock domains is needed).

3.4.1.9 Reset Strategy

There are two asynchronous reset inputs, active low, one for each clock domains:

- AI/ML Accelerator reset;
- AXI4 reset.

Additionally, software reset / initialization functionality is provided through the configuration registers.

3.4.1.10 Power Management Strategy

Clock Gating

As mentioned above, the clocks to each AI/ML accelerator sub-module and SRAM instance are only active when the corresponding module/SRAM processes data.

Memory Sleep

Each SRAM instance inside the cache has its own sleep input. The sleep signals are automatically controlled by the control logic module. Any SRAM that is not needed for the current processing has the clock gated and is put to sleep.

Power Gating

The whole AI/ML accelerator can be placed on a separate power island. Once a program is completed, the accelerator power can be switched off as it is always reinitialized every time a new program is executed.

3.4.1.11 Debugging Strategy

The RISC-V processor has standard JTAG interface for in-circuit debugging.

The AI/ML Accelerator provides several ways of debugging its program:

- AXI to Cache Bridge – Provides direct access to all cache values;
- Extra interrupt lines are available and can be used by the CPU to monitor the progress of the AI/ML Accelerator program;
- Status registers give insight into the status of each AI/ML sub-module;
- Single instructions can be pushed to the AI/ML Accelerator in debug mode, bypassing the program DMA module.

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

Part of Task 3.4 SIMD/Vector, AI accelerator and tensor processor unit design.

3.4.2.1 General Information

Convolutional neural networks (CNNs) have become a standard in computer vision given their capability to process complex visual data, in contrast to more hand-crafted traditional approaches. They are characterized by high complexity and require substantial amounts of memory, computing power and energy, which can be challenging in resource-constrained environments and applications.

One of the promising ways to tackle this is through event-based processing, a paradigm that arouses inspired by the biological mechanisms that the brain uses to decode signals. This type of processing is highly sparse in nature, and therefore can be done with a much smaller memory footprint and savings in computation.

This module is an event-based CNN accelerator capable of exploiting the inherent sparsity present in event-based data. It supports convolution, maxpool and sparse fully connected.

3.4.2.2 Purpose and Scope

Because of the nature of event-based data, where information is only generated by changes in the visual scene, our accelerator is particularly suitable for dynamic environments and applications such as autonomous vehicles.

3.4.2.3 Place in the System

The accelerator has an AHB-Lite subordinate port that writes to a bank of configuration registers that set the following parameters:

- Stride: supports only power of 2 values.
- Padding.
- Kernel size: maximum allowed size of 7x7
- Activation functions: ReLU, and Leaky ReLU
- Quantized precision: 8bits

It can work either coupled with a DMA, or with the CPU that writes the data in and out from the accelerator through routines triggered by interruptions.

3.4.2.4 Block Diagram

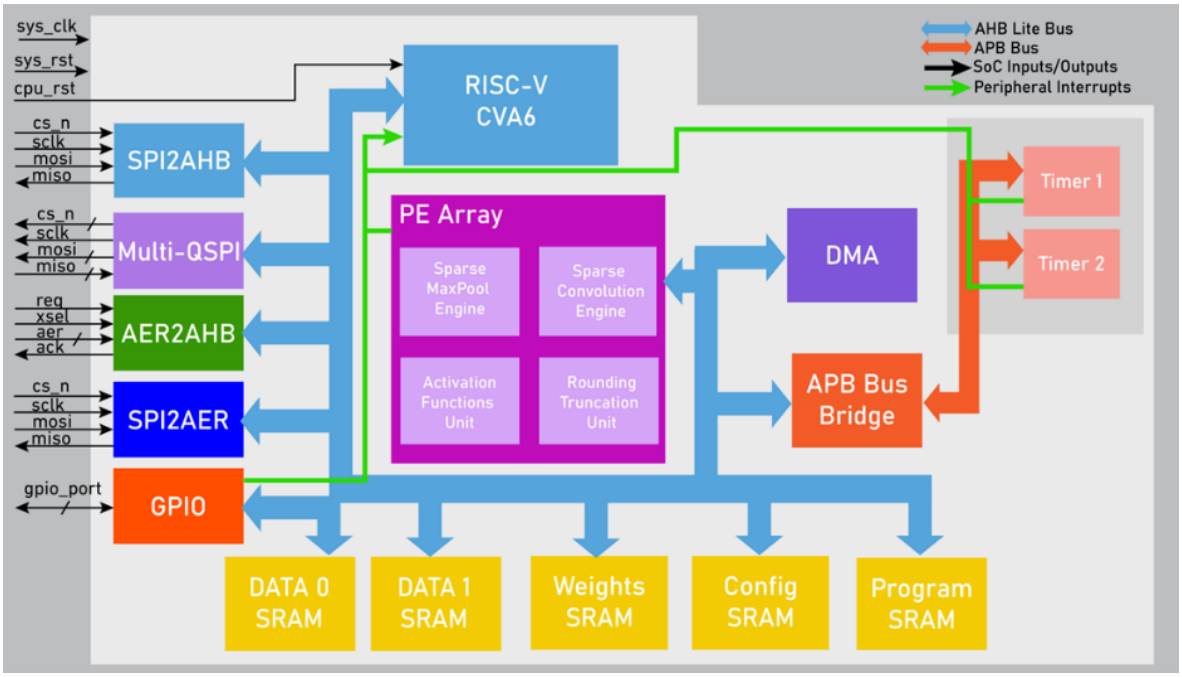


Figure 3.4.2.4-1: Event-Based CNN Accelerator - Block diagram

The block diagram in Figure 3.4.2.4-1 illustrates a proposed system integration for the accelerator, modeled after an Application-Specific Integrated Circuit (ASIC) recently fabricated using 65nm TSMC technology. The system features a CV32E40P RISC-V CPU (in the diagram replaced by the CVA6) connected via a 64bits AHB-Lite Bus. This CPU commands the accelerator, which processes inputs from two peripheral devices: an Address Event Representation (AER2AHB) block for native interfacing with a Dynamic Vision Sensor (DVS) camera, and a Serial Peripheral Interface (SPI) slave block (SPI2AHB) that can receive events from an FPGA or another microcontroller.

The system was integrated with five 16KiB SRAM memories. Two of these are allocated for data storage, one for storing weights, another for the program code, and the final one for holding the CNN layer configurations accessed by the CPU. Also, the system includes a DMA that, with a multi-QSPI peripheral (consisting of eight QSPI master operated in parallel under a shared controller), enables high-speed data transfers up to 1.6Gbps. Peripheral interfaces also include a 32-bit General-Purpose Input/Output (GPIO), and two timers connected to an APB bus, which facilitate events integration and timestamping tasks.

3.4.2.5 Debugging Strategy

The ASIC version counts with an SPI peripheral to AHB-Lite master peripheral (SPI2AHB in the block diagram) that works as a debugger. For the FPGA implementation, any debugging strategy can be used where an AHB-Lite controller port can be reached. Additionally, the accelerator counts with multiple interruption lines that can trigger the CPU after each processing step.

3.4.3 Parallel Computing Accelerator (PCA) – POLITO

Part of Task 3.4 SIMD/Vector, AI accelerator and tensor processor unit design.

3.4.3.1 General Information

The parallel computing accelerator is a loosely coupled processing cluster architecture, which can operate with approximate arithmetic units also supporting on-line change of the approximation level.

3.4.3.2 Purpose and Scope

In the last years AI and ML have gained a lot of popularity in different fields ranging from automotive, to aerospace, speech recognition, image, and video processing, thus enabling to possibilities and challenges. However, most of the computing schemes and algorithms employed in AI and ML have noteworthy computational complexity, which can be difficult to manage in software. As a result, hardware accelerators are an interesting and viable solution to such a problem.

From another perspective, most of the applications which take advantage of AI and ML exhibit intrinsic resilience to arithmetic errors. For this reason, the approximate computing paradigm can be exploited to implement approximate arithmetic operators that introduce errors in the computed values with negligible performance loss in terms of accuracy.

A loosely coupled accelerator, based on an approximate processing cluster architecture, offers the possibility to accelerate the computation in an approximate computing fashion. This is achieved by relying on a set of processing engines working concurrently.

3.4.3.3 Place in the System

The approximate computing cluster is loosely coupled to the CVA6 system CPU via the system-level crossbar. Additional AXI ports are used to access the system memory through an internal DMA unit.

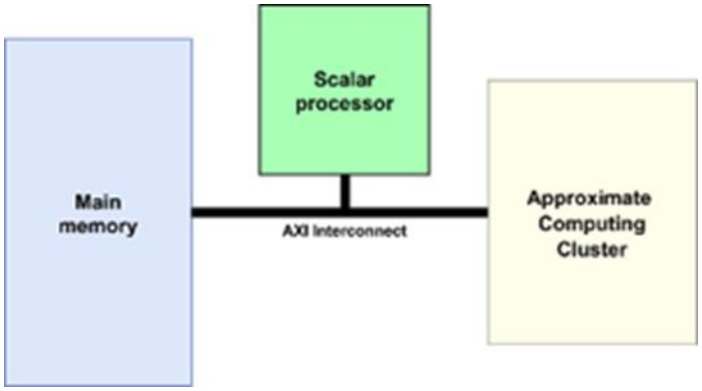


Figure 3.4.3.3-1: Parallel Computing Accelerator - Place in the system

3.4.3.4 Block Diagram

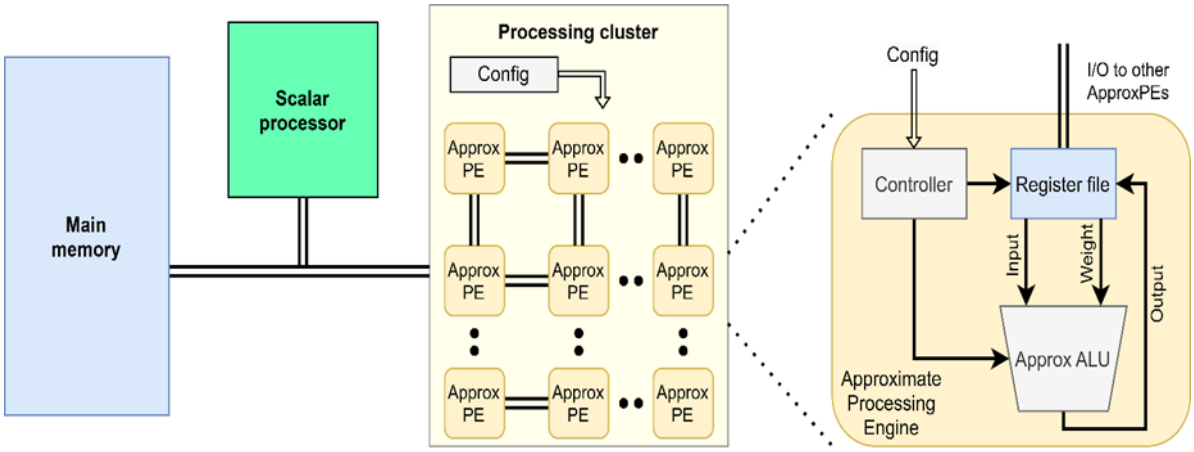


Figure 3.4.3.4-1: Parallel Computing Accelerator - Block diagram

As shown in Figure 3.4.3.4-1, the proposed parallel computing accelerator relies on an approximate processing cluster architecture. The cluster is made of a programmable number of approximate processing elements, each of which contains a register file and an ALU. The accelerator can be configured via different parameters. Part of these parameters (such as the maximum number of processing elements and the maximum precision) are configured at design time, while other parameters (such as the approximation level) can be set at the run-time by writing configuration registers. These registers' content triggers proper masking mechanisms to change the approximation level during the computation to save power. The accelerator is connected to the CVA6 system architecture through an AXI interface, and it processes a subset of ALU operations in approximate mode.

3.4.4 Tensor Processing Unit (TPU) – UNIBO

Part of Task 3.4 SIMD/Vector, AI accelerator and tensor processor unit design.

3.4.4.1 General Information

A Tensor Processing Unit (TPU) speeds-up FP matrix-matrix, matrix-vector, and vector-matrix operations through a simplified programming interface, providing memory-mapped configuration registers that can also be accessed through an eXtension Interface (XIF), supporting 8-bit and 16-bit precision.

3.4.4.2 Purpose and Scope

A hardware TPU for matrix multiplication in the space domain offers significant advantages over programmable multicore accelerators in terms of both performance and energy efficiency. The primary reason for this superiority lies in the specialized nature of TPUs, which are explicitly designed to handle the operations fundamental to all Deep Learning and Machine Learning models, such as high-throughput matrix multiplications and additions.

TPUs use arrays of arithmetic units specifically optimized for tensor operations, enabling the execution of thousands of operations in parallel to achieve higher throughput and lower latency in matrix multiplication tasks. In contrast, programmable multicore accelerators, while versatile, are not specifically optimized for such operations, leading to less efficient execution of high-volume matrix multiplications due to their more generalized processing cores.

TPUs also provide high energy efficiency. Their specialized hardware is designed to maximize operations per watt of power consumed, a critical consideration in space applications where power availability is limited. TPUs achieve this efficiency through optimizations such as reduced precision arithmetic, which is suitable for neural network computations and significantly reduces power consumption with reduced accuracy loss. Programmable multicore accelerators, on the other hand, tend to consume more power for equivalent tensor operations because they lack these specialized optimizations and often operate at higher precision than necessary for the task.

3.4.4.3 Place in the System

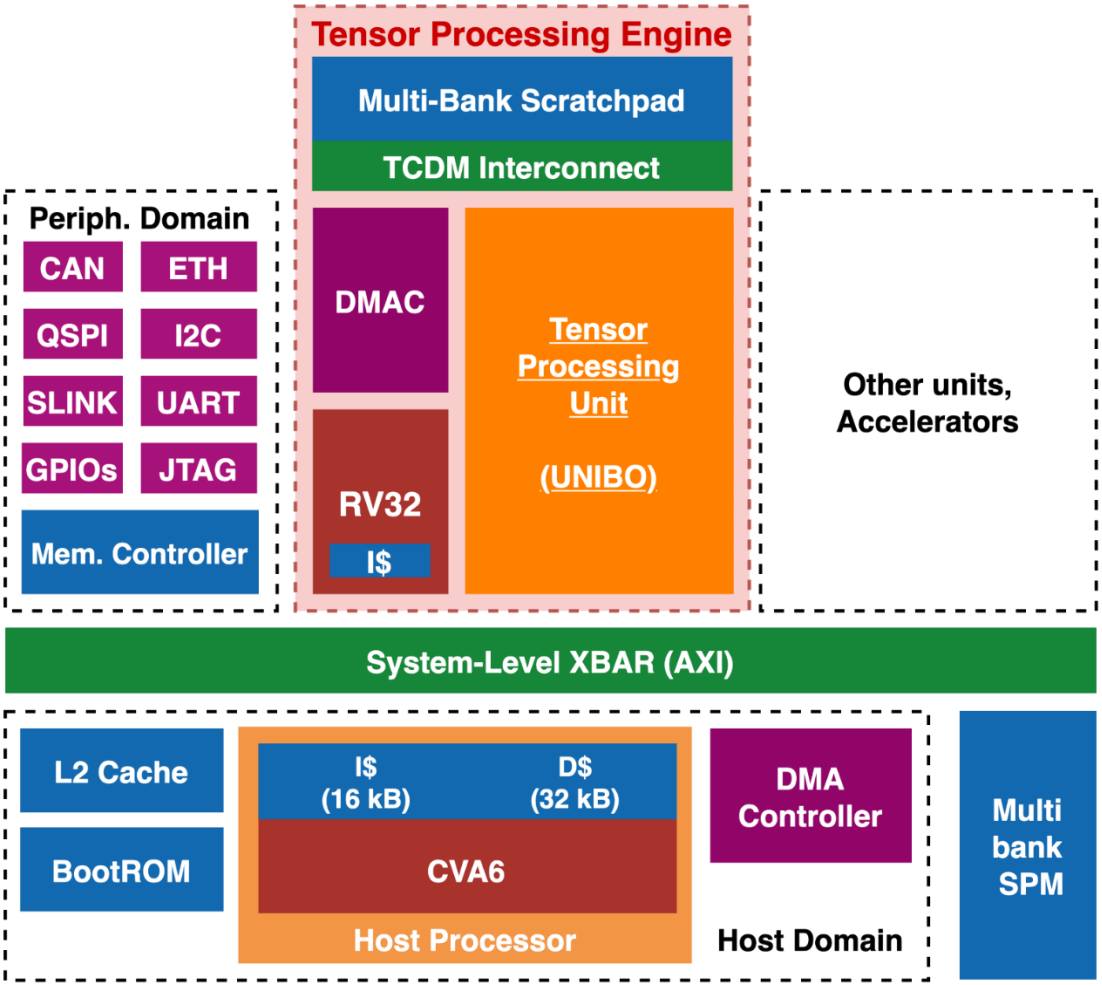


Figure 3.4.4.3-1: Tensor Processing Unit - Place in the system

The TPU is localized inside of a tensor processing cluster together with a DMA controller, RISC-V, and scratchpad memory. This is shown in Figure 3.4.4.3-1. Figure 3.4.4.3-2 shows two different mechanisms under exploration to control the TPU. In the leftmost option, the TPU is controlled by means of a *hwpe-ctrl* target[11] exposing a set of memory-mapped registers. In the rightmost option, the TPU is controlled by means of direct communication between the processing core's register file and a set of registers in the TPU via an instruction-set extension realized using the CORE-V XIF interface[12].

[11] https://github.com/pulp-platform/hwpe-ctrl

[12] https://github.com/openhwgroup/core-v-xif

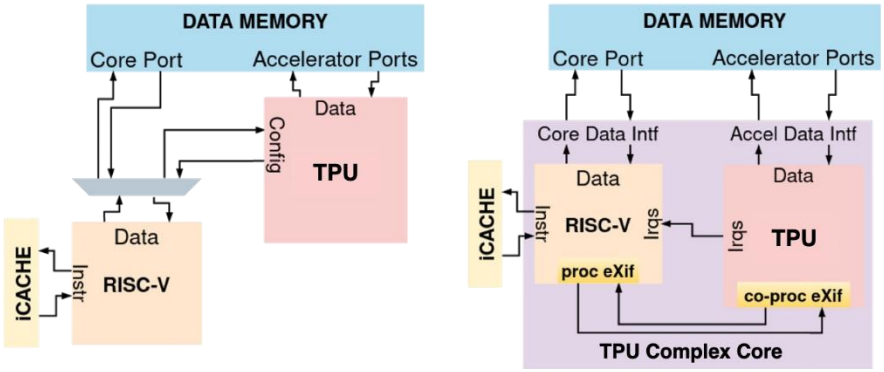


Figure 3.4.4.3-2: Tensor Processing Unit - Controlling

3.4.4.4 Block Diagram

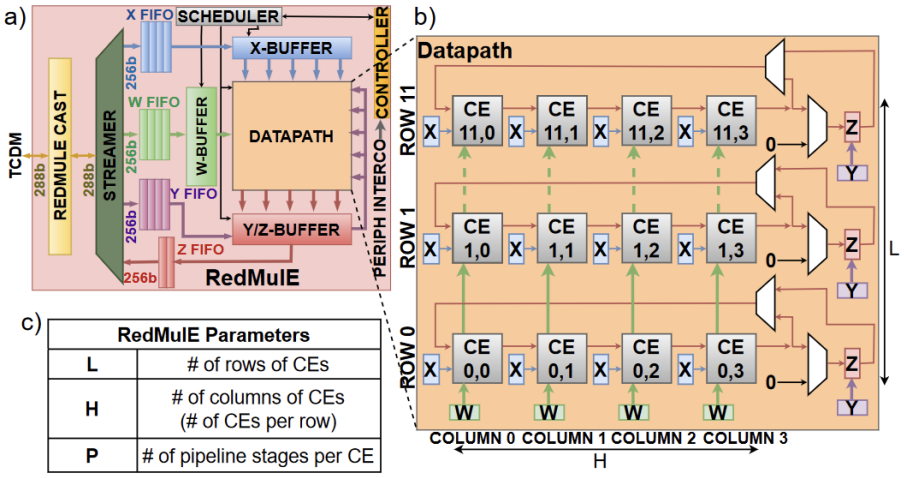


Figure 3.4.4.4-1: Tensor Processing Unit - Block diagram

Figure 3.4.4.4-1 details the internal architecture of the Tensor Processing Unit. The data path is constituted of a series of Computing Elements (CEs) in a systolic array configuration. The default array size is L=12 rows x H=4 columns. Each CE contains a 16-bit floating point fused-multiply-add (FMA) unit and P=4 pipeline stages. The systolic array is fed by one stationary input (X) and one non-stationary input (W) broadcasted along the column direction. Both are streamed through a high-bandwidth streamer using the Heterogeneous Cluster Interconnect (HCI) protocol[13]. The output buffer (Z) can also be preloaded with existing content (Y) to implement a complete General Matrix Multiply (GEMM) functionality.

[13] https://github.com/pulp-platform/hci

3.4.4.5 Power Management Strategy

Dynamic clock gating of the internal computing elements depending on the utilization in the executed kernel.

3.4.5 Vector Processing Unit (VPU) – ETHZ

Part of Task 3.4 SIMD/Vector, AI accelerator and tensor processor unit design.

3.4.5.1 General Information

The RISC-V vector accelerator with multi-precision capabilities is a tightly coupled accelerator designed to work in tandem with the CVA6 Application-Class RISC-V core to accelerate parallel workloads with support for multiple data formats.

3.4.5.2 Purpose and Scope

In an era dominated by data, the demand for computational power has skyrocketed, particularly in ML and signal processing. These domains are characterized by their intensive computational requirements, often necessitating the manipulation and analysis of vast datasets to derive meaningful insights. A pivotal challenge in these areas is the efficient handling of multi-precision data formats, which vary in precision and are critical for optimizing performance and accuracy in computational tasks. Addressing this challenge requires innovative hardware solutions capable of accelerating these workloads while maintaining flexibility in data precision.

To effectively tackle these challenges, we design a RISC-V vector processor with multi-precision capabilities, following the RISC-V V 1.0 specifications. This processor aims to enhance the computational efficiency of ML and signal processing applications by offering tailored support for multi-precision data formats. By leveraging the RISC-V architecture, known for its simplicity, modularity, and extensibility, this project endeavors to introduce a versatile solution that can address the evolving demands of these computationally intensive fields.

The vector accelerator will be able to accelerate parallelizable workloads from various domains (ML, signal processing, linear algebra, etc.) and support multiple integer and floating-point data formats from 64 bits down to 8 bits. The vector accelerator targets higher performance and efficiency if compared to the scalar-only computation of the same task, as a single vector instruction triggers the computation of multiple elements in parallel, amortizing the instruction fetching-decode-issue cost more effectively than its scalar-only counterpart. To maximize performance, our vector accelerator exploits lane-parallelism and packed-SIMD parallelism.

3.4.5.3 Place in the System

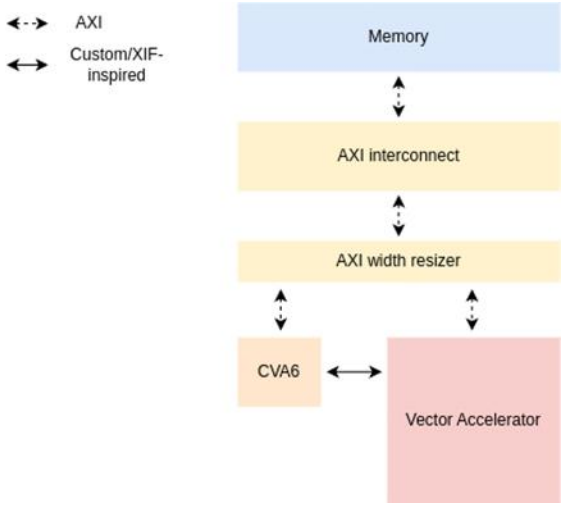


Figure 3.4.5.3-1: Vector Processing Unit - Place in the system

The system architecture is described in Figure 3.4.5.3-1. The vector accelerator is tightly coupled to CVA6 with a custom or XIF-inspired [OpenHWGroup2021] interface and communicates via AXI with the Memory. CVA6 is the RISC-V core and dispatches all the RISC-V vector instructions to the vector accelerator.

3.4.5.4 Block Diagram

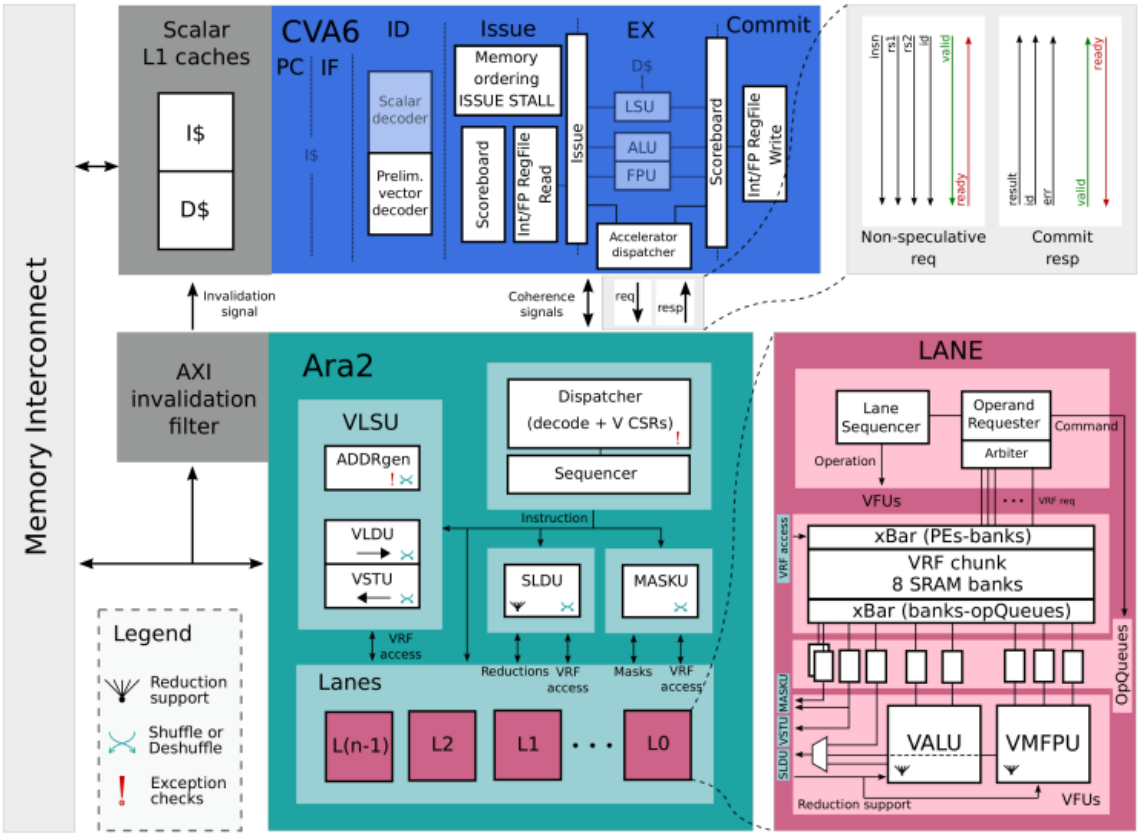


Figure 3.4.5.4-1: Vector Processing Unit - Block diagram

The architectural diagram in Figure 3.4.5.4-1 refers to an implementation that supports a custom interface between CVA6 and the vector accelerator (Ara2 in the schematic).

The vector accelerator is composed of parallel lanes (L0 -> L(n-1)) that contain chunks of the Vector Register File (VRF), the internal buffer for vector elements. Each lane also contains a vector ALU and a vector Multiplier and Floating-Point Unit (VMFPU).

The vector accelerator comprises vector CSRs as specified by the RISC-V V 1.0 specifications, a decoder, a sequencer to account for dependencies between instructions, a private vector load-store unit (AXI-compliant), a slide unit (SLDU) to handle vector permutations and shuffles, and a mask unit, to work with 1-bit-granular mask vector, which implement predicated execution.

The architecture bus is 64-bit * #Lanes, while the memory data bus is 32-bit * #Lanes.

3.4.5.5 ISA

The vector accelerator is based on RISC-V V 1.0 [RVI2021].

3.4.5.6 Interfaces

The architecture communicates with the memory through AXI protocol, and with a custom accelerator interface or XIF-inspired interface with the CVA6 scalar core.

3.4.5.7 Sub-Modules

The vector accelerator cannot fetch instructions from memory; therefore, it needs a scalar core that can dispatch them. Moreover, it internally uses the CVFPU to process floating-point data.

CVA6

CVA6 [OpenHWGroup2024] is a RV64GC Application-Class scalar that, via a custom or XIF interface, can dispatch instructions to a tightly coupled accelerator.

CVFPU

CVFPU [OpenHWGroup2023] is the floating-point unit used within the architecture, currently maintained by OpenHW Group.

3.4.5.8 Reset Strategy

The module is reset with an active-low reset shared by all the architecture modules. All the internal sequential status is reset, except for the internal vector register file, which is implemented with SRAM banks.

3.4.6 Vector-SIMD Accelerator – IMT

Part of Task 3.4 SIMD/Vector, AI accelerator and tensor processor unit design.

3.4.6.1 General Information

SIMD is a processor able to process multiple data elements in parallel with a single instruction. A classical processor can process a single data element per instruction. The SIMD accelerator exploits available data level parallelism and may improve performance. The SIMD processor may also reduce the size of the program code as fewer instructions are needed to process the data. However, vector processors need to define the vector size, load data, compute them and write back. On scalar processors, work is performed on one element at the time and overhead instructions to perform index computation and looping are required.

The SIMD processors are also called vector processors and, like scalar processors, have a register file. In a vector register the user can store multiple data elements. Some common dimensions for this kind of register are 128 bits, 256 bits or 512 bits. These registers may also allow narrow data types for subword level parallelism. For example, if you have a vector register of 128 bits, you can store four integer numbers of 32 bits, or 16 8 bits integers.

A vector register typically stores multiple elements in it. For a vector processor, for each vector lane there exists an arithmetic unit allowing parallel processing.

Common applications for vector processors are operations like addition, subtraction, multiplication, and addition of all vector elements. Other applications are matrix operations like matrix addition (usually handled like the vector addition), matrix multiplication and convolution – operations that are very common in AI.

This vector processor requires understanding of the hardware platform, the algorithm and the parallelization of the data. At the design stage, the programmer needs to be aware of the dimensions of the vector registers; these limit how much data can be loaded into the register and processed in parallel.

The use of vector registers may imply a lot of changes in source code, and sometimes data organization in memory needs to be changed.

3.4.6.2 Purpose and Scope

Canonical vector processors help improve the speed; they have been proved to have great computation capabilities. There are still some limitations, e.g., the registers' fixed dimension and their small storage size. As a result, the conversion of the source code from a scalar CPU to a vector one requires some changes. Also, the traditional vector processors lack support for 2D vectors.

We propose an accelerator for matrix operations, with applications in AI. The accelerator is tightlycoupled to the CPU, and the new instructions will simplify coding and improve programming experience.

Our SIMD/Vector Accelerator supports operations such as matrix addition, subtraction, multiplication, element-by-element multiplication, element-by-element division and convolution. The accelerator features a reconfigurable register file and a simple programming interface. The user only specifies the operation and operands. The 2D vector accelerator can operate with a large set of data types, from different sized integer representations to floating point ones.

3.4.6.3 Place in the System

We want to place the accelerator very close to the core to achieve high speed communication. The accelerator is tightly coupled with the core and features a dedicated memory interface, see Figure 3.4.6.3-1.

The accelerator-core interface is the CoreV eXtension Interface (CV-X-IF). This interface allows us to extend RISC-V ISA and enable access to core registers.

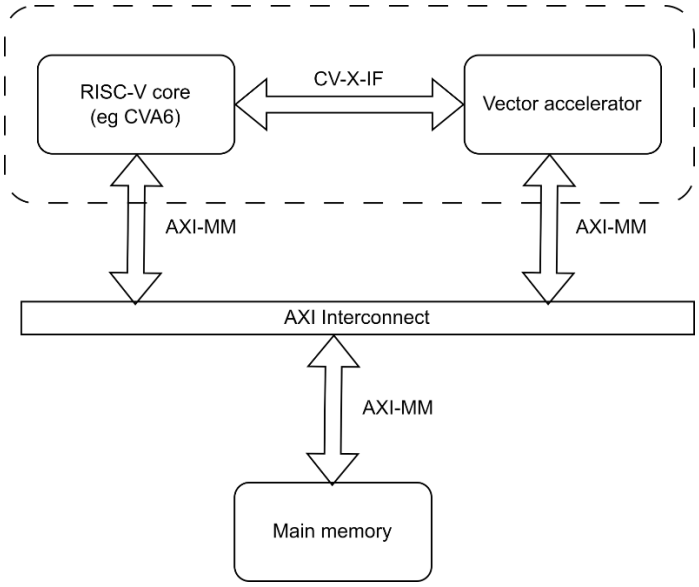


Figure 3.4.6.3-1: Vector-SIMD Accelerator - Place in the system

Because memory bandwidth may represent bottleneck for memory bound applications, a dedicated memory interface will be added to the vector accelerator. This interface relieves the main core from memory operation and allows the accelerator to work independently. Because the most popular interfaces in SoCs are from the AMBA family, the memory interface is an AXI-MM. The accelerator and the core use the same address space; both memory interfaces use the same interconnect.

3.4.6.4 Block Diagram

The internal architecture is presented in Figure 3.4.6.4-1. The main components of the accelerator are the Control Unit, the Register File and the Arithmetic Unit.

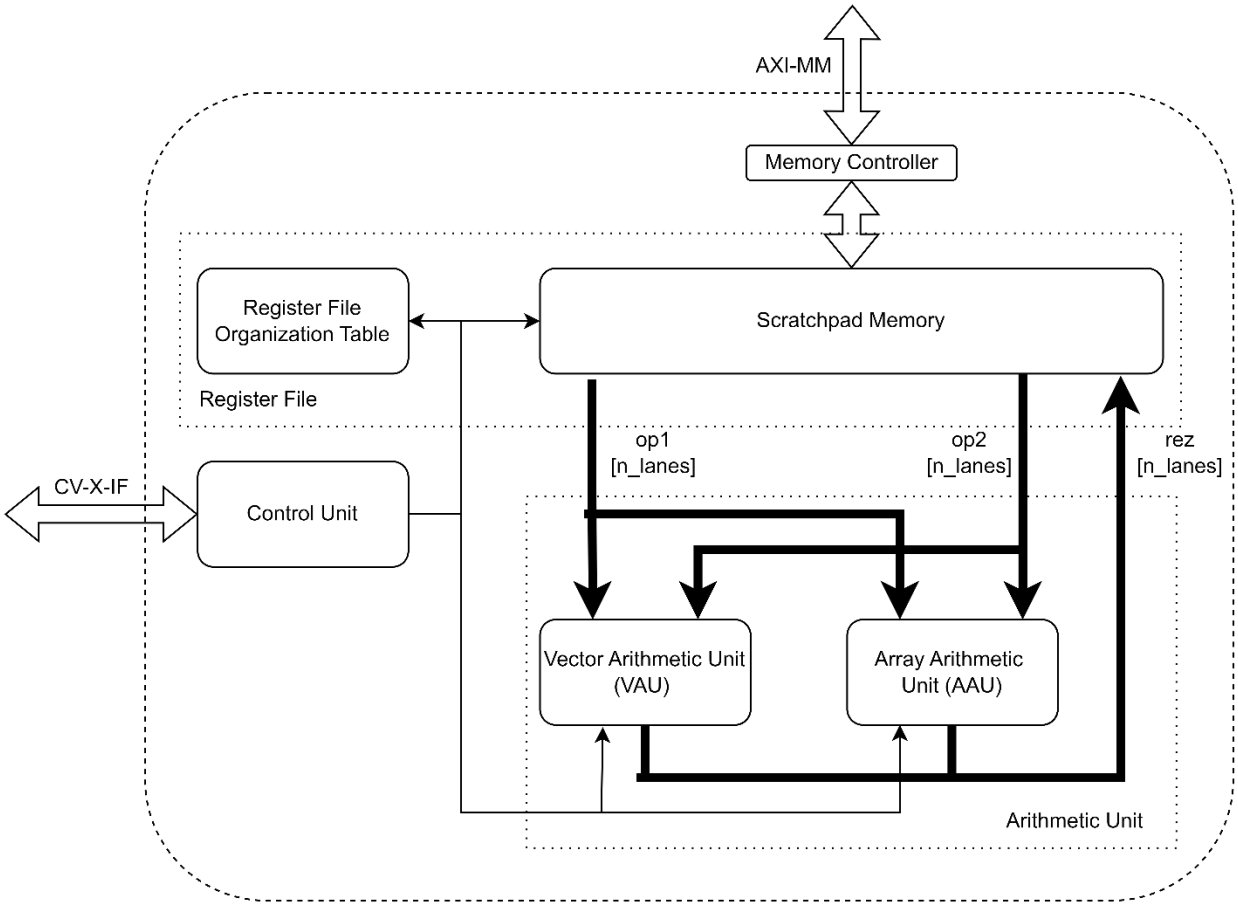


Figure 3.4.6.4-1: Vector-SIMD Accelerator - Internal architecture

The most important component is the Control Unit. This component gets instructions from the scalar CPU core and exchanges data using the core registers. An important feature for our SIMD/vector accelerator is the software defined 2D vector registers. Based on the operand and their sizes, the fastest track for the vector operation is decided.

The Register File is based on the Polymorphic Register File (PRF) and PolyMem [Ciobanu2013, Ciobanu2018], which represent the base for the IMT Scratchpad memory described in Section 3.2.3. The Register File Organization Table (RFOG) stores the defined registers, their size, data type and base address. This component has a dedicated connection to the main memory, to get the highest bandwidth for main memory related operations. The Register File communicates with the Arithmetic Unit via three data streams: two of them feed the input operands and the other one gets back the results. All streams have multiple lanes and send data in parallel.

Figure 3.4.6.4-2 presents an example of a software defined register table and data in memory. On the right side of the figure is presented the Register File organization table. It stores the specification of every register: width, height, start position, data type and whether the register is defined. On the left side there is a visual representation of that register.

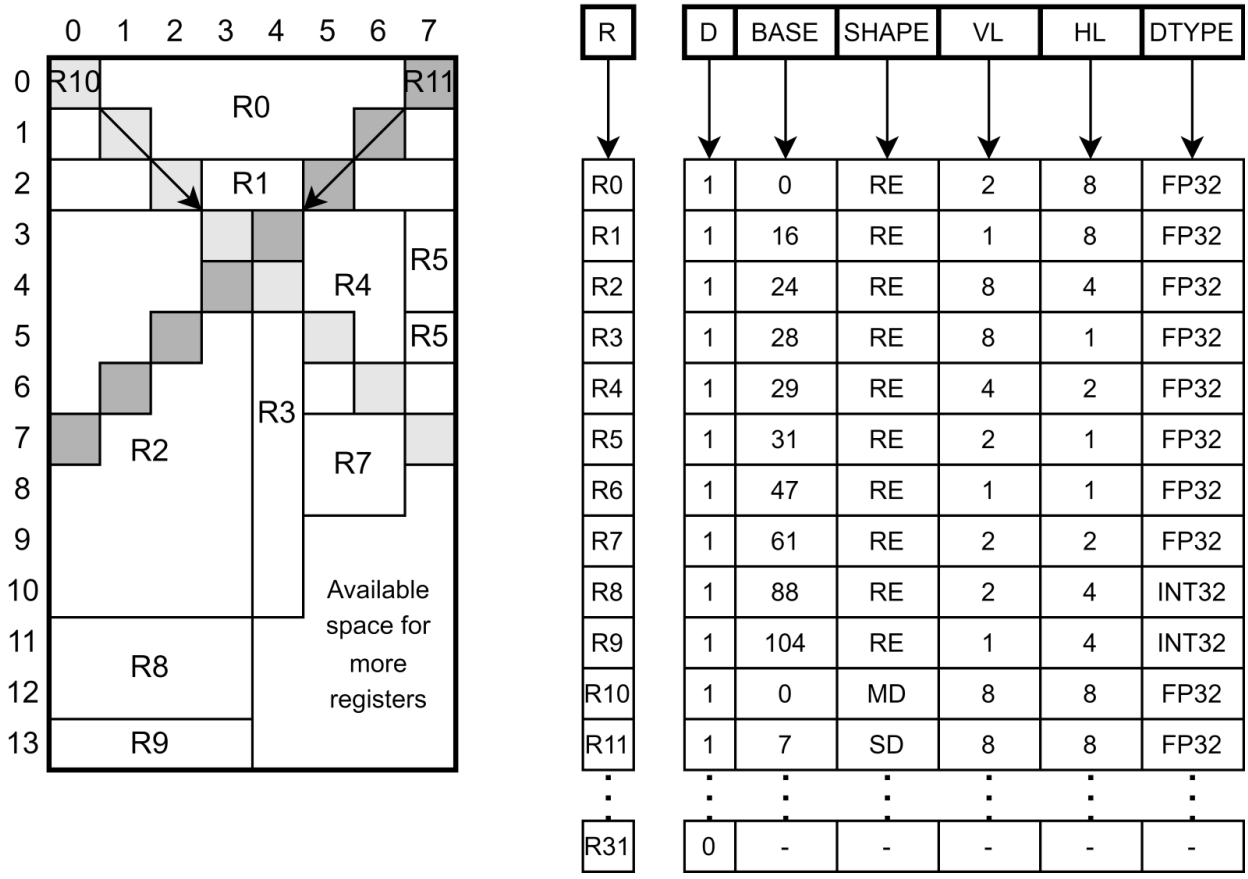


Figure 3.4.6.4-2: Vector-SIMD Accelerator - Example of software defined register in register file, based on [Ciobanu2013].

The Arithmetic Unit gets the data from the Register File, computes it and sends the results back. The data type is versatile and can operate with a large set of data types: integer and floating point on different bit lengths. The Arithmetic Unit has two types of circuits, one Array Arithmetic Unit for fast convolution and matrix multiplication, and another Vector Arithmetic Units, parallel arithmetic units for addition, subtraction and cross product.

3.4.6.5 ISA

The CV-X-IF interface sends only the invalid opcodes to the decoder. From the perspective of the main core this means that this custom instruction needs to have the seven less significant bits storing the opcode. Also, to gain access to core registers, custom instructions are required to strictly follow RISC-V instruction encoding. The source registers, `rs1` and `rs2` need to be in specific locations.

The RISC-V ISA was designed to be extended with custom opcodes, and for that purpose a dedicated range of opcodes for ISA extension is available. For convenience we chose the first value from that interval. For our accelerator the opcode value is configurable. After many iterations we managed to use only one opcode, and with the `funct3` field (see RISC-V base instruction formats) we identified the instruction format and the meaning of it. Because we use only one opcode, that also is configurable, it means that this accelerator is suitable to integrate with other accelerators that use CV-X-IF.

The ISA includes instructions to define 2D register, their sizes, data type and their base address, instructions to load and store data from register file, support for masking mode, synchronization instruction,

sectioning instructions and algebraic instructions. Accelerators allow vector-vector and vector-scalar operations. The supported operations are addition, subtraction, multiplication, element-by-element division, element-by-element multiplication and convolution.

Table 3.4.6.5-1 presents the instructions, their RISC-V encoding type, source of the registers and data for supplementary field (`funct3` and `funct7`). The `v2ddef` instruction defines a 2D vector, get width and height from core register, and on `func7` store the data type. There are two instructions for main memory access that have the same format as load and store in RISC-V ISA. The most important instruction types are `.vv` and `.vs`. These instructions handle mathematics operations and the difference between those two instructions is the `.vv` type works with two 2D vector and another one (`.vs`) works with one 2D vector and one scalar accessed from the scalar core register file. To fully define a register, three instructions are required: one to define register dimensions (`v2ddef`), one to define data type (`v2dtype`) and the last one to define base address (`v2dbase`). A dedicated instruction allows 2D register copy, the `v2dmov2d` instruction. There is also an instruction to support sectioning (`v2dsetvl`). Accelerators could work in masking mode; two instructions enable or disable this function (`maskon, maskoff`). To populate the masking array, two 2D accelerators registers can be compared element-by-element, or a 2D register can be filled with a scalar value. Our accelerator works in asynchronous mode, and with a dedicated instruction the user can synchronize the main core with the accelerator.

Detailed information about instructions is provided in Table 3.4.6.5-3. The arithmetic instructions are of type R and on `funct7` field stores the operation type. All supported operation and their encoding are provided in Table 3.4.6.5-2.

Mnemonic	**Type**	**Opcode**	**`rs1` place**	**`rs2` place**	**`rd` place**	**`funct3`**	**`funct7`**
`v2ddef`	R	0001011	core	core	acc	000	-
`v2dtype`	I	0001011	core	-	-	001	-
`v2dsetvl`	R	0001011	acc	acc	-	111	-
`v2dmov2d`	I	0001011	acc	-	acc	010	-
`accsetl.r`	I	0001011	core	-	-	100	
`accsetl.i`	I	0001011	core	-	-	101	-
`v2dsgt.vv`	R	0001011	acc	acc	-	011	-
`v2dsgt.vs`	R	0001011	core	acc	-	110	-
`v2dbase`	I	0001100	core	-	acc	000	-
`v2dld`	I	0001100	core	-	-	001	-
`v2dst`	I	0001100	core	-	-	010	-
`*.vv`	R	0001101	acc	acc	acc	000	See Table 3.4.6.5-2
`*.vs`	R	0001101	core	acc	acc	111	See Table 3.4.6.5-2
`sync`	I	0001110	-	-	-	000	-
`maskon`	I	0001110	-	-	-	011	-
`maskoff`	I	0001110	-	-	-	010	-

Table 3.4.6.5-1: Vector-SIMD Accelerator - Main instruction format and operand sources

Instruction name	Operation	funct7	Comment
vadd2d	A+B	0000001	
vsub2d	A-B	0000010	
vcnv2d	Convolution	0000100	rs2 as kernel matrix
vdiv2d	A/B	0001000	Element by element
vmul2d	A*B	0100000	Matrix multiplication
vsmul2d	A*B	1100000	Cross product, element by element multiplication

Table 3.4.6.5-2: Vector-SIMD Accelerator - Operations list

Mnemonic	Name	Description
v2ddef	Vectors define	This instruction defines a vector register, the output is a vector in accelerator. As input get width and height, from the core registers in rs1 and rs2. In the funct7 field is encoded the data type.
v2dld	Vector load	This instruction loads data from the main memory in the register file. The format is the RISC-V one. The start address in main memory is rs1 + immediate.
v2dst	Vector store	This instruction stores data from the register file in main memory. The format is the RISC-V one. The destination address in main memory is rs1 + immediate.
***.vv**	Vector-vector op	This instruction takes three accelerator registers, two as source and one as result. The supported matrix operations are addition, subtraction, multiplication (matrix-matrix and element-by-element), division (element-by-element) and convolution
***.vs**	Vector-scalar op	This instruction takes two vectors register and one core register, one vector register is the destination one and the other one is source for operation with scalar. The supported operations are addition, subtraction, multiplication and division.
v2dtype	Set data type	If data in rs1 is 0, then all register gets the same data type, else only the rd register get that type.
v2dsetvl	Set 2D vector length	This instruction is for sectioning. The rs1 register is for X direction and rs2 instruction is for Y direction.
v2dmov2d	Move 2D vector data	Copy register data from one to another. rs1 is source register, rd is destination register
accsetl.r	Set number of lanes	Set number of lanes, data from register, rs1 store this data.
accsetl.i	Set number of lanes by immediate	Set number of lanes, by immediate data.
v2dsgt.vv	Set greater than	Set 1 in masking array if rs1 < rs2. rs1 and rs2 are accelerator registers.

v2dsgt.vs	Set greater than	Set 1 in masking array if any element of `rs1` is less than `rs2`. `rs1` - 2d accelerator data, `rs2` - core scalar data.
v2dbase	Set register base address	Set accelerator register base address.
sync	Synchronization	By default, the accelerator works in asynchronous mode. After fetching an instruction, it will flag the instruction as done. This synchronization instruction stalls the scalar core until the accelerator instruction is completed and all data is written back to the main memory.
maskon	Start working on masking mode	Working with masking array.
maskoff	Stop working on masking mode	Working without masking array.

Table 3.4.6.5-3: Vector-SIMD Accelerator - Instruction with detailed explanations.

3.4.6.6 Interfaces

We decided to go with a tightly coupled accelerator. For the RISC-V cores, OpenHW group defined a dedicated accelerator interface. The interface is called CV-X-IF and it is very versatile and easy to extend. The scalar core sends to the accelerator only the core's invalid opcodes and the accelerator has access to the scalar core's registers.

The second interface we need is one for memory access, and there are multiple solutions. We decided to deliver a solution simple to integrate. We decided to use an AXI interface because in modern systems the AMBA buses are very commonly used.

CV-X-IF

CV-X-IF is an interface available on specific RISC-V cores. This interface was designed for tightly coupled accelerators. CV-X-IF uses a dedicated protocol to send the unknown opcodes to an external unit. Furthermore, this interface allows read and write access to the scalar core registers. Additionally, this interface includes a handshake method to notify the scalar core when the instruction is done.

Currently this interface is implemented in NOEL-V and CVA6 cores. With this interface, an accelerator could be coupled to a multicore microprocessor, because every core has a unique identifier associated.

AXI-MM

We wanted a common interface for memory access. The most used interfaces in modern SoCs come from the AMBA family (designed by ARM). It defines interfaces with different performance levels targeting various applications.

The AXI-MM is an AMBA interface designed for data transfer.

3.4.6.7 Clocking Strategy

The accelerator has two interfaces on two different buses. This implies the accelerator already has two clock domains. The internal logic will work at a different and higher clock speed. Inside the accelerator there are two components: the Register File and the Arithmetic Unit. These work at different speeds, so a design space exploration to accommodate that is to be employed.

Working on multiple clock domains requires extra logic. This is needed to prevent metastability and ensure data integrity. We opted for two solutions: asynchronous FIFO and clock crossing synchronizing. The asynchronous FIFO is used where there is a large and fast data exchange, like communication between

Register File and Arithmetic Unit. The clock crossing synchronization is used for control signals, because the changes are very slow. The Control Unit signals are synchronized with the destination components. A detailed clocking strategy and synchronization is presented in Figure 3.4.6.7-1.

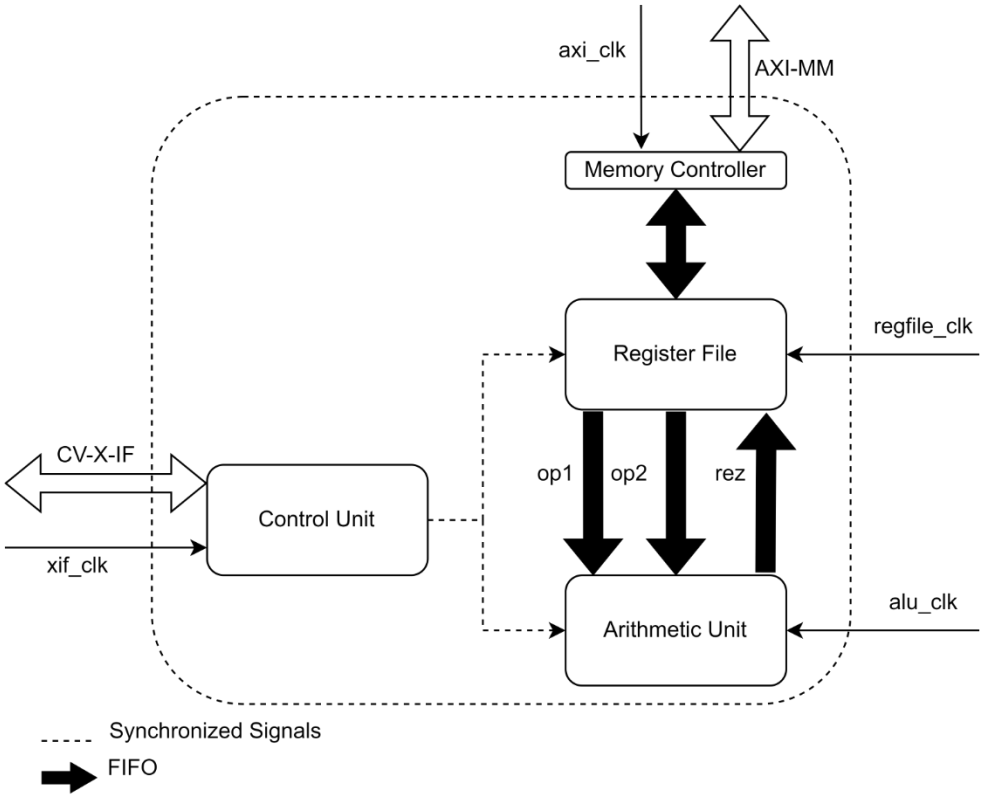


Figure 3.4.6.7-1: Vector-SIMD Accelerator - Clocking scheme

3.4.6.8 Reset Strategy

The accelerator will work with an asynchronous active in low reset. Both buses provide this type of reset. Because we have two resets, we will combine them with an AND gate to have a single reset signal for accelerators. For detailed reset scheme see Figure 3.4.6.8-1.

The reset will erase all software defined registers, reset the valid flag, and set all of the other fields to zero. Upon resetting of all the counters, the user configuration will be also set to zero. It will not clear the scratchpad memory.

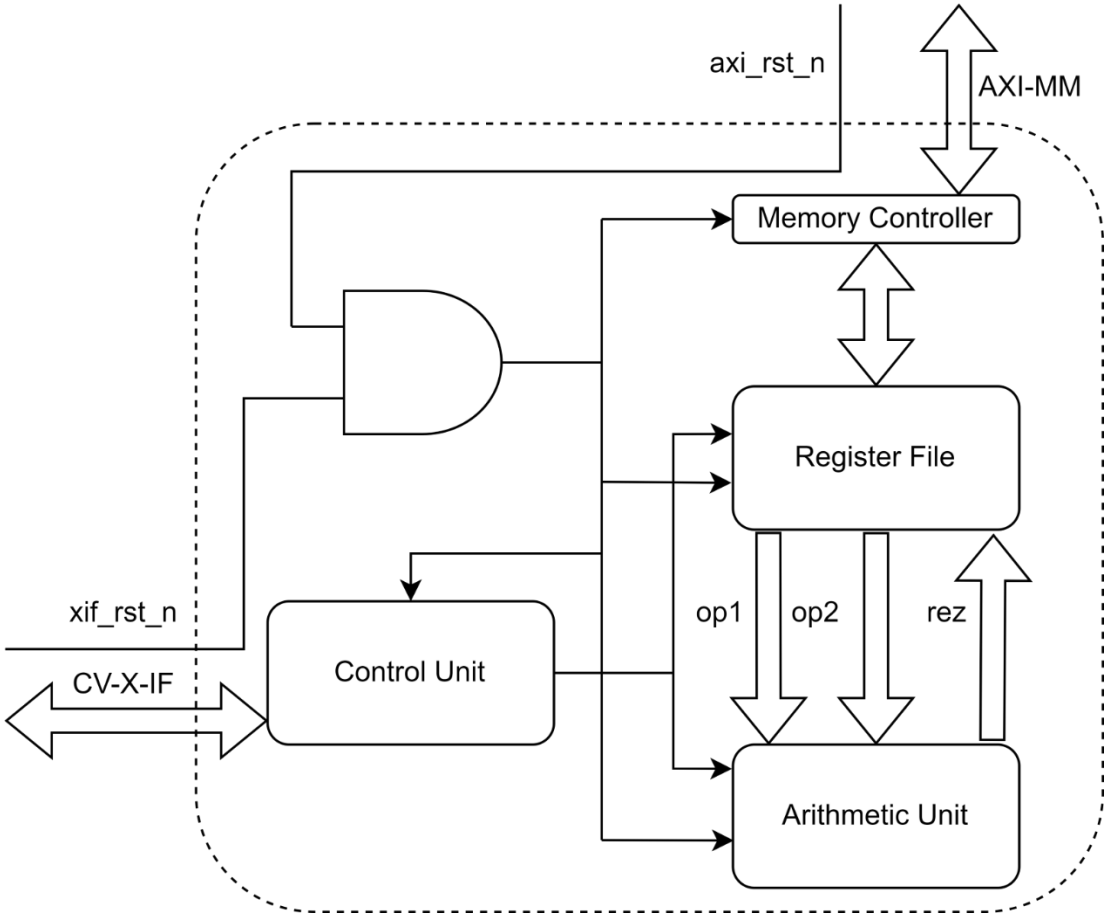


Figure 3.4.6.8-1: Vector-SIMD Accelerator - Reset scheme

3.4.7 Extension Platform (EXP) – TUI

Part of Task 3.4 SIMD/Vector, AI accelerator and tensor processor unit design.

3.4.7.1 General Information

The Extension Platform (EXP) is a component that can be instantiated in a RISC-V based SoC. The architecture of the EXP comprises data processing units, a memory and an interconnection network alongside the required control logic. A data processing unit is named Processing Engine (PE). It is formally defined as a digital design employed to accelerate computations often found in the digital signal processing domain. One such unit implements a part of the Coordinate Rotation Digital Computer (CORDIC) [Walther2000] dataflow. Another example is the computation of streamed Discrete Fourier Transform (DFT). These units can be viewed as operators processing the input data. The EXP architecture allows for composition of these operators to reduce latency.

3.4.7.2 Purpose and Scope

EXP's main purpose is to allow simultaneous processing of data words (SIMD) with a low latency. EP should mainly target digital signal processing specific computations in a streaming input/output fashion. A goal is to permit composition of computations to further contribute to CPU offloading.

3.4.7.3 Place in the System

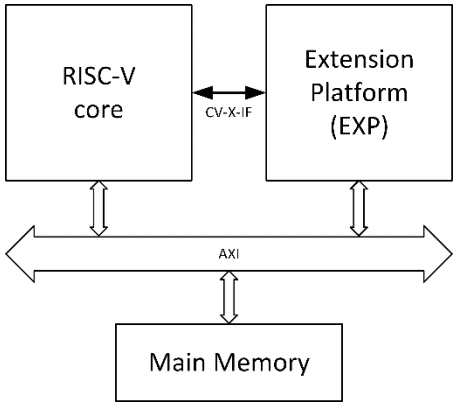


Figure 3.4.7.3-1: Extension Platform – Place in the system

EXP is presented in Figure 3.4.7.3-1. EXP is the generic name of the hardware module that incorporates computation specific VPUs. EXP is connected to the RISC-V core using a CV-X-IF interface. EXP has a high bandwidth link to the main memory for large size data transfers (blocks, vectors).

3.4.7.4 Block Diagram

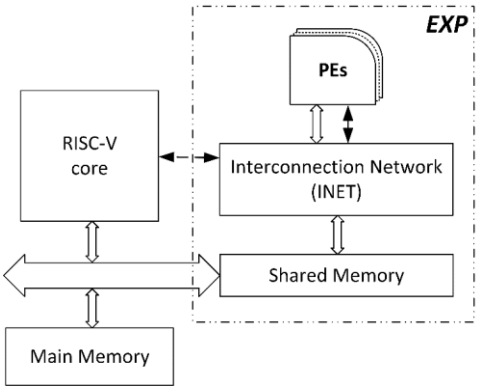


Figure 3.4.7.4-1: Extension Platform – Block diagram

EXP high-level block diagram is presented in Figure 3.4.7.4-1. It encompasses several PEs. Multiple PEs may be instantiated and connected through an interconnection network (INET) to scale performance as needed. PEs may be simple ALUs or more complex VPUs. PEs may be homogenous or heterogenous. External DMA modules move data to or from this shared memory. The proposed architecture provides composability and should facilitate design space exploration. The communication interfaces are standard interfaces such as CV-X-IF or AMBA family.

One possible PE operation mode is described next: PE starts execution immediately after input data is available in its own input FIFO. It will continue execution for at least the number of cycles required by its processing logic. The result of the processing flow will be stored in the PE's own output FIFO. This completes the data flow. PE is now ready for new data. If the own input FIFO is empty, PE execution will be suspended. Otherwise, PE will read the next input from its own input FIFO and process it. If its own output FIFO happens to be full, it will stall execution and will wait for clearance. An example of a dataflow chaining between PEs could be the computation of the complex DFT, which output will be transferred to the input FIFO of the PE implementing the `atan2` function. Another feature considered is the usage of tags to identify streamed data.

3.4.7.5 ISA

The number and format of the instructions needed to interact with EXP will be available later. These instructions, however, will implement the communication of information about the location of the input (and output, respectively) in the shared memory. Also required is the number of vectors to be processed in a pipelined fashion. These vectors represent the dataset on which PEs operate. Another piece of information is about the composition of PEs, composition order and, possibly, the number of iterations.

3.4.7.6 Interfaces

CV-X-IF and one or more from AMBA family.

3.4.7.7 Clocking Strategy

The EXP architecture incorporates a minimum of two clock domains, utilizing clock gating techniques. PEs can function at varying clock frequencies and feature clocks that are gated. Due to this design decision, there is a requirement for clock domain crossing modules to facilitate communication between certain components.

3.4.7.8 Power Management Strategy

The design adheres to the functionalities outlined in the clocking strategy section. These have a positive impact on power usage. The plan is to implement multiple power states, each offering different degrees of functionality within the design and associated benefits in power consumption.

3.4.7.9 Debugging Strategy

The debugging infrastructure needs to support inspection and modification of state in specific sections within PEs. The central debugging mechanisms are located within the INET module. All blocks that can be debugged are designed to react to control signals sent from the central debugging system. Scan chains could be integrated into these blocks.

3.5 Cryptographic and Security Accelerators

Task 3.5, M3-M33, Task Leader: SAL

The aim of this work package is the execution of cryptographic primitives using parametric hardware accelerators, with protection against side channel attacks enabled by design. These HW building blocks will be integrated with a RISC-V processor in an FPGA to support non-accelerated control operations. POLIMI is working on the integration of a hardware accelerator for the post-quantum key encapsulation mechanism (KEM) BIKE - the design will support key generation, encapsulation, and decapsulation primitives of the BIKE KEM - this design will be a part of the space demonstrator. BSC aims to integrate the Module-Lattice-Based Key-Encapsulation Mechanism (ML-KEM; CRYSTALS-Kyber), and the Module-Lattice-Based Digital Signature Algorithm (ML-DSA; CRYSTALS-Dilithium) into the NOEL-V platform, intended for the space and automotive demonstrators. IMT is developing optimized algorithms for NTT for polynomial multiplications used in Post-Quantum Cryptography (PQC) algorithms. This will be developed into a generic accelerator for polynomial multiplication based on the NTT in a large ring $Z_N[X]$, taking advantage of many roots of unity with trivial multiplication and of fast Fast Fourier Transform (FFT) algorithms (SIMD like vectorization for lengths 2a, 2a3 and 2a5). It is intended for the automotive and space demonstrators. SAL is working on the Classic McEliece implementation as a separate coprocessor, with the polynomial multiplication accelerated by an NTT block. The possibility of other PQC accelerators is also being explored, with the final use case in the automotive and possibly the space demonstrators.

Investigation of side channel and fault injection resistance of these systems will be achieved by comparing the HW accelerator implementations against a pure SW implementation at a later stage.

IP	Lead Beneficiary	Type	Domain	Dependencies	Licensing
ACC-BIKE	POLIMI	Core	PQC acceleration	AMBA AXI4	Proprietary
HLS-PQC	BSC	Core	PQC acceleration	Pulp Platform AXI	Permissive open source (SHL-0.51)
NTT	IMT	Algorithm	PQC acceleration	None	Restrictive open source (GPL-3.0)
PQC-MA	SAL	Core	PQC acceleration	CVA6, CV-X-IF	Open source
SEC	BEIA	Core	Processor with cryptographic accelerators	None	Permissive open source (CC-BY-4.0)

Table 3.5-1: Overview of contributions in Task 3.5

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

Part of Task 3.5 Cryptographic and security accelerators.

3.5.1.1 General Information

This module handles the integration and documentation of a hardware accelerator for the post-quantum KEM BIKE, which is a candidate in the PQC standardization process by National Institute for Standards and Technology (NIST), USA. The NIST competition aims to design crypto schemes that can be executed on traditional computers and are secure against both traditional and quantum attacks. BIKE is a code-based KEM that makes use of quasi-cyclic moderate-density parity-check (QC-MDPC) codes [Baldi2014]. Such QC-MDPC codes are employed in a scheme similar to the well-studied Neiderreiter cryptosystem, which dates to the early 1980s [Niederreiter1986]. 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.

3.5.1.2 Purpose and Scope

This accelerator aims to provide hardware support for the key generation, encapsulation, and decapsulation primitives of the BIKE KEM. It is designed to be integrated in platforms making use of an AXI interface.

3.5.1.3 Place in the System

The accelerators for the BIKE cryptosystem can be interfaced with the RISC-V CVA6 or NOEL-V cores through the AXI interface they expose.

3.5.1.4 Block Diagram

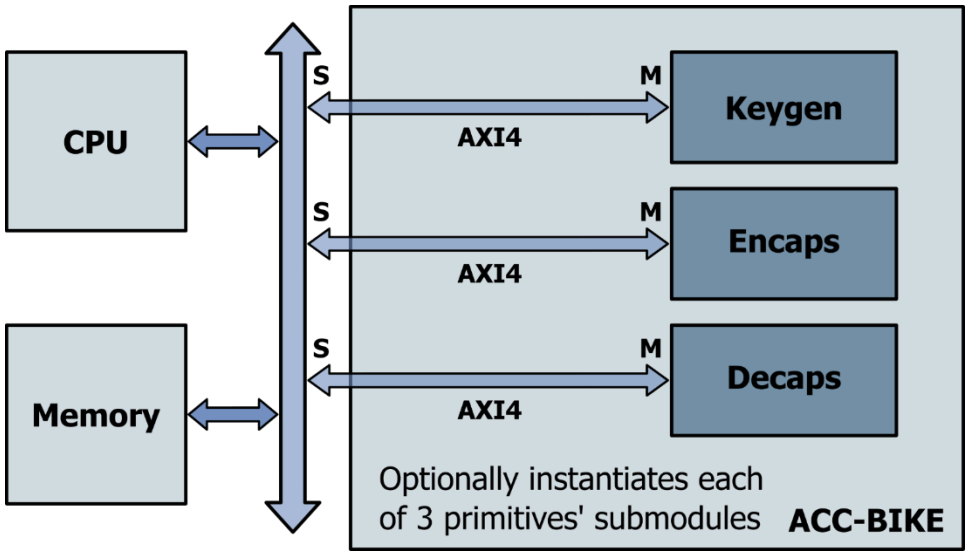


Figure 3.5.1.4-1: ACC-BIKE - Block diagram

The accelerator provides separate support for the three primitives, each of whom can be optionally instantiated in hardware. The hardware acceleration for the BIKE KEM can be integrated into computing platforms by making use of the AXI interface exposed by the primitives' submodules as shown in Figure 3.5.1.4-1. Due to its QC-MDPC code-based nature, BIKE makes use of binary polynomial and QC-MDPC codes arithmetic. In particular, the most computationally intensive operations are binary polynomial inversions (in the key generation primitive), the Black-Gray-Flip variant of QC-MDPC bit-flipping decoding (in decapsulation), and binary polynomial multiplication (in all three primitives).

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

Part of Task 3.5 Cryptographic and security accelerators.

3.5.2.1 General Information

Communication security is one of the most important characteristics of a system. In recent years, researchers have discovered significant weaknesses in current public-key cryptographic algorithms using quantum computing. As a result, organizations such as NIST are standardizing several quantum-resistant algorithms. Therefore, we have decided to integrate an accelerator based on High Level Synthesis (HLS) into the SELENE SoC (NOEL-V core), implementing the Key Encapsulation Mechanism ML-KEM (FIPS-203) and the Digital Signature Scheme ML-DSA (FIPS-204). Both are based on CRYSTALS-Kyber and CRYSTALS-Dilithium schemes respectively.

3.5.2.2 Purpose and Scope

Our developments focus on optimizing the PQC standardizations. One way to improve the performance is to use PQC specific accelerators instead of executing PQC functions in a general-purpose processor. Our acceleration technique is based on high-level synthesis. By using high-level software language extensions, tools can interpret software code to generate HDL. This allows, in an easier way, the translation of a software application/algorithm to hardware description.

NIST purposes to standardize one KEM and two Digital Signature Schemes (DSS). The KEM is the ML-KEM. The DSS are the ML-DSA and Stateless Hash-Based Digital Signature Algorithm (SLH-DSA). For this project, we decided to integrate an HLS-based accelerator for the ML-KEM and the ML-DSA, as they share similar computation modules.

3.5.2.3 Place in the System

The modules are connected to the core via the NOC with the AXI4 protocol, as shown in Figure 3.5.2.3-1.

The AXI Lite connects the core with the accelerator to select the memory directions of the data and the control signals.

Each accelerator is connected using a Network on Chip (NoC), adhering to the AXI-Full protocol. For every data argument, there exists a dedicated AXI-Full data bus that facilitates data transmission to the memory.

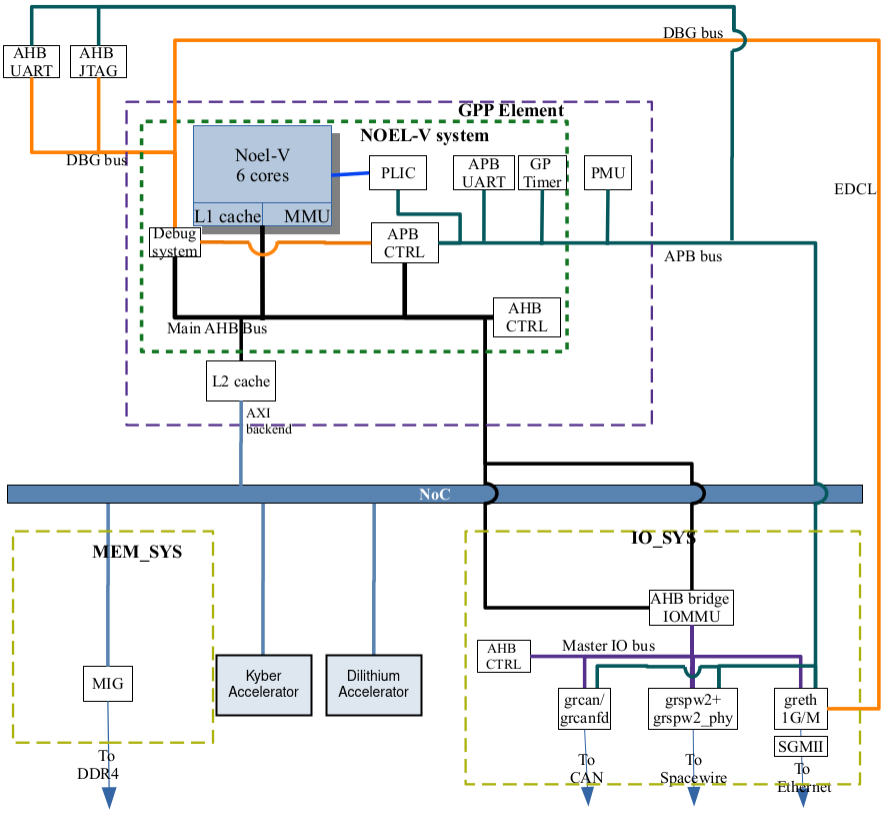


Figure 3.5.2.3-1: HLS-PQC - Place in the system

3.5.2.4 Block Diagram

The algorithms are split into modules depending on their functionality in such a way that we maximize the parallelization so that, by pipelining the algorithm, its different parts can be performed in parallel. Each module contains a self-descriptive name for the task it performs, as shown in Figures 3.5.2.4-1, 3.5.2.4-2, and 3.5.2.4-3.

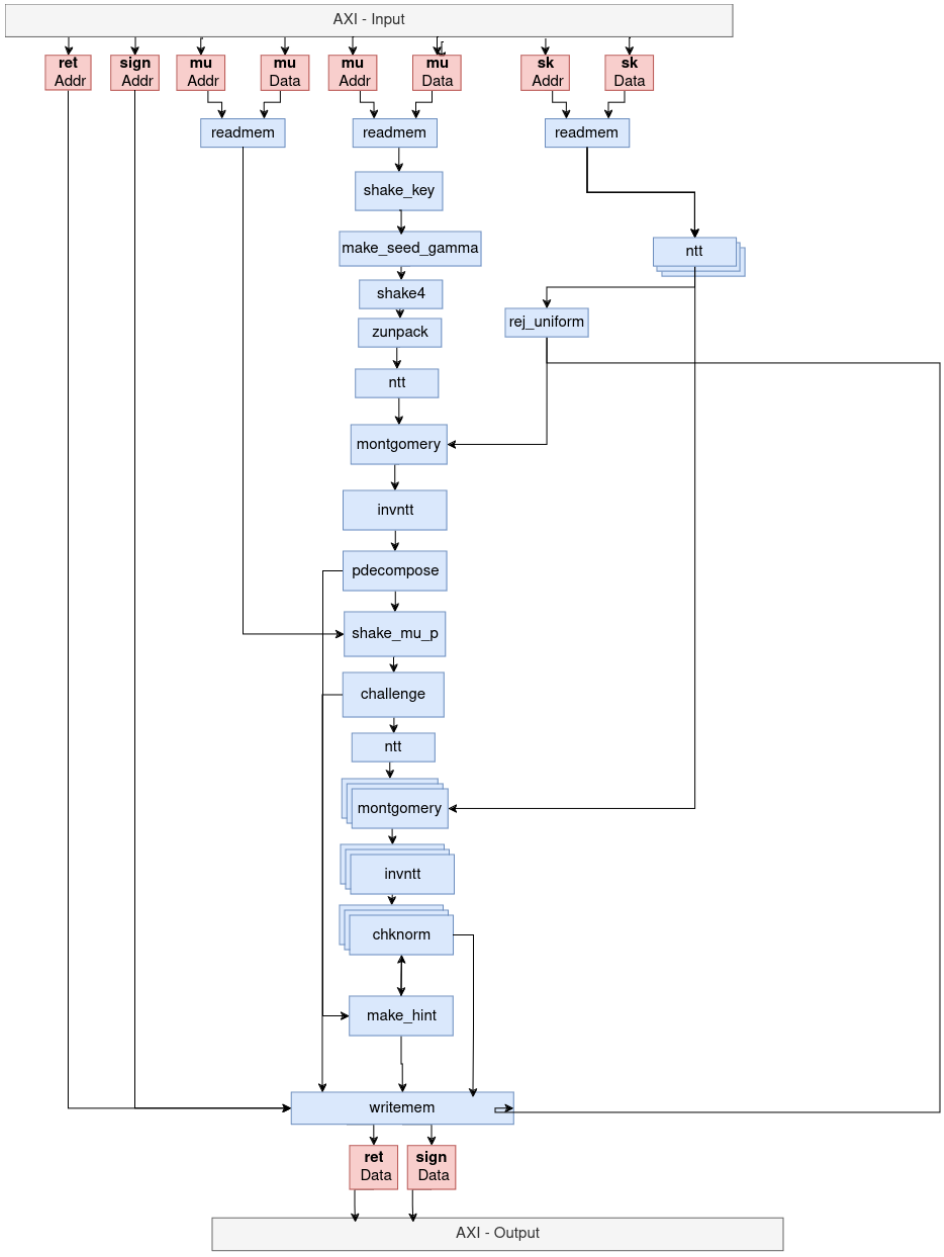


Figure 3.5.2.4-1: HLS-PQC - ML-DSA Sign

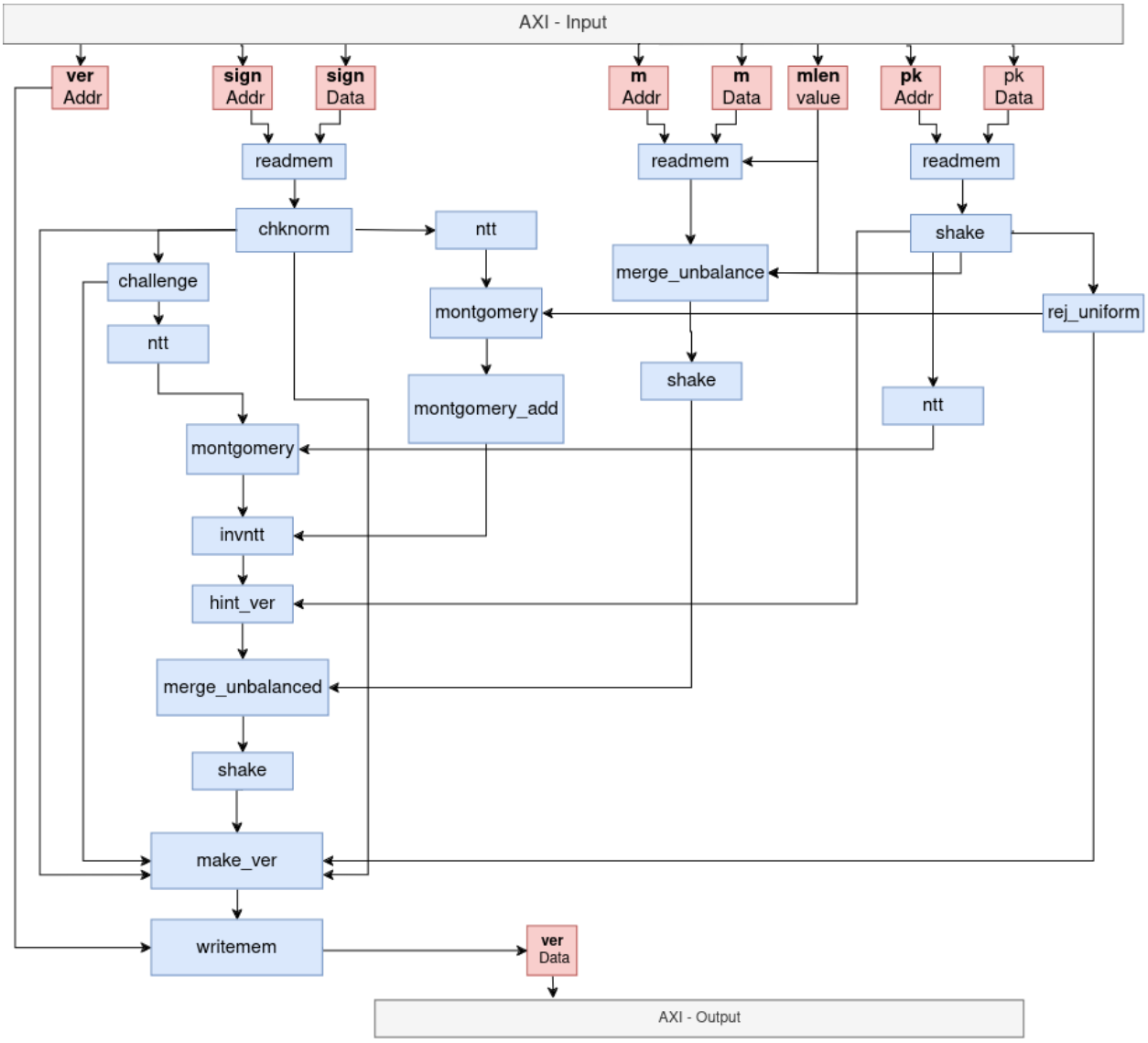


Figure 3.5.2.4-2: HLS-PQC - ML-DSA Verify

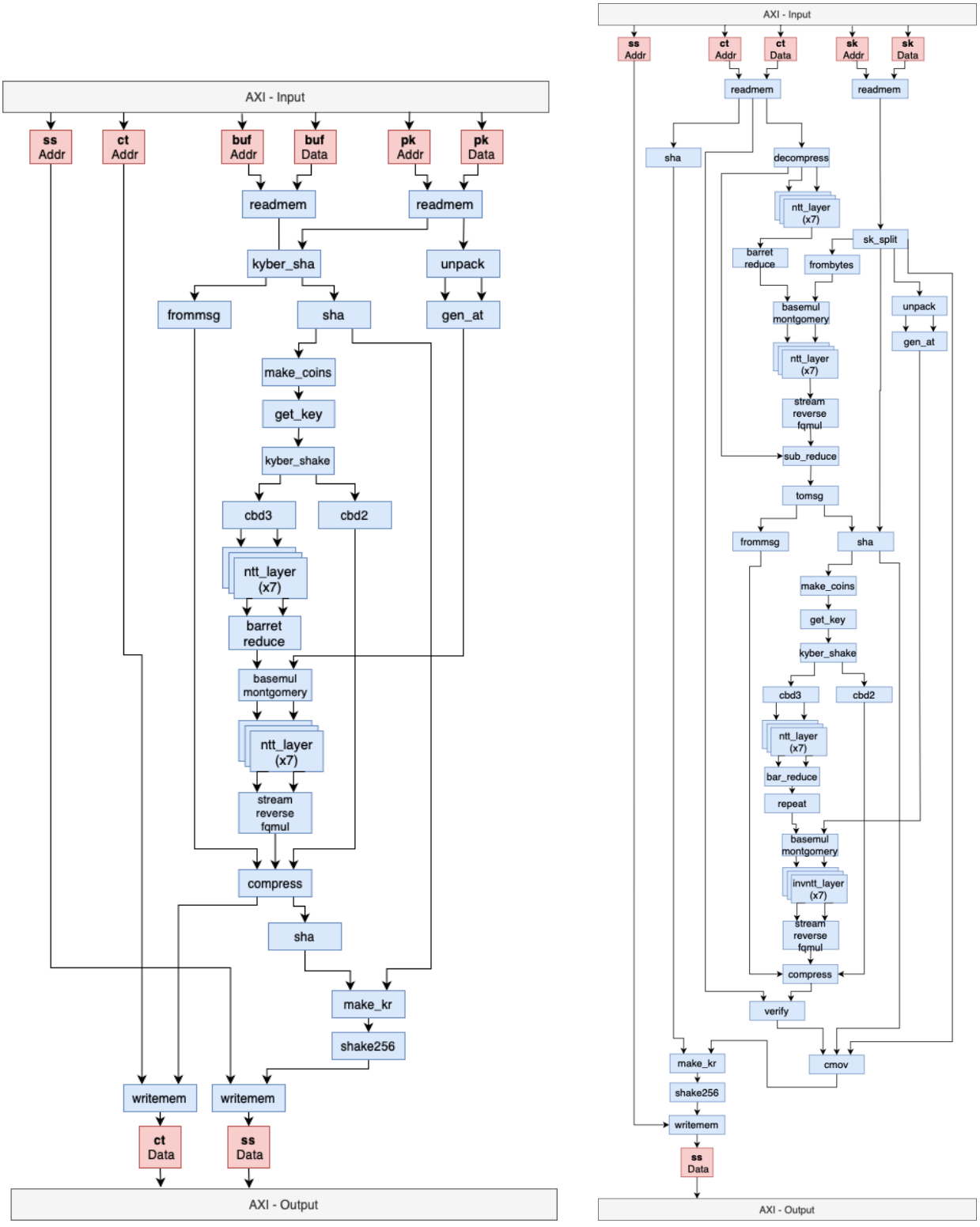


Figure 3.5.2.4-3: HLS-PQC - ML-KEM Encapsulation (left) and Decapsulation (right)

3.5.2.5 Interfaces

In this interface communication the processor is the Master, and the accelerator is the Slave.

ML-KEM Interface

In the first table below (Table 3.5.2.5-1), we can see all the configuration registers They are divided into control (`CTRL`), interruption management (`GIER`, `IER`, `ISR`), the operation selector (`kem_cfg`), and the base memory address for each I/O data port. All these configuration registers must be set by the processor before starting a new operation (except for the interrupt status register `ISR`). In addition, we have 5 I/O data buses of 32-bit data width, managed by the AXI4-Full protocol. In the second table (Table 3.5.2.5-2), we can see these AXI interfaces, where in this case, the accelerator acts as a Master. Some buses (i.e, `gmemct` and `gmemss`) are used by both dataflows (Encapsulation and Decapsulation).

Register	Interface	Offset	Width	Access	Description
CTRL	s_axi_control	0x00	32	RW	Control signals (bit[0]=1 starts computation)
GIER	s_axi_control	0x04	32	RW	Global Interrupt Enable Register
IER	s_axi_control	0x08	32	RW	IP Interrupt Enable Register
ISR	s_axi_control	0x0C	32	RW	IP Interrupt Status Register
kem_cfg	s_axi_control	0x10	64	W	Operation Selection (Enc=0 Dec=1)
ct_in	s_axi_control	0x18	64	W	Memory Address of data signal ct_in
ct_out	s_axi_control	0x20	64	W	Memory Address of data signal ct_out
ss_enc_out	s_axi_control	0x28	64	W	Memory Address of data signal ss_enc_out
ss_dec_out	s_axi_control	0x30	64	W	Memory Address of data signal ss_dec_out
buf_in	s_axi_control	0x38	64	W	Memory Address of data signal buf_in
pk_in	s_axi_control	0x40	64	W	Memory Address of data signal pk_in
sk_in	s_axi_control	0x48	64	W	Memory Address of data signal sk_in

Table 3.5.2.5-1: HLS-PQC - Configuration registers managed by `s_axi_control` (AXI-Lite Interface)

Interface	Argument	Addr Width	Data Width	Access	Description
m_axi_gmembuf	buf	32	32	R	Bit Stream Randomizer Buffer
m_axi_gmemct	ct	32	32	RW	CipherText
m_axi_gmempk	pk	32	32	R	Public Key
m_axi_gmemsk	sk	32	32	R	Secret Key
m_axi_gmemss	ss	32	32	W	Shared Secret

Table 3.5.2.5-2: HLS-PQC - AXI-Full Data Interface

ML-DSA Interface

In the first table below (Table 3.5.2.5-3) we can see all the configuration registers, which are divided into control (`CTRL`), interruption management (`GIER`, `IER`, `ISR`), operation selection (`kem_cfg`), and the base memory address for each I/O data port. All these configuration registers must be set by the processor before starting a new operation (except for the interrupt status register `ISR`). In addition, we have 5 I/O data buses of 32-bit data width, managed by the AXI4-Full protocol. In the second table (Table 3.5.2.5-4), we can see these AXI interfaces, where in this case, the accelerator acts as a Master. Some buses (i.e, `gmemout`) are used by both dataflows (Signature and Verification).

Register	Interface	Offset	Width	Access	Description
CTRL	s_axi_control	0x00	32	RW	Control signals (bit[0]=1 starts computation)
GIER	s_axi_control	0x04	32	RW	Global Interrupt Enable Register
IER	s_axi_control	0x08	32	RW	IP Interrupt Enable Register
ISR	s_axi_control	0x0C	32	RW	IP Interrupt Status Register
kem_cfg	s_axi_control	0x10	64	W	Operation Selection (Sign=0 Verify=1)
ret_out	s_axi_control	0x18	64	W	Memory Address of data signal ret_out
sign_out	s_axi_control	0x20	64	W	Memory Address of data signal sign_out
sign_in	s_axi_control	0x28	64	W	Memory Address of data signal sign_in
mu_in	s_axi_control	0x30	64	W	Memory Address of data signal mu_in
m_in	s_axi_control	0x38	64	W	Memory Address of data signal m_in
mu2_in	s_axi_control	0x40	64	W	Memory Address of data signal mu2_in
sk_in	s_axi_control	0x44	64	W	Memory Address of data signal sk_in
pk_in	s_axi_control	0x48	64	W	Memory Address of data signal pk_in
ver_out	s_axi_control	0x58	64	W	Memory Address of data signal ver_out
mlen_in	s_axi_control	0x60	64	W	Memory Address of data signal mlen_in

Table 3.5.2.5-3: HLS-PQC - Configuration registers managed by `s_axi_control` (AXI-Lite Interface)

Interface	Argument Verify sign	Addr Width	Data Width	Access	Description
m_axi_gmemout	ver_out ret_out	32	32	W	Checks Marking
m_axi_gmemsign	Sign_in Sign_out	32	32	RW	Signature
m_axi_gmemm	mu_orig_In mu_processed_in	32	32	R	Original message Processed message = CRH(H(rho, pk), mu_orig_in)
m_axi_gmempk	Pk_in mu2_processed_In	32	32	R	Publick key Processed message = CRH(H(rho, pk), mu_orig_in)
m_axi_gmemsk	sk_in	32	32	R	Secret Key

Table 3.5.2.5-4: HLS-PQC - AXI-Full Data Interface

3.5.2.6 Clocking Strategy

The accelerator contains only one clock, which is provided by the system clock. In our experiments, the system clock reaches a frequency of 100 MHz. However, the accelerator itself can reach up to 500 MHz.

3.5.2.7 Reset Strategy

The PQC accelerator integrates only one reset method, the hardware active low reset signal through the input port `ap_rst_n`. Hardware reset completely wipes all data from both ML-KEM and ML-DSA accelerators, resetting the module to a blank state and interrupting any on-going transaction.

3.5.2.8 Verification Strategy

To verify the design, we have designed two C tests for each scheme (encapsulation, decapsulation, signature and verify) to be executed by the core in a "baremetal" way. Thus, to see the results of these tests, we can simulate the SoC in a software simulator (Xcelium [33]) or in an FPGA.

Regarding the implementation of the tests, first we obtained the inputs and outputs of the original Kyber/Dilithium algorithm (executing it in an x86 machine), and we incorporated them in a header file for our test. Thus, by including the same inputs to the accelerator, it should give the same outputs as the extracted ones. In summary, the methodology followed by the tests is the following:

1. Write to the `kem_cfg` accelerator register to choose the functionality of the accelerator (For ML-KEM: Encapsulation=0, Decapsulation=1. For ML-DSA: 0=Signature, 1=Verify).
2. Load the input/output vector addresses (the ones we have as global on the header file) to the correspondent configuration registers of the accelerator.
3. Raise the start flag (write 1 to `CTRL` configuration register) and enable interruptions (`IER=1` and `GIER=0`).
4. Wait for the accelerator results, performing a busy-wait polling. This means reading the ISR configuration register in a loop until it returns 1.
5. Read the results from global memory and compare them with the golden references extracted from the x86 execution.
6. Go back to step 1 for another try, where the inputs can be changed.

To extract performance results, the SELENE SoC contains a PMU. With some directive calls (i.e., reset, start, stop), we can extract the cycles taken from a part of the code. In our case, the part of interest is the waiting poll (step 4), which reflects the time it takes for the accelerator to perform the entire computation.

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

Part of Task 3.5 Cryptographic and security accelerators.

3.5.3.1 General Information

Post quantum cryptography and NTT

PQC aims to replace classical public-key cryptography with new schemes that are robust against attacks using quantum computers. Two of the most widely used public-key cryptographic algorithms: elliptic curve cryptography (ECC) based on the discrete logarithm problem, and Rivest–Shamir–Adleman (RSA) based on the prime factorization problem, are both prone to be solved in polynomial time by Shor's quantum algorithm.

During the fourth round of its standardization process for PQC, NIST has selected several algorithms (based on lattice, hash and code schemes) for public-key encryption and digital signature. The analysis done in a recent work building a crypto-processor for RISC-V [Lee2023], has identified the multiplication of polynomials over finite rings $Z_q[X]/P(x)$ as a possible target for accelerating many PQC computations (other identified targets are hash functions, modular multiplication and modular reduction and sampling from a given probability distribution).

There are many algorithms to accelerate the multiplication of two polynomials, for example Karatsuba and Toom-Cook multiplication [Bernstein2001]. Polynomial multiplication is equivalent to the convolution of two vectors (containing the polynomial coefficients), and the best asymptotical algorithm for large polynomial degrees is a special case of the Fast Fourier Transform – adapted to finite rings – called the Number Theoretic Transforms (NTT). However, not all choices of q and P(x) are compatible with the NTT. In these cases, the usual solution is to use Karatsuba multiplication combined with various tricks specific to each case to accelerate the calculations. A recent review is in [Liang2022].

We note that all lattice based PQC algorithms selected by NIST are NTT friendly, being all built around choices of the finite polynomial rings that admit NTT for powers of 2 and giving a tremendous speed advantage compared to other algorithms from the previous rounds. However, there were concerns among some crypto experts that the additional algebraic structure needed to be NTT friendly increases the surface attack of these algorithms, even if today there is no known attack using this.

As a cautionary tale, one third round finalist, SIKE, appears on the NIST website with the comment: "The SIKE teams acknowledges that SIKE and SIDH are insecure and should not be used", due to the recent (2022) discovery of an efficient and practical key recovery algorithm [Castryck2022]. This shows that there is a risk that algorithms already chosen for standardization may prove in the future insecure. It is important then to have flexible implementations, especially hardware ones, which are not tied to the particularities of one PQC algorithm, making it easy to switch to some other algorithm as need arises.

It seems therefore reasonable to develop an "universal" algorithm based on NTT for the product of two polynomials in finite rings, without restrictions on the modulus and polynomial degree and then provide a generic NTT accelerator, useful for all cases, including PQC algorithms with that are NTT friendly.

Polynomial products in $Z_q[X]/P(x)$ via NTT

A post-quantum cryptography module will need to provide the product of two polynomials in finite rings $Z_q[X]/P(x)$, one of the most time-demanding operations. Not every choice of q and P(x) permits the use of NTTs, a special case of the FFT to accelerate this calculation. Therefore, at first view, the utility of an NTT hardware accelerator seems limited. However, a similar problem in the DSP world (how to use the power-of-two FFT for other lengths) has a simple solution – extend the original vectors by zeros up to the next power of two, calculate the cyclic convolution with power of 2 FFTs, reinterpret the result as the linear

convolution of the original vectors, and finally wrap this linear convolution to obtain the sought-after cyclic convolution.

Adapting this idea to the product of two polynomials in $Z_q[X]/P(x)$, we need to calculate their product in $Z_q[X]$, which is equivalent to the linear convolution, and then reduce it modulo P(x), equivalent to a wrapping around P(x). To calculate the linear convolution, we can extend the vectors by zeros up to a length N power of 2 as above and calculate their cyclic convolution in $Z_q[X]/(X^N-1)$. However, a new problem appears as Z_q does not in general have roots of unity of order N like in the real/complex case (we need N to be a divisor of q-1 to have the required roots of unity). The solution is to work in another finite ring Z_p that admits roots of unity of order N. To be able to recover the original residues modulo q, we need this p to be so large that the result in $Z_p[X]/(X^N-1)$ is the same as if the cyclic convolution would have been done in $Z[X]/(X^N-1)$. Then we can easily do a reduction modulo q as the last step. If n is the degree of P(x), then the above condition on p becomes:

$$p > n\,(q-1)^2 \qquad (3.5.3.1\text{-}1)$$

We now propose two choices for p, giving two "universal" algorithms that use NTT for powers of 2 to calculate the product of two polynomials in $Z_q[X]$ for arbitrary choices of q and degree n of the polynomials, with the only constraint being Equation 3.5.3.1-1.

Algorithm 3.5.3.1-1: NTT based on a single large prime (NTT_LARGE_PRIME)

Our choice for p is the prime number:

$$p = 2^{64} - 2^{32} + 1, \qquad p-1 = 2^{32} \times 3 \times 5 \times 17 \times 257 \times 65537$$

Therefore, we have roots of unity of order powers of 2 up to 2^{32} in Z_p. With this choice, we can treat all PQC choices for modulus/polynomial degrees in the NIST submissions. For example, we can cover polynomial products for all 16-bit integers q and degrees up to 2^{31} ($q < 2^{16}$, $n < 2^{31}$). Or we can cover all 24-bit integers q and degrees up to 2^{15} ($q < 2^{24}$, $n < 2^{15}$).

When doing arithmetic in Z_p, we may use Montgomery modular multiplication with auxiliary modulus $R= 2^{64}$, which implies two supplementary multiplications with p and p'=(p-2) such that pp'= -1 mod(R). For the hardware implementation, these multiplications with p and p' can be replaced each with two adds/subtractions and two shifts. Another option that takes advantage of p being a Solinas prime (or generalized Mersenne prime), replaces the two multiplications needed for the reduction modulo p with only one addition and two subtractions of 64-bit integers. This second option is especially suitable for hardware implementation.

Another advantage is that 2 is a root of unity of order 192 = 3 x 64, so that multiplication by roots of unity of order 64 are actually trivial shifts. This is similar to multiplication by j and –j being trivial for real/complex FFT and can be used to further accelerate the NTT. We note that also multiplication by the square root of 2 is almost trivial, being only 2 shifts and an addition, and therefore all multiplications by roots of unity of order 128 are trivial (no multiplication involved).

The only disadvantage of this choice is that we need to work internally with 64-bit integers, which is not always suitable for embedded applications, and we need the full 128-bit results of multiplying two 64-bit integers.

Algorithm 3.5.3.1-2: NTT based on several small primes combined using the Chinese Remainder Theorem (NTT_CRT)

This algorithm reduces the burden of using 64 bits integers for doing polynomial multiplication in $Z_q[X]/(P(X))$ for relatively small values of the modulus q and degree n. It does the polynomial multiplication in $Z_{pi}[X]$ for several "small" primes p_i and then combines the results via the Chinese Remainder Theorem (CRT) to obtain the product polynomial in $Z_p[X]$ where $p = p_1\,p_2\,p_3$ ... is the product of these primes. Note that we still need to respect Equation 3.5.3.1-1. For example, the choice:

$$p_1 = 12289 = 2^{14} - 2^{12} + 1,$$

$p_2 = 40961 = 2^{15} + 2^{13} + 1$,

$p_3 = 61441 = 2^{16} - 2^{12} + 1$,

all with auxiliary modulus $R=2^{16}$ and each having roots of unity for powers of 2 up to 2^{12} permits the multiplication of two polynomials of degree up to 2048 in $Z_q[X]$ with $q < 2^{16} = 65536$, and all operations done on 16-bit integers ($q < 2^{16}$, $n < 2^{11}$).

The disadvantage of this method is that we need to calculate several polynomial products and their CRT combination, but this could be parallelized in a hardware implementation.

Finally note that this method can be easily adapted to larger values of q and n by choosing larger primes p_i and also a larger auxiliary modulus R, for example $R=2^{32}$, using operations on 32-bit integers.

Acceleration of NTT

We have seen above that we need to calculate the *cyclic convolution* cyc_conv(a,b) of two polynomials a,b in $Z_p[X]/(X^N-1)$, with p a prime number and N a power of 2 that is a divisor of p-1. This cyclic convolution can be done via algorithms similar to the FFT, by using NTT and its inverse NTT^{-1}:

$$\text{cyc\_conv}(a,b) = NTT^{-1}(NTT(a).*NTT(b)), \qquad (3.5.3.1\text{-}2)$$

where ".*" is pointwise multiplication similar to MATLAB notation and where the complex roots of unity of order N are replaced with roots of unity of order N in Z_p.

The same algorithms NTT, NTT^{-1}, and a pointwise multiplication like in Equation 3.5.3.1-2 but with different roots of unity can be used to calculate the *Nega cyclic convolution* negcyc_conv(a,b) of two polynomials a,b in $Z_p[X]/(X^N+1)$, which is widely used in PQC algorithms that are NTT friendly. However, a subtle difference is that some of these algorithms use NTT even in the case where some roots of unity are still missing. For example, in Kyber with p = 3329 ($p-1 = 2^8 \times 13$) and the ring $Z_{3329}[X]/(X^{256}+1)$, there are roots of unity of order 256 but none of order 512, needed as $(X^{512} - 1) = (X^{256} - 1)(X^{256} + 1)$.

In this case, the NTT transform of a vector a, NTT(a), does not represent 256 numbers in Z_{3329} but 128 polynomials of degree 1 and with coefficients in Z_{3329}, and the pointwise multiplication in Equation 3.5.3.1-2 must be replaced with a modular multiplication of these degree-one polynomials. To cover this case, our NTT module must be able to stop at some predefined level.

The number of operations (additions and multiplications) for NTT is proportional to $N \log_2 N$, making Algorithm 3.5.3.1-2 the fastest algorithm, at least asymptotically. However, for a practical implementation, the constant that multiplies the $N \log_2 N$ term, the capability to vectorize / parallelize the chosen algorithm, and many other factors contribute to its speed. By using the same techniques as for the FFT accelerator in Task 3.6 (Section 3.6.1), we hope to obtain a generic NTT accelerator able to compete with state-of-the-art algorithms for NTT friendly choices, and useful for most PQC schemes, including those without an algebraic structure permitting the use of NTT.

3.5.3.2 Purpose and Scope

We have shown in the introduction that the power-of-two Number Theoretic Transform (NTT) may be used in most PQC schemes to calculate the product of two polynomials over finite rings $Z_q[X]/(P(X))$ for arbitrary q and degree n of P(X).

For the first algorithm NTT_LARGE_PRIME, we propose a radix-128 NTT that reduces drastically the number of multiplications by taking advantage of the fact that roots of unity of degree 128 are either powers of 2 or a sum of two powers of 2, such that multiplication with these roots becomes trivial (equivalent to shifts and additions). We also show that the algorithm can be easily vectorized for short vectors, typical for SIMD, with vector lengths V=2,4,8,16.

For the second algorithm, NTT_CRT, where there are no trivial multiplications beyond 1 and –1, we propose a radix-8 NTTT algorithm, which also covers the case of negative wrapped (Nega cyclic) convolutions,

extensively used by some NTT-friendly NIST PQC schemes. Again, the algorithm is easily vectorized for short vectors.

In both cases we also discuss a possible module architecture implementing the NTT and its inverse. The focus of our work was less hardware oriented, and more algorithm oriented.

3.5.3.3 Place in the System

The NTT algorithms we propose can be used in two ways. First, they can be used as software algorithms taking advantage of various extensions to RISC-V (vectorial extension, SIMD extension, or an extension for modular arithmetic). Second, they could be implemented in hardware as memory-mapped accelerators attached to a system bus such as AMBA AXI, while a separate cryptographic coprocessor attached to the CPU by a dedicated interface offloads the NTT and its inverse as needed.

3.5.3.4 Block Diagram

NTT_LARGE_PRIME

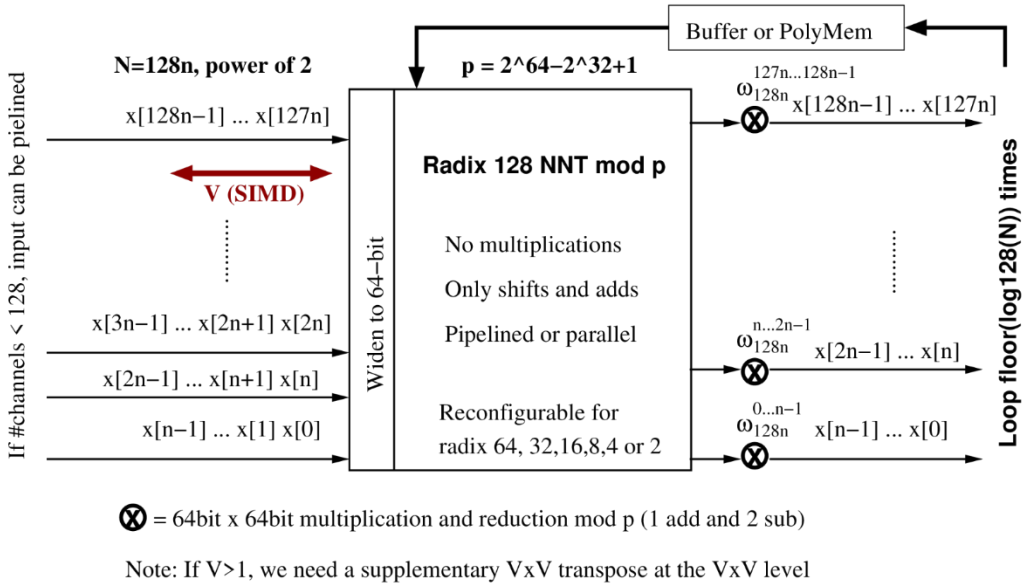


Figure 3.5.3.4-1: NTT - Diagram for a radix 128 NNT for a large prime p = 2^{64}- 2^{32} +1

Figure 3.5.3.4-1 shows a recursive decimation-in-frequency NNT with scrambled output using radix-128 steps, equivalent to 7 radix 2 steps. The radix128 block has 128 inputs and 128 outputs. Note that algorithm is not in place, as the input coefficients (16-bit integers) need to be immediately widened to 64-bit integers. Therefore, the output of the radix-128 block (after multiplication with twiddle factors) needs to be saved in a buffer and used as input for the next step.

The 128 outputs need to be multiplied by 128 twiddle factors or roots of unity, and then reduced modulo p. Each of these 128 operations implies a full multiplication of two 64-bit integers to a 124-bit result and reduction back to a 64-bit result modulo p. The reduction itself does not need any supplementary multiplication like in Montgomery or Barrett reductions, only 1 shift, one addition and two subtractions of 64-bit integers. If there is no hardware option for full 64-bit multiplication, one can easily mimic it using multiplication of 32bits integers with a full 64bits result.

We need to use a buffer to save the full output with N values, as we cannot write back to the memory due to the widening to 64 bits. Then the same radix-128 block can be applied to each chunk of N/128 values, with a new value of N=N/128, and with new roots of unity. At the last step, depending on the value of N,

one needs a 64, 32, 16, 8, 4 or 2-radix step. In software, the simplest way is to program such blocks independently. If the radix-128 block is built in hardware, then it can be configured to use only a part of the 128 inputs and bypass some internal levels, equivalent to a reduced radix block. Clock gating can be also used to reduce the power for the unused parts.

The algorithm can be easily parallelized in SIMD manner, by working in parallel on V values, with V a power of 2. The only problem appears at the last levels when N is smaller than V itself. A simple solution, used especially for small values of V (2,4,8,16) is to stop the radix steps at N=VxV, then transpose each VxV chunk as a VxV matrix, and then continue down to N=1 as before.

A different and better solution is to replace the buffer with a scratchpad memory, or PolyMem, polymorphic memory as proposed in [Ciobanu2018]. Then there will be no need to do the VxV transposes, as this memory can be configured with VxV matrices and read as rows or as columns. Note that an extension of this polymorphic memory will be developed in ISOLDE by IMT, and we plan to test NTT with this extension.

A similar diagram as in Figure 3.5.3.4-1 can be used for the inverse NTT. Note that due to the scrambled output, we cannot reuse the same radix-128 block for the inverse NTT, and we need to implement a separate block that is the transpose of the direct block, reversing the flow from outputs back to inputs.

To compare to the fastest NTT used in PQC, that used by Kyber, we need to extend the original arrays of length 256 by zeros to 512. Then, for each array of length 512, we need to apply a radix-128 step and then a radix-4 step, with only 512 multiplications. Then we need another 512 multiplications to multiply the transforms pointwise, and then a single inverse NTT with another 512 multiplications. At the end, we still need to reduce first with respect to $(X^{256} + 1)$ with 256 subtractions, and then to reduce each coefficient modulo 3329, equivalent to other 3x256=768 simple multiplications. In total we need 2816 integer multiplications. The NTT used in Kyber needs 7 steps of radix 2 with 7*128 modular multiplications, a multiplication of 128 pairs of polynomials of degree 1 with 3*128 modular multiplications and then an inverse NTT again with 7*128 modular multiplications. In total we obtain 3072 modular multiplications, each being done with Montgomery or Barrett multiplications and needing each 3 integer multiplications. In total we obtain 9216 integer multiplications, 3.3 times more than our proposal. However, we need to consider that all integer multiplications for Kyber are done with 16bit numbers, while our multiplications are done with 64bit numbers. Also, in practice, some software implementations of Kyber use SIMD acceleration to drastically reduce the total running time, whereas the use of 16bit integers may prove advantageous as more values can be compressed in a SIMD vector. However, a hardware implementation of NTT using the diagram above could prove comparable to Kyber specific implementations, with the bonus of having an “universal” NTT applicable to many other PQC algorithms.

NTT_CRT (NTT Chinese Remainder Theorem)

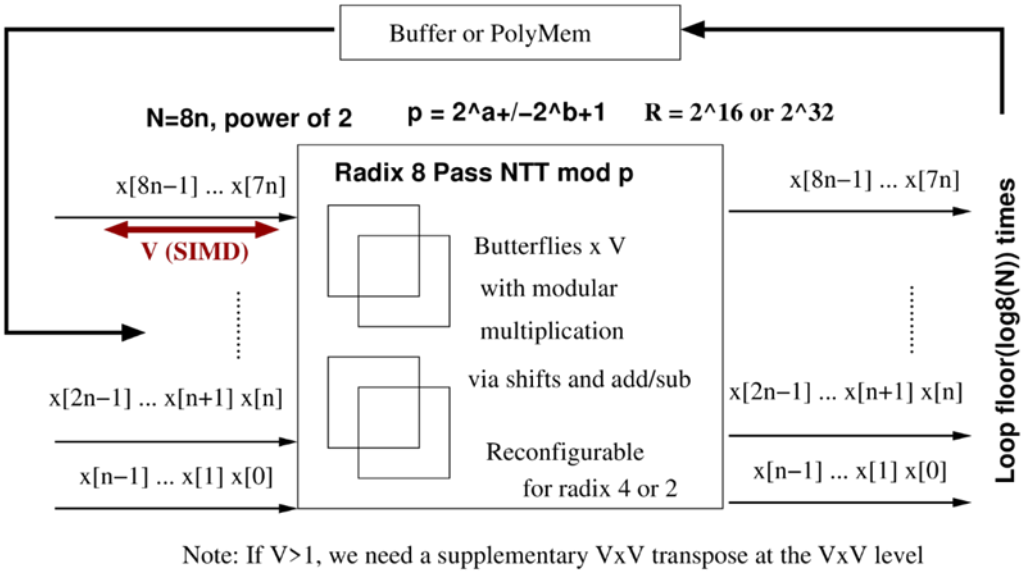


Figure 3.5.3.4-2: NTT - Diagram for a radix 8 NNT for small prime p

The diagram in Figure 3.5.3.4-2 shows a radix 8 NTT modulo a small prime of a special form, such that Montgomery or Barrett multiplication does need only a couple of shifts and integer additions or subtractions to achieve the reduction modulo p. Compared to the previous diagram, the prime p is rather small, so we can use 16bit or 32bit integers. The diagram shows how to use a vectorial or SIMD accelerator to do these NTT faster. Note that a transpose step is needed at level VxV where V is the length of the SIMD, similar to Figure 3.5.3.4-1. Also, in this case we will explore the use of PolyMem, a polymorphic memory, to cache intermediate results.

A similar diagram as in Figure 3.5.3.4-2 is used for the inverse NTT, which is essentially the transpose of the direct one. Note that one cannot use the same algorithm for direct and inverse due to the scrambled output. For the convolution, one uses Equation 3.5.3.1-2 - NTT for both arrays, point-wise multiplication of transforms, followed by one inverse NTT. The same convolution needs to be done for several primes p_i, and their results combined via the Chinese Remainder Theorem to obtain the convolution in $Z_p[X]$ where $p = p_1\ p_2\ p_3$. Finally, one needs to reduce modulo the polynomial P(x) and then modulo the original prime q, using an auxiliary modulus $R= 2^{64}$.

For the case of NTT-friendly algorithms, the diagram in Figure 3.5.3.4-2 can be used directly for a single prime, the one used in the algorithm. Acceleration can be obtained by using SIMD type vectorization with small values of V.

A possible hardware implementation would follow the same diagram and ideas.

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

Part of Task 3.5 Cryptographic and security accelerators.

3.5.4.1 General Information

Our module is a PQC coprocessor for the Classic McEliece with polynomial multiplication accelerated by a Number Theoretic Transform module. The coprocessor talks to a CVA-6 RISC-V core via the CVX-IF interface. The rest of the systems integrate an AMBA-AXI-4 bus as the main communication channel. We are targeting the system to be demonstrated on a Xilinx VCU128 FPGA board.

3.5.4.2 Purpose and Scope

The module targets the Automotive and Space demonstrators, where they will have similar interfaces to their own RISC-V cores in hardware and software.

3.5.4.3 Place in the System

The SAL PQC module is a hardware accelerator with specialized modules that speeds up computation performed by some of the security modules developed by the ISOLDE project. The module uses the RISC-V core from the CVA-6 being developed as part of WP2. The accelerator instructions are transferred from the core via the CV-X-IF, also under development. We will also contribute to the development of the necessary instructions as part of WP3, as well as to the software toolchains and compilers as part of WP4. The completed module is aimed mainly towards the automotive demonstrator as shown in the block diagram found in Figure 3.5.4.3-1, but we study its possible application in the space demonstrator.

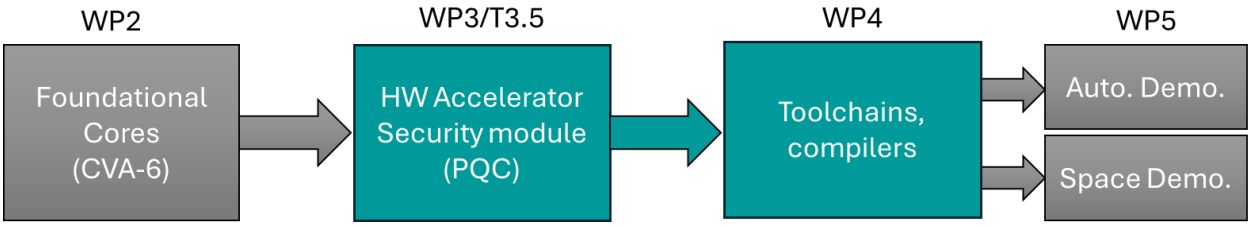


Figure 3.5.4.3-1: PQC-MA - Place in the system

3.5.4.4 Block Diagram

The architecture of our implementation in ISOLDE involves multiple open-HW modules along with the modules developed at SAL. The design incorporates a CVA6, AXI-4 Bus, SRAM blocks, and various peripheral/interface-bridge IPs and our own co-processor that focuses on processing PQC primitives as shown in Figure 3.5.4.4-1. Our core RISC-V processor is the Open-HW Group's CVA6. It has seen widespread adoption for RISC-based PQC designs in recent years with its ability to extend the RISC-V instruction set for PQC operations via the CV-X interface. This allows the implementation of a PQC Instruction Set Extension (ISE) on-top of the RISC-V standard ISA, capable of speeding up the computations compared to any memory mapped solution. We hope to have a partial implementation of RISC-V's ongoing PQC ISE.

For our co-processor, we draw inspiration from a number of papers that proposed a method of PQC acceleration and existing non-PQC-based co-processors, particularly PULP's vector co-processor, Ara. The PQC co-processor will contain NTT & Inverse NTT (INTT) accelerators, targeted specifically for the large key sizes involved in code based PQC schemes. In addition, an accelerator for non-NTT based polynomial multiplications may be designed and included to provide better performance for the symmetric-key portion of lattice-based communications. One final module we may decide to include is a memory

management unit for the larger-keys to boost efficiency and security in key storage and retrieval, with the additional functionality of improving security against side channel attacks and error-detection/- correction.

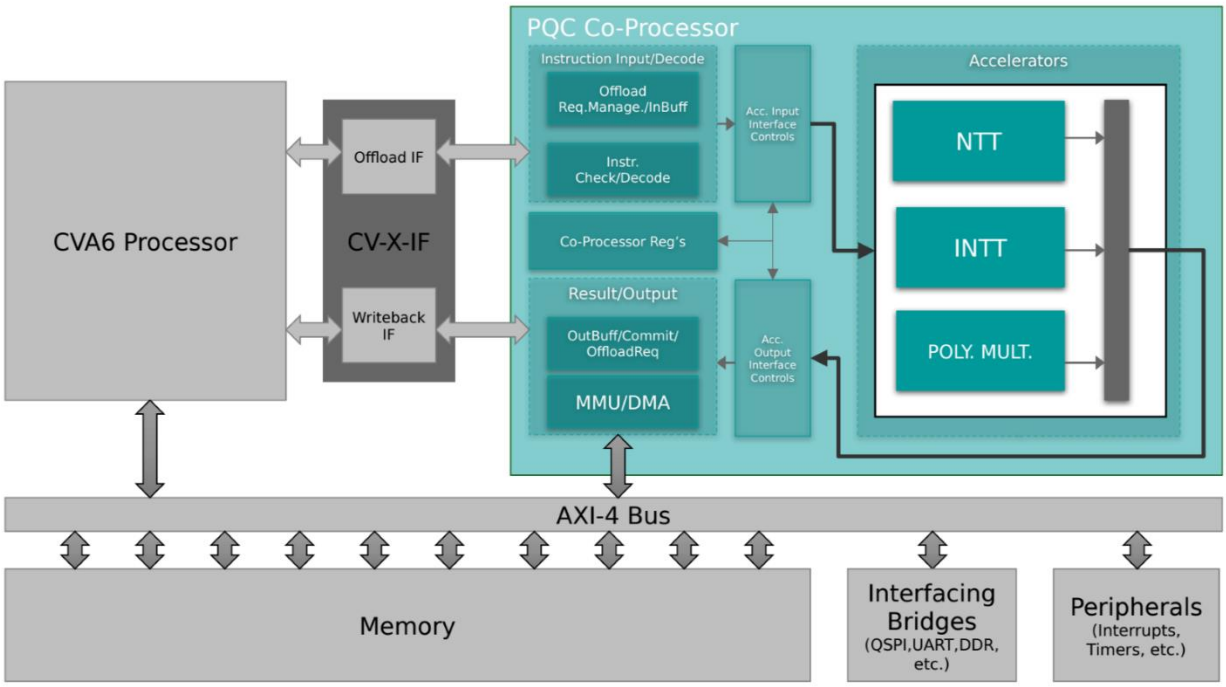


Figure 3.5.4.4-1: PQC-MA - Block diagram

3.5.4.5 Interfaces

The accelerator talks to the CVA6 core with a CV-X-IF and AXI. The rest of the modules are interfaced via an AXI bus, for example for the memory units.

CV-X-IF

This handles the transfer of instructions from the core to the coprocessor and transfers the results back to the core registers.

AXI

The overall system is built with an AMBA AXI bus, with the accelerator module also AXI-enabled.

3.5.4.6 Sub-Modules

The PQC module hosts an NTT accelerator submodule.

NTT/INTT

It allows to perform the multiplication of two discrete polynomials, which is a linear convolution in the finite field as a much simpler, pointwise, multiplication operation. With NTT, frequently-used, complex computational operations in PQC can be made quasi-linear, or “linearithmic” with time complexity of O(n log n) compared to polynomial with a time complexity of $O(n^2)$.

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

Part of Task 3.5 Cryptographic and security accelerators.

3.5.5.1 General Information

An open-source distribution of a lightweight RISC-V processor is being enhanced to create a secured version by incorporating cryptographic accelerators, designed to speed up encryption and decryption operations. This secure version will be deployed in commercial applications and then contributed back to the open-source community. The RISC-V architecture, known for its flexibility and cost-effectiveness, is gaining traction as an alternative to proprietary ISAs. While it has advantages such as being free, sanction-free, and easier to modify, it's still relatively new and faces challenges in terms of ecosystem support and feature parity with established ISAs like Arm or x86. The project aims to improve security while maintaining an open-source approach.

3.5.5.2 Purpose and Scope

The primary goal is to take an existing open-source RISC-V processor and adapt it into a secure microcontroller suitable for commercial applications. The focus is on enhancing security features, including cryptographic acceleration, to ensure robust protection against threats. These accelerators enhance the microcontroller's ability to handle cryptographic algorithms efficiently as presented in Figure 3.5.5.3-1.

3.5.5.3 Place in the System

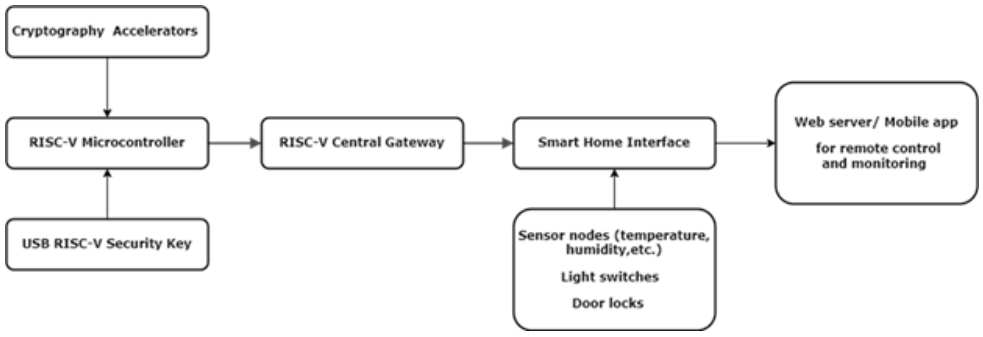


Figure 3.5.5.3-1: SEC - Place in the system

We must take into account the overall architecture, connectivity, and the function of each component while designing a secure RISC-V microcontroller architecture with cryptographic acceleration for a smart home use case (see Figure 3.5.5.3-1).

Primary Components:

- RISC-V Microcontroller (with cryptographic acceleration)

  Location: Serving as the central processing unit, at the center of the system.
  Its goal is to oversee all smart home features, such as data processing, connectivity with peripherals, and security management using cryptographic techniques to ensure safe data storage and transfer.

- Cryptographic Accelerator

Location: Integrated inside the RISC-V microprocessor itself or attached to it as a separate module offloading and speeding up cryptographic processes (encryption and decryption).

3.5.5.4 Block Diagram

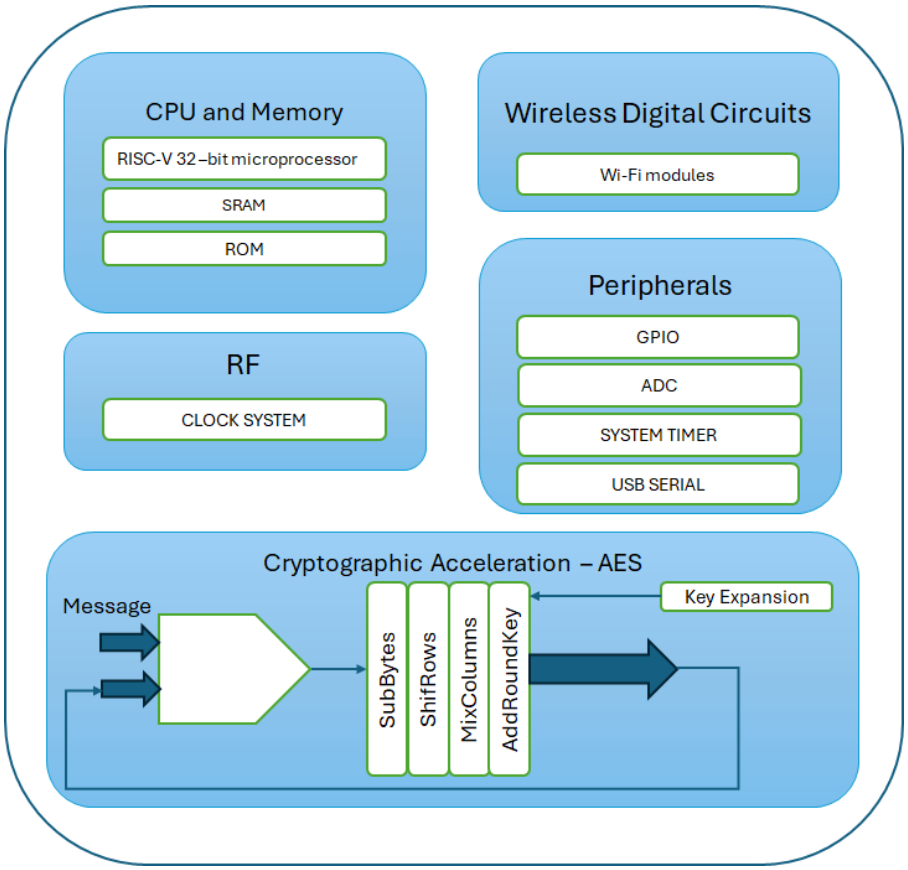


Figure 3.5.5.4-1: SEC - Block diagram

CPU and memory, wireless digital circuits, peripherals, RF clock system and cryptographic acceleration are among the components shown in Figure 3.5.5.4-1 that coalesce together to build an embedded system which is robust as well as secure.

Also, the embedded system manages memory peripheral interactions, executes encryption algorithms from initialization of the system and handles all user-peripheral interactions. This CPU has been designed following the RISC-V ISA. During its operation it fetches instructions through a combination of two types of memories: SRAM and ROM. The firmware that tends to be seldom changed on the other hand is usually stored in ROM which includes such essential software as the bootloader for the systems and minimal operating systems. In this case however, during start-up the CPU uses ROM to initialize itself. On the other hand, SRAM provides high-speed temporary storage for data or instructions that are being actively used by the CPU thus enabling rapid read/write operations which support dynamic processing tasks.

Modern connected applications depend on Wi-Fi modules for wireless communication. Networking protocols at lower levels are dealt with by this Wi-Fi module that does signal modulation/demodulation besides securing data while it traverses through network channels in form of encrypted/decrypted format. The CPU interfaces with the Wi-Fi module to send and receive data packets, allowing the system to communicate with other devices and networks wirelessly.

Peripherals extend the system's capabilities, allowing it to interface with a wide range of external devices and sensors. Various input and output activities can be connected to GPIO pins. These pins can either be programmed by the CPU to receive signals from external sources like sensors or transmit signals for controlling other hardware. The Analog-to-Digital Converter (ADC) enables the conversion of analog signals into digital data which is then manipulated by the CPU. This feature is essential in interfacing with analog sensors that detect physical quantities such as temperature or light. Furthermore, there is a system timer that provides accurate timing functions which help in scheduling tasks, generating time delays and timestamping events by CPU. USB Serial allows serial communication with external devices.

The microcontroller's CPU is in charge of carrying out the operating system and application code. It coordinates the actions of every other part of the system.

Cryptographic Accelerator: A specialized hardware component made to effectively carry out encryption and decryption operations, greatly boosting the system's overall security and performance of cryptographic duties. The details of the cryptographic accelerator are still under development and will be added in future deliverables. Accelerating other primitives like the Data Encryption Standard (DES) and 3DES in addition to AES is under evaluation.

Communication Interfaces: These consist of a secure Wi-Fi interface, which allow the microcontroller to connect to different networks and gadgets in the ecosystem of smart homes. The cryptographic accelerator makes secure communication easier.

3.5.5.5 Clocking Strategy

Clock signals act as the beating heart that synchronizes the activity of several components in a microcontroller architecture. For the integration of a secure RISC-V microcontroller module, especially one with cryptographic capabilities for smart home applications, the following kinds of clock signals are taken into account:

1. **Main System Clock:**
    a) Source: May come from an external source or is often produced by an on-chip oscillator.
    b) Purpose: The timing for instruction fetch, decode, execute, and write-back cycles is provided by it, which powers the CPU core.
    c) Frequency: Depending on the power limitations and performance requirements, it usually spans a few MHz to GHz.
2. **Peripheral Clocks:**
    a) Source: Obtained from either independent oscillators or the main system clock.
    b) Purpose: Numerous peripheral interfaces, including SPI, I2C, UART, and others, use these clocks. They can frequently be lowered from the primary clock in order to conserve energy or satisfy the peripheral's timing needs.
    c) Frequency: Typically, lower than the system clock to accommodate the particular peripheral's requirements.
3. **Cryptographic Accelerator Clock:**
    a) Source: It may be the cryptography unit's own dedicated clock or the same as the main system clock.
    b) Purpose: The efficient and timely execution of cryptographic operations is guaranteed by this clock. The cryptographic accelerator can function independently of the CPU core when it has its own clock, which is advantageous for carrying out cryptographic operations in the background.

c) Frequency: When processing cryptographic methods quickly is important, it must be set high enough. However, it can also be set to power-saving modes.

3.5.5.6 Reset Strategy

The reset signal in a microcontroller system is essential for guaranteeing that the system boots up in a known condition. The following describes the common reset signals used in RISC-V microcontroller modules, along with the module's actions during a reset:

1. Power-On Reset (POR):
   a) Activated upon microcontroller power-up.
   b) Guarantees that, prior to the CPU beginning execution, all registers and states are initialized to their default settings.
2. External Reset:
   a) A specific pin on the microcontroller that is often activated by an outside source.
   b) Used by external watchdogs or for manual resets.
3. Software Reset:
   a) Caused by the microcontroller's software, frequently by writing to a particular register.
   b) Can be used to force a software restart of the system in the event of an unrecoverable error.

Behavior During Reset

1. Core CPU:
   a) The CPU halts the execution of commands.
   b) The reset vector address, which is usually the beginning of the bootloader or original firmware, is where the program counter is set.
   c) The initial state of the CPU registers is set.
2. Memory:
   a) RAM and registers that are volatile are wiped or reset to their initial settings.
   b) Non-volatile memory, such as Flash, does not alter.
3. Communication Interfaces:
   a) Any active transactions are stopped when serial ports, network interfaces, and other communication modules are reset.
4. Cryptographic Accelerator:
   a) Cryptographic activities that are in progress are terminated.
   b) To stop leaks, sensitive data in registers and cryptographic keys should be purged.
5. Clock System:
   a) The clock system is reset, which can entail turning off internal oscillators or putting clock multipliers and divisions back in their initial settings.

The RISC-V microcontroller's particular implementation and configuration will determine the precise behavior. Additional precautions are taken in secure applications to guarantee that resets do not jeopardize the device's security status and that sensitive data, including keys, are sufficiently safeguarded even during reset procedures.

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

Task 3.6, M3-M33, Task Leader: CODA

Task 3.6 of the ISOLDE project focusses on three main areas of interest. The first area is an application specific instruction set processor focused on motor control partially developed by cooperation of NXP-CZ, CODA and BUT. NXP-CZ uses their highly valued SW expertise to provide the source code of the real application snippets which are profiled by the CODA tools and the results of the profiling are used to drive the tailoring process of the baseline RISC-V CPU. BUT performs Power, Performance, and Area (PPA) analysis of the developed IP and uses the result of the analysis to provide feedback during the implementation phase.

The second area of interest is the signal processing domain. IMT and ACP are implementing domain specific accelerators of the FFT/iFFT algorithms.

The third area of interest is neuromorphic computing. POLITO is working on the integration of their neuromorphic accelerator into the RISC-V framework.

IP	Lead Beneficiary	Type	Domain	Dependencies	Licensing
FFT	IMT	Algorithm	Signal processing	None	Restrictive open source (GPL-3.0)
LDPC	ACP	Core	Signal processing	Previous parts of receive chain (synchronization, FFT, equalization, LLR extraction)	Proprietary
Motor Control Accelerator	CODA	Core	Motor Control	CODA background IP	Proprietary
Neuromorphic HW Accelerator	PoliTo	Core	Neuromorphic computing acceleration	CVA6, AXI	Permissive open source (SHL, MIT)
SCA	ACP	Core	Signal processing	RF transceiver	Proprietary

Table 3.6-1: Overview of contributions in Task 3.6

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

Part of Task 3.6 Signal processing, neuromorphic and application-specific instruction set processors.

3.6.1.1 General Information

The FFT is widely used in signal processing and numerical simulations. For example, it can be used to accelerate the convolution of two arrays with application to finite impulse response (FIR) filtering.

There are many algorithms for FFT, most of them variants of the Cooley-Tuckey algorithm. The total number of arithmetic operations (additions/subtractions and multiplications) for these algorithms is close to 5 N $\log_2$(N), where N is the length of the array (and a power of 2). For modern CPUs, there is almost no difference in latency and throughput for additions /subtractions and multiplications. However, for hardware implementation, a multiplier, especially for floating point, is rather complex and large.

The split-radix FFT reduces the total number of operations to 4 N $\log_2$(N), and this was for a long time the best possible result. A recent improvement has given a slight reduction in the number of arithmetic operations with the constant 4 reduced to 34/9=3.78. However, this new algorithm has proven difficult to implement and numerically unstable.

To accelerate the FFT one can use either parallel or vectorial acceleration. For the many variants of the Cooley-Tuckey algorithm, one can find some which are better for one type of acceleration, and in many cases, they each have a transpose form which is good for the other type of acceleration. However, for the split-radix FFT, there are much less variants, and it is not clear which one is best suited for parallel or vectorial acceleration.

3.6.1.2 Purpose and Scope

We propose a split-radix FFT algorithm for arrays of floating-point values with lengths a power of 2, adapted for SIMD-type vectorization. The algorithm is rather flexible and can be easily adapted to various SIMD accelerators or to a hardware implementation. The biggest limitation is related to the number of values V in a SIMD vector, and the algorithm is practically limited to relatively small values of V (2,4,8,16).

The proposed split-radix FFT algorithm is recursive, decimation-in-frequency, in-place, with scrambled output and precomputed roots of unity. It uses optimally an existing cache (or buffer) without any knowledge of its size, due to the recursive nature (it works recursively on smaller and smaller parts of the original array, until the part fits in the cache). For real arrays it is still in place and twice as fast as the complex version. A scalar implementation was proposed by D.J. Bernstein [Bernstein1999]; our proposal is practically the SIMD vectorization of this scalar version.

For the case of large values of V, like in the RISC-V vectorial extension, the optimal FFT algorithms are more akin to those invented in the last century for various vector processors. We also explore a split-radix version suited for this case.

We have also developed an FFT algorithm that can be used for fixed point arrays, to calculate the convolution of two real-valued arrays with fixed-point values. Compared to the usual FFT fixed point implementations that mimic the floating-point algorithms and loose around $\log_2$(N)/2 bits of precision, where N is the length of the arrays, we calculate the convolution without any loss of precision, like a fused operation (with the entire convolution fused). We start by interpreting the fixed-point values as integers, and then do the convolution as the multiplication of two polynomials with integer coefficients. At the end we obtain the desired convolution with integer values (but with a larger width), and the user can reinterpret them as fixed-point values as needed. For example, for two arrays of length 32768 = 2^{15} with 24-bits fixed-point values, we obtain the exact convolution as integers with 15+24+24=63 bits (actually 64bit integers). The user can then reduce each coefficient to the desired fixed-point width (24 or larger). The multiplication of two polynomials is done using Number Theoretic Transforms (NTT), as we proposed in Task 3.5 for

cryptographic applications (Section 3.5.3). Note that if implemented in hardware, such NTT accelerator could be used both for cryptography and for signal processing.

3.6.1.3 Place in the System

The FFT algorithms we propose can be used in two ways. First, they can be used as software algorithms taking advantage of various extensions to RISC-V (vectorial extension, SIMD extension). Second, they cand be implemented in hardware as a memory-mapped accelerator attached to a system bus such as AMBA AXI, with the CPU or a Digital Signal Processor attached to the CPU by a dedicated interface offloads the FFT and its inverse as needed.

3.6.1.4 Block Diagram

SIMD (small vector) FFT split radix algorithm and implementation

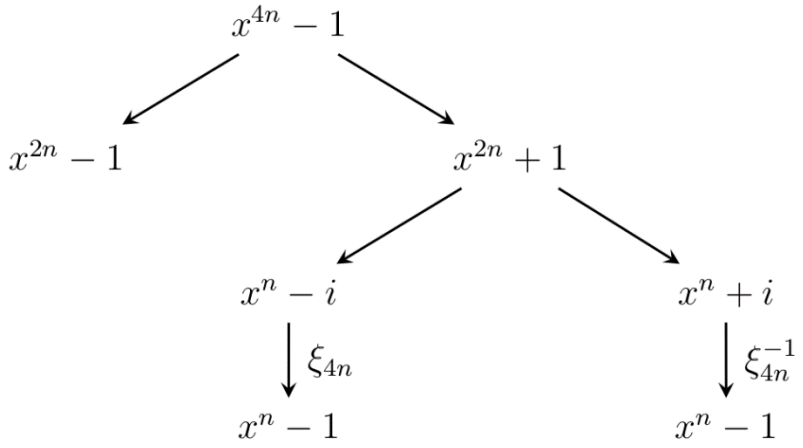
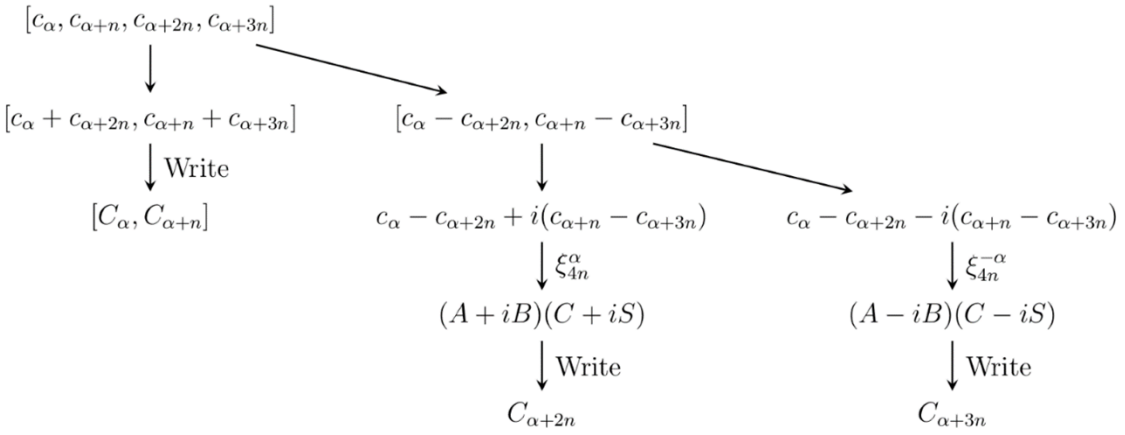


Figure 3.6.1.4-1: FFT - Decimation in frequency split radix step interpreted as polynomials transforms. The 4n step is reduced to one 2n step and two n steps. Note the twisting of the polynomial coefficients using roots of unity and their inverse (conjugates).

Figure 3.6.1.4-1 shows the decimation in frequency split radix 2/4 algorithm, presented as a series of polynomial transformations that calculate from a given residue modulo $X^{4n}-1$, the residues modulo of smaller degree polynomials [Bernstein2007]. Note that after such a step, one can apply recursively the same step to smaller degrees polynomials.



Split radix FFT applied to complex input. Here $\alpha = [0..n-1]$, $A = c_\alpha - c_{\alpha+2n}$, $B = c_{\alpha+n} - c_{\alpha+3n}$ and $C + iS = e^{-2\pi i\alpha/4n}$ is a root of unity of degree $4n$. Note that A and B are complex numbers. An in-place algorithm is obtained if the complex output values $[C_\alpha, C_{\alpha+n}, C_{\alpha+2n}, C_{\alpha+3n}]$ actually overwrite the initial ones $[c_\alpha, c_{\alpha+n}, c_{\alpha+2n}, c_{\alpha+3n}]$.

Figure 3.6.1.4-2: FFT - In-place implementation for complex input

Figure 3.6.1.4-2 shows an in-place implementation for complex input. Note that SIMD-like parallelism is easy to implement if each complex input contains V contiguous values. However, this easy parallelism stops once the degree of the polynomial is less than V. A solution for this challenge is presented later.

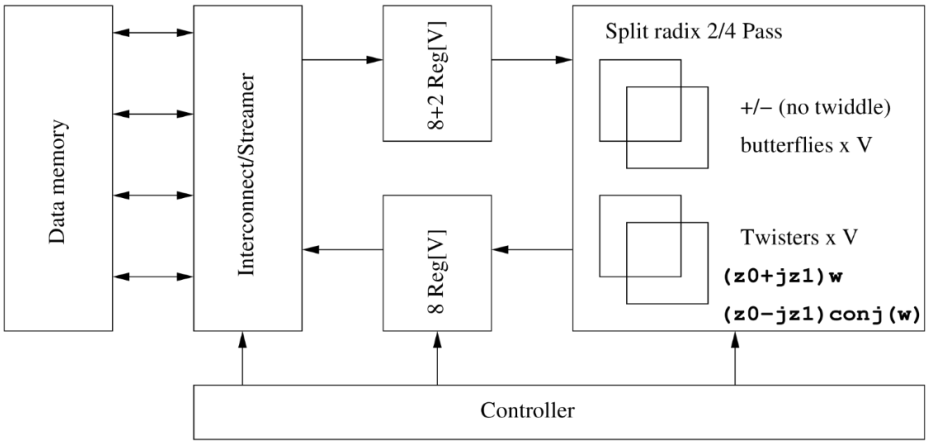


Figure 3.6.1.4-3: FFT - Possible hardware implementation

The diagram in Figure 3.6.1.4-3 shows the split radix algorithm, adapted for a hardware implementation. The 2/4 split radix block operates on 4 complex numbers, or 8 real numbers and outputs also 4 complex/8 real numbers. Being in-place, the output is written back to the same memory locations as the input. Therefore, we need 8 registers for input and output. The two supplementary input registers are needed for one complex root of unity which is precomputed. Compared to the classical split-radix algorithm, which needs two roots of unity, the second being the cube power of the first, we use the trick proposed by Bernstein [Bernstein1999] that replaces the cube with the conjugate of the root. Note that this will alter the output order, but this does not affect a convolution.

Inside the 2/4 split radix block we have some classical +/- butterflies (without multiplication) and a new type of butterfly called a twister (its operation is shown in the diagram). The V represents the number of values on which we operate in parallel (for example the number of values in a SIMD array). To accelerate the computations, we apply all butterfly operations on V values in parallel. For a hardware implementation, one could choose to increase the number of twisters working in parallel or use less twisters with a pipeline approach.

The same approach can be used for the same value of V if there are more registers available. For example, a 4/8 split radix block uses 8 complex values as input and two complex roots, needing 16+4 input registers and 16 output registers. Again, each operation applies on V values in parallel and may be interpreted as an “horizontal parallelism”.

The algorithm must stop when we reach a level of V, as now the horizontal parallelism should be done inside a V-long vector. A simple solution is to stop the split radix steps much earlier, at the V^2 level, and transform the horizontal parallelism to a “vertical” one by transposing each V^2 contiguous chunk as a matrix. Then one can continue to the lower levels by applying the same split-radix pass to V different vectors, each with length V.

Finally, let us note that we need a separate implementation for the inverse FFT transform, as we cannot reuse the direct FFT due to the scrambled output. The inverse pass is the transpose of the direct one, going back from the output to the input of the direct pass.

Finally, for a real array, one can simply discard the second complex number generated by a twister, as it is exactly the complex conjugate of the first complex number. For a hardware implementation one could use the same split-radix block with an inactivated path to reduce the consumed power.

Compared to [Bernstein1999] where a real array needs an initial permutation (2 x N/2 -> N/2 x 2), we propose a new algorithm for real arrays that requires no permutation, with some penalization in memory access (doubling the number of channels) which can be absorbed by caching or other means.

Vector FFT split radix algorithm

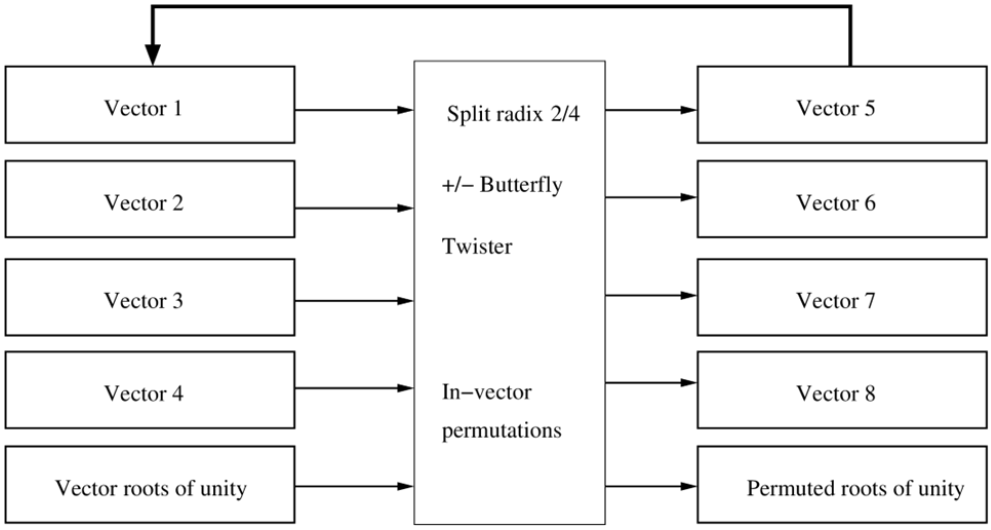


Figure 3.6.1.4-4: FFT - Diagram showing a possible hardware implementation for large values of V.

For large values of V (SIMD vector length), we need a completely different approach for vectorizing the FFT, as stopping at the V^2 level is not practical (as V^2 could be much larger than the length of the array).

There are several approaches possible. In [Kwong2012], Kwong and Goel propose a constant geometry architecture for the split radix FFT, by interpreting it as a radix-2 FFT where some twiddle coefficients are

trivial. Then they apply the usual Pease algorithm (parallelizing the radix-2 FFT) to obtain the sought after constant geometry architecture. Then they propose a hardware implementation copying the Pease radix-2 FFT, with no gain in throughput or latency. However, they propose to use clock-gating to save power when the twiddle factors are trivial. Note that this approach was patented by Texas Instruments in 2011, [Kwong2013].

The dual algorithm for the Pease algorithm is called Korn-Lambiotte and is a constant geometry radix-2 algorithm suitable for vector processors (see [Franchetti2011]). It was recently used in [Vizcaino2023] for long vector architectures, including the vectorial extension for RISC-V.

Our proposal is to adapt the Korn-Lambiotte algorithm from the radix-2 algorithm to the split-radix one. First, the SIMD-type split radix recursive algorithm introduced before can still be used down to a level 4V (using also some radix 2 steps as the last ones). However, at that level the entire array enters in a couple of registers of length V. From this level down to level 1, we do not need to write back intermediate results to memory, but all calculations can be done inside the registers. The diagram in Figure 3.6.1.4-4 shows four vectors of length V – each holding half of the V values, either real or imaginary parts. At every step, we apply a split radix 2/4 to the vectors, together with some in-vector permutations and the results are saved in new vectors. All roots of unity enter in a single vector (real and imaginary parts, as only 2V/4 roots are needed for the first 2V step). At each step we also recalculate a new vector of roots of unity, by deleting some roots, duplicating others and applying some permutations. Another solution is to use at every step a different precomputed vector with correct roots of unity, read from memory.

Compared to a vector radix 2 algorithm, the split-radix 4 needs the same number of steps, $\log_2(2V)$. Contrary to [Kwong2012], where the same steps from radix 2 were used for split-radix 2/4 with a gain in power for trivial twiddles, in our approach the steps for the split-radix 2/4 are exactly the same as for the scalar case. However, note that after $\log_4(2V)$ steps, some of the values in each vector are already the final ones and should not be changed in further steps. One idea is to push these values to the end of the vector at each step and mask them so that subsequent steps do not affect them. The efficiency of this approach will largely depend on the implementation of masked vector operations and will give some gains compared to vectorial radix 2 algorithms only if these masked operations have lower latencies or higher throughput. A better solution is to use a polymorphic memory, as proposed in [Ciobanu2018]. In this case, we can reduce the length of the vectors at each step as needed, retaining only the values that still need to be acted on, thus decreasing the latency, and at the same time start moving the finished values back to memory. Note that such a polymorphic memory will be developed in this project by IMT (Section 3.2.3), and we intend to check it for vectorized FFTs with large values of V.

Fixed point convolution using number theoretic transforms (NTT)

This algorithm has been presented in Section 3.5.3 as the NTT for post quantum cryptography. There are two variants, a large prime one using 64bit integers and another one based on residue arithmetic using 16bit integers. Both give the same end result.

The idea of using number theoretic transforms for convolution of real vectors is not new. One advantage is that the algorithm is intrinsically adapted to real values, without the need of steps for complex arrays and with real roots of unity. Other advantages are described below for the case of fixed-point values.

We start by interpreting the fixed-point values as unsigned integers. Then we can do the convolution defined as the multiplication of two polynomials with integer coefficients via NTT, as described in detail in Section 3.5.3. The end result is the desired convolution with integer coefficients with larger width, without any loss in precision (perfectly exact solution). For example, for two arrays of length $32768 = 2^{15}$ with values fixed-point with 24 bits, we obtain the exact convolution as integers with 15+24+24=63 bits (actually 64-bit integers). The user can then reduce each coefficient to the desired fixed-point width.

Note that if implemented in hardware, such an NTT accelerator could be used both for cryptography and for signal processing.

3.6.2 Low Density Parity Check Decoder (LDPC) – ACP

Part of Task 3.6 Signal processing, neuromorphic and application-specific instruction set processors.

3.6.2.1 General Information

Low density parity check (LDPC) codes are linear block channel codes defined by a sparse parity check matrix. Due to the superior error correcting performance and high degree of parallelization, they have been adopted and used in many wired and wireless standards.

Among the different type of LDPC codes, quasi-cyclic (QC) LDPC codes are very popular as they enable an efficient hardware implementation for both the encoder and the decoder. The 5G New Radio (NR) LDPC code is a (systematic and irregular) QC-LDPC, designed for a peak throughput 20 Gbps and has two base graphs, BG1 and BG2, that can be expanded to support various code rate and block length by changing the lifting factor.

3.6.2.2 Purpose and Scope

Different hardware architectures for LDPC decoders have been proposed to meet the power and throughput requirements of various standards. Among those, various degrees of resource sharing yield flexible decoders with varying area requirements. For this project's purpose, we opted for row-parallel architecture as it strikes the balance between area and the decoder throughput. In this architecture, multiple rows of the parity check matrix are processed in parallel, enhancing throughput. The degree of parallelization offers flexibility to trade throughput for area, as the required decoder throughput may vary depending on preceding signal processing blocks. Additionally, this architecture must support both base graphs and various combinations of lifting factors and block lengths.

3.6.2.3 Place in the System

The LDPC decoder is placed in the baseband domain of the IoT demonstrator together with other baseband accelerators and memories as shown in Figure 3.6.2.3-1. The processor domain that executes the protocol stack is in control of the baseband domain and configures, starts, and stops the different accelerators at the right time.

The LDPC decoder is controlled via the baseband controller block, where a corresponding command is sent to the decoder indicating a decoding task. The decoder starts the decoding by fetching the code blocks LLRs (logarithmic likelihood ratios, a common term used in communications) stored in the LLR buffers and writes the results into Tightly-Coupled Data Memory (TCDM).

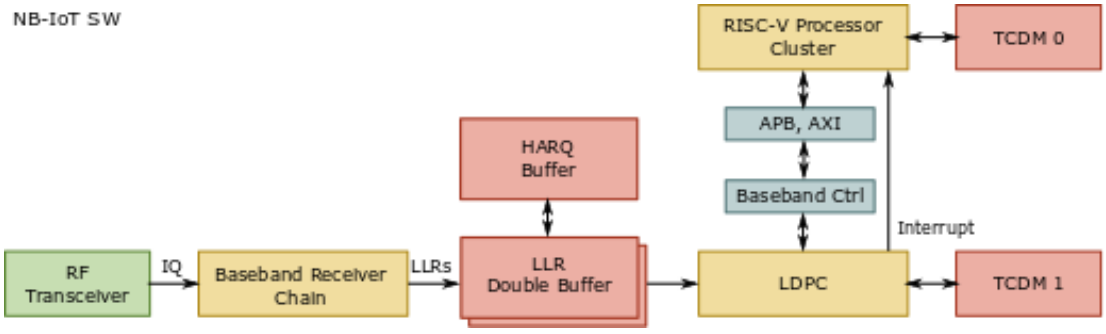


Figure 3.6.2.3-1: LDPC - System integration

3.6.2.4 Block Diagram

The LDPC decoder is designed to decode 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 is designed with a row-parallel architecture and supports the different lifting sizes as required by the standard.

The LDPC decoder is designed with a row-parallel architecture with a penalization factor of p, as illustrated in Figure 3.6.2.4-1. It consists of p internal LLR memories and p decoder functional units (DFU). The internal LLR memory provides Q-massages to the DFUs via the distribution network, where in the DFUs check node, variable node, LLR operations are performed. Each DFU has a local memory to store the resulting R-massages, which are feedback to the LLR memories via the gathering network.

At start, the LLRs are loaded into internal LLR memories and each hardware iteration processes p rows of the parity-check matrix. For example, for p=32 and Z=384, each Z rows (1 row of the base graph) is processed via 12 iterations. At the end, the decoded LLRs (or their sign bit) are loaded from the internal LLR memories into TCDM.

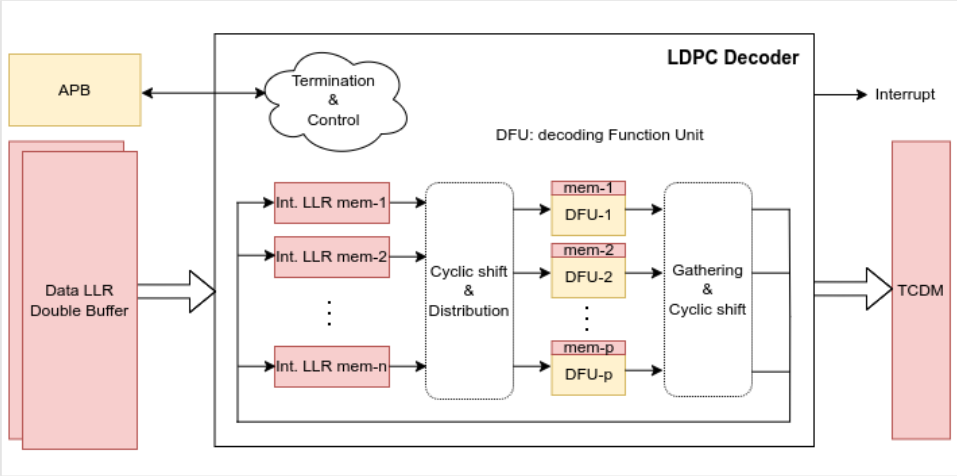


Figure 3.6.2.4-1: Internal Organization of the LDPC decoder with p parallel functional decoding units

3.6.2.5 Interfaces

The LDPC decoder has three interfaces and one interrupt line to indicate completion of a block decoding. The interrupt is connected to the processor's main interrupt controller and can be used to start subsequent processing.

Direct Memory Interface

The LDPC decoder has access to the accelerator TCDM through a dedicated interface. The decoder can read and write data to the TCDM. Processors, or other accelerators can then further process the results by accessing the TCDM.

LLR Buffer Interface

The LDPC decoder has access to the LLR buffer through a dedicated interface. The decoder can load input LLRs accumulated with Hybrid Automatic Repeat Request (HARQ) into its internal memory for each decoding iteration. The buffer is shared with a Turbo decoder that is used for a different protocol standard, but two decoders will not run concurrently.

APB Configuration Interface

The APB interface is used to configure, start, and stop the accelerator and is connected to the main bus of the baseband processor.

3.6.2.6 Clocking Strategy

The LDPC decoder is part of the digital baseband and clocked by the digital baseband clock.

3.6.2.7 Reset Strategy

The LDPC decoder is part of the digital baseband and can be reset asynchronously together with the rest of the digital baseband after power up.

3.6.2.8 Power Management Strategy

The LDPC decoder can be powered down with the rest of the digital baseband while leaving the processor domain on. This is typically the case when the chip is not connected to the base station or during DRX, or PSM cycles.

3.6.2.9 Debugging Strategy

The LDPC decoder’s Finite State Machine (FSM) cannot be interrupted by the debugger, but the number of iterations can be adjusted such that decoding results can be investigated after each iteration. In addition, the internal state of the decoder can be observed through the APB interface.

3.6.3 Motor Control Accelerator – CODA

Part of Task 3.6 Signal processing, neuromorphic and application-specific instruction set processors.

3.6.3.1 General Information

The IP proposed by CODA targets the automotive domain. NXP-CZ provides the Model Predictive Control algorithm that needs to be profiled. Based on the preliminary analysis of the algorithm, a 9-stage application class processor was selected to be the main core of the CODA accelerator. Further profiling and performance analysis will be used to identify the extension(s)/customization(s) for the processor to improve its performance to meet the control algorithm needs.

3.6.3.2 Purpose and Scope

The purpose of the IP developed by CODA is to demonstrate the usability of RISC-V CPU cores for computationally intensive applications in the automotive domain. CODA uses the Codasip Studio tool to tailor an existing RISC-V processor to effectively support the selected control application. The used RISC-V processor contains partial support of the Vector extension. The IP by CODA implemented as part of the ISOLDE project, further reported as foreground IP, will further enhance the existing implementation of Vector extension to be fully compliant or compatible with the RISC-V Vector extension (RVV) specification [RVI2021] and further tailor the processor by adding additional functionality and instructions necessary for the control application.

3.6.3.3 Place in the System

One of the most complex parts of model predictive control is the quadratic solver used to find the optimal solution for the motor control problem. The quadratic programming solver provided by the NXP-CZ will be the central part of the system. Codasip's 9-stage application processor will run the solver as well as any other software task. The processor contains a customizable application RISC-V core. Based on the profiling result, the new instruction(s) will be introduced into the ISA by CODA and their micro-architecture implemented accordingly. The foreground IPs developed by CODA as part of the ISOLDE project will be tightly connected with the background IPs, such as the existing baseline A730[14] CPU and/or VPU. The foreground IP may be considered another customization of the A730 CPU. The tight interconnection between foreground and background IPs provides important performance benefits and simplifies the design.

The use of the developed foreground IP with a different processor is not recommended since the foreground IP is being developed as the customization of the baseline CODA processor and therefore makes a significant amount of assumption about the processor. The baseline CPU can be connected to a host processor by a standard interface (such as AXI), or it can be used to run any standard RISC-V workload in addition to the accelerator task and remove the need of the host CPU from the system. In such a case, the AXI interface would be used to connect the CPU to the required peripherals.

[14] https://codasip.com/products/application-risc-v-processors/a730/

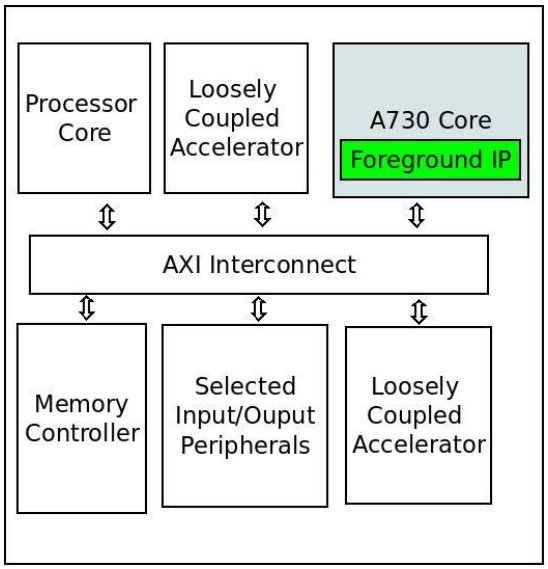


Figure 3.6.3.3-1: Motor Control Accelerator - Position in the system

Figure 3.6.3.3-1 demonstrates the position of the IP developed by CODA in a generic exemplary system-on-chip in a stand-alone configuration. The foreground IP is depicted by a shaded box in green color and is placed inside the background IP depicted by the gray to symbolize that the foreground IP is connected only to the background IP. The CODA A730 is just one part of the system connected by the AXI4 interface. The A730 may be used to control all remaining subsystems, or it can be controlled by another CPU depending on the system architecture.

3.6.3.4 Block Diagram

The block diagram in Figure 3.6.3.4-1 describes Codasip's approach to the design of the accelerator. The boxes shaded in gray depict components considered the background IP that will be provided by the CODA to evaluate foreground IP but will not be developed as part of the ISOLDE project. The green shaded components describe the foreground IPs that may be developed as part of the CODA contribution. If the customization for the specific step is implemented depends on the profiling results. The dark green represents the decoding of the new instructions and the execution of these instructions.

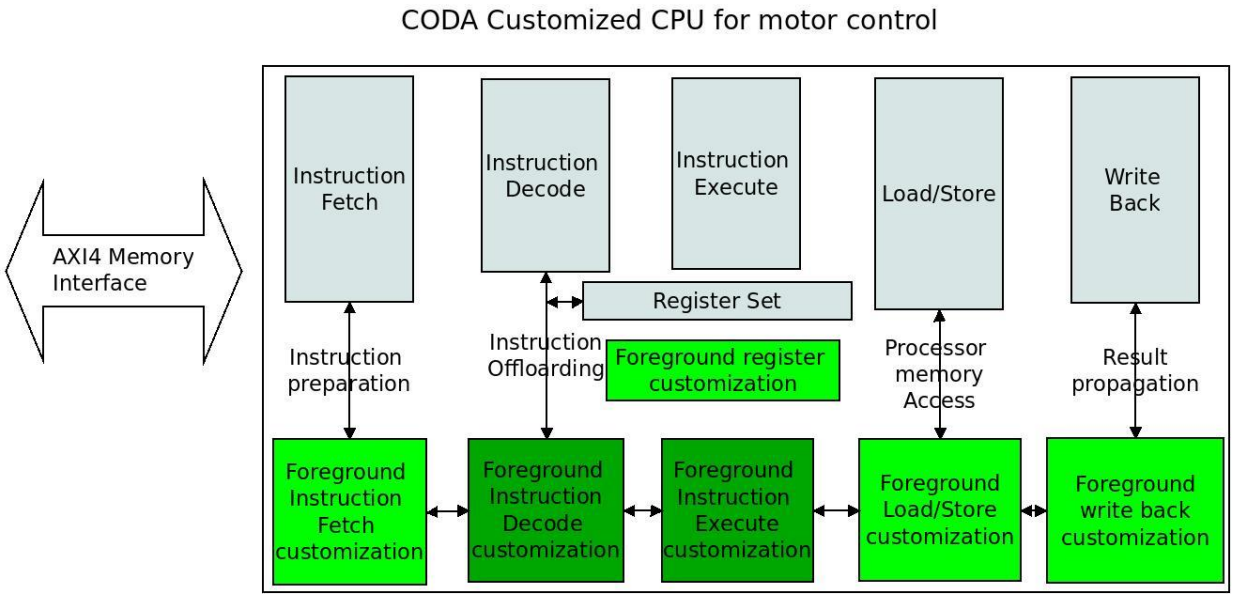


Figure 3.6.3.4-1: Motor Control Accelerator - CODA CPU tailored motor control domain

3.6.3.5 ISA

The CODA processor IP blocks are written in the Codal3 language, a high-level language used to automatically generate both the Software Development Kit (SDK) and Hardware Development Kit (HDK). The SDK contains the C compiler with assembler and instruction accurate simulator while HDK contains the RTL representation of the customized processor.

As can be seen in the block diagram in previous section, the CODA customization only adds new functionality and never removes already existing functionality. Therefore, CODA can guarantee that any functionality existing in the background IPs will remain unchanged and therefore standard RISC-V code is binary compatible. However, if the standard binary code is run on the customized processor, the new instructions will not be used and therefore the customization performance benefits will be lost.

The best way to approach this problem is to compile the application from the C code. The customized compiler automatically generated by CODA tools will be able to use new instructions to achieve the best performance.

The exact functionality of the custom instruction will be known based on the result of the profiling done by CODA, which will be repeated during the whole implementation phase. Any introduced custom instruction will be compliant with RISC-V specification.

3.6.3.6 Interfaces

The HW interfaces between the newly developed IPs and already existing IPs are automatically generated during the HLS synthesis from the Codal3 language.

The A730 CPU core is connected by the AXI4 interface.

3.6.3.7 Clocking Strategy

The newly implemented IP will be connected to the processor core by the automatically generated interface. It is recommended to use existing IPs provided by CODA to integrate into the system. Therefore, this section will describe the clocking strategy of the A730 processor core instead of the newly developed IP itself.

The A730 contains three main clock domains which are all driven by the single clock pin `CLK`. Each of the clock domains has its own `CLK_EN` pin. The following table contains the list of the enable signals together with a brief description.

Signal Name	Description
MEMSYS_CLKEN	Controls the memory management clock domain. Memory management system is shared between multiple processor cores in case of multicore configuration.
DBG_CLKEN	Controls the debug clock domain. Debug subsystem is shared between all cores in the multicore configuration.
CORE_CLKEN[n]	Controls the clock domain for the processor core. Each core has its own clock enable signal in case of multicore configuration.

Table 3.6.3.7-1: Motor Control Accelerator - List of the enable signals together with a brief description.

The foreground IP will be tightly connected with the processor pipeline and as such will operate on the core clock of the given core. In the case of the multicore configuration, each core will instantiate its own foreground IPs.

3.6.3.8 Reset Strategy

The resets of the CODA foreground IPs will be tightly coupled with the reset of the processor pipeline. Since the A730 core is the recommended IP, this section describes the reset process of the A730.

Signal Name	Polarity	Description
MEMSYS_RST	Active low	Reset for memory subsystem.
DBG_RST	Active low	Reset for debug subsystem
CORE_RST	Active low	Reset signal for the core. Each core in the multicore system has its own reset.

Table 3.6.3.8-1: Motor Control Accelerator - Reset of the A730

The foreground IP implemented as part of the ISOLDE project will be using `CORE_RST` since it is connected to the CPU core.

3.6.3.9 Debugging Strategy

All background CODA IP implement debug strategies described in the RISC-V specification. The foreground IP will continue in this trend and all additional functionalities will be accessible from the debug manager implemented according to the RISC-V External Debug Support [Donahue2024].

3.6.4 Neuromorphic HW Accelerator – POLITO

Part of Task 3.6 Signal processing, neuromorphic and application-specific instruction set processors.

3.6.4.1 General Information

Neuromorphic computing is a research field where Spiking Neural Networks (SNN), also called third generation neural networks, are explored to overcome the limitations of traditional Von Neumann architectures. SNNs represent a subset of AI applications that take inspiration from the human brain by emulating biological neurons and synapses for data processing and transfer, enabling event-driven, fault-tolerant computation with low latency and high parallelism, thanks to the proximity of memory and processing units.

3.6.4.2 Purpose and Scope

Following the growing interest in SNNs, we want to demonstrate how neuromorphic systems can perform well in computation from the point of view of power efficiency, latency and parallelism. The flexibility given by the RISC-V ISA and FPGAs allows the creation of digital/neuromorphic prototyping platform where edge and low latency computation from the neuromorphic accelerators is complemented by the control of a digital processor.

3.6.4.3 Place in the System

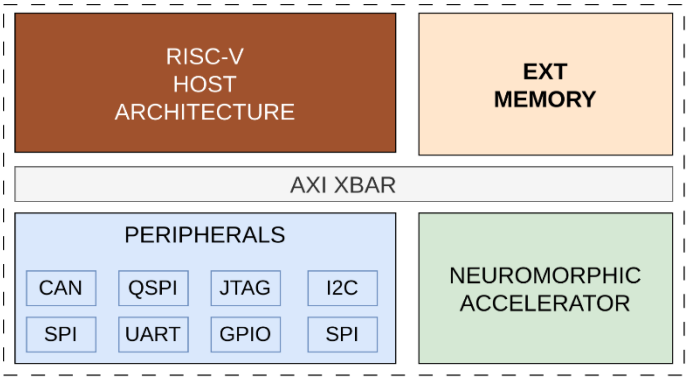


Figure 3.6.4.3-1: Neuromorphic HW Accelerator - Place in the system

In Figure 3.6.4.3-1 we report the high-level architecture of our HW. The high-speed AXI protocol will allow fast reconfiguration of the accelerator and delivery of input data encoded in spikes. Internal BRAM memories available on the FPGA are used by the packet encoder to store sample batches and by the neuromorphic HW to store networks weights and neuron statuses. However, this memory type offers limited capacity. Hence, for larger datasets an external, larger, memory will be used to store all the data that will be progressively offloaded to the accelerator.

3.6.4.4 Block Diagram

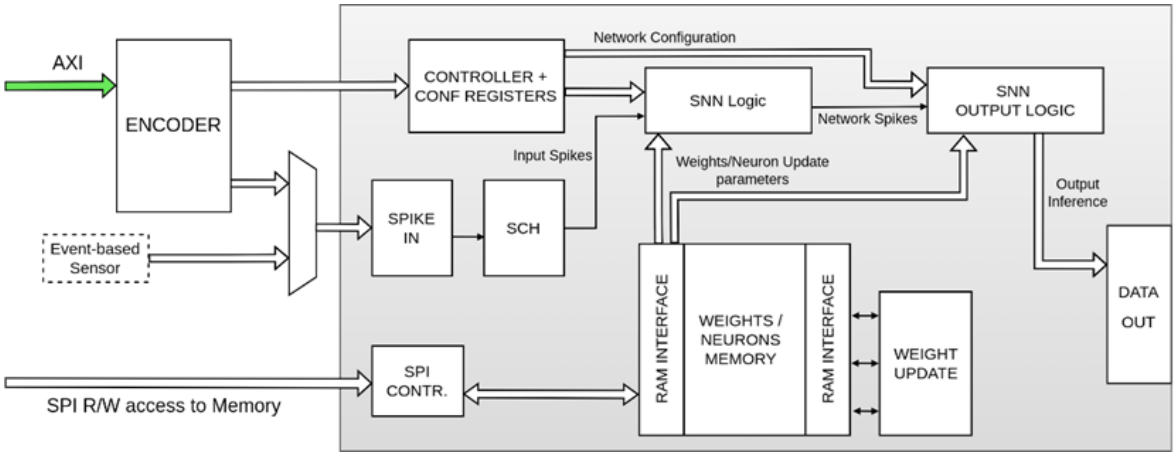


Figure 3.6.4.4-1: Neuromorphic HW Accelerator - Block diagram

The behavior of the neuromorphic accelerator can be controlled via a dedicated set of parameters that are stored internally in a dedicated register file (*CONF REGISTERS* in picture 3.6.4.4-1) operated by a *CONTROLLER*, which oversees delivering the network configuration to the SNN. Moreover, internal RAM elements are used to store weights and neuron statuses: They are accessible from the outside through a SPI-controlled device for writing or reading.

The possibility to instantiate the accelerator as a multicore architecture is to be investigated. In that case, a Scheduler (SCH) unit shall be introduced that handles spikes between cores.

The accelerator's operations are managed by a main FSM inside the *SNN logic* unit. It receives input spikes, computes the evolution of the network by implementing the behavioral neuron logic (mostly Leaky-Integrate-and-Fire - LIF) and (optionally) implements online learning through the configurable *WEIGHT UPDATE* unit.

The *SNN OUTPUT LOGIC* unit elaborates the output of the network and provides the inference results.

3.6.4.5 Interfaces

AXI

The AXI bus can be used to communicate the input data to the accelerator or the network configuration when it's necessary to implement a new design for a different use-case.

SPI

The SPI bus is used to access the memory elements internal to the accelerator, probably through an AXI-to-SPI bridge to allow the utilization of one unique transfer protocol from the main Host.

3.6.4.6 Clocking Strategy

Main clock signal of the system, autonomous time ticks' generation for event-based computing.

3.6.5 Shared Correlation Accelerator (SCA) – ACP

Part of Task 3.6 Signal processing, neuromorphic and application-specific instruction set processors.

3.6.5.1 General Information

In cellular wireless communication, it is necessary to align the timing and frequency references of the user equipment (UE) and base station (BS). This is necessary both during initial synchronization, when the connection is first set up, and after the UE exits low-power sleep modes. If the UE uses a low-cost crystal oscillator or the cell is operating in a high-frequency band, the initial frequency error can be substantial. To correct for this, the BS regularly transmits one of several known synchronization sequences. The UE then looks for these sequences using either low-complexity auto-correlation or cross-correlation approaches [Hazy1997]. The cross-correlation method is preferred for its superior performance but comes at a heavy computational cost.

The cross-correlation approach involves computing the cross-correlation of the received signal with the possible synchronization sequences over all time offsets [Kroll2017, Lippuner2020]. Additionally, this process needs to be repeated for a grid covering the range of the possible frequency offsets. Accordingly, the computational effort scales linearly with the maximum possible frequency error, which in turn scales with the maximum carrier frequency. We estimate the required correlation throughput at 81 Mcorr/s (million correlations per second) for LTE. For 5G NR Frequency Range 1 (FR1), the required throughput rises to 230 Mcorr/s due to the higher bandwidth and carrier frequency.

3.6.5.2 Purpose and Scope

The purpose of the Shared Correlation Accelerator (SCA) is computing the cross-correlations for the synchronization for a modem that supports both LTE and NR FR1. The SCA shall compute these correlations in real time, as storing the received samples for a longer duration is not feasible. If the cross-correlations are naively computed in time-domain, this would correspond to 470 Gop/s. A better approach is computing the cross-correlations in the frequency domain using the Overlap-Save method [Kroll2017, Lippuner2020]. With this approach, the complexity can be reduced to approximately 13 Gop/s. While still substantial, this is achievable using a dedicated hardware accelerator.

Additionally, the SCA shall also handle the accumulation of the correlation values in a shared TCDM. It shall also detect if a sequence has been found, allowing for early termination in that case.

3.6.5.3 Place in the System

The SCA is a part of the Digital Base Band (DBB). It is controlled by the software running on the processor cluster via an APB register set as shown in Figure 3.6.5.3-1. It is able to directly receive IQ samples from the RF transceiver and gets timing information from the DBB timekeeping unit. It has high-bandwidth access to the local TCDM, where the correlation results are stored and accumulated. Once processing is completed, an interrupt is raised on the processor cluster.

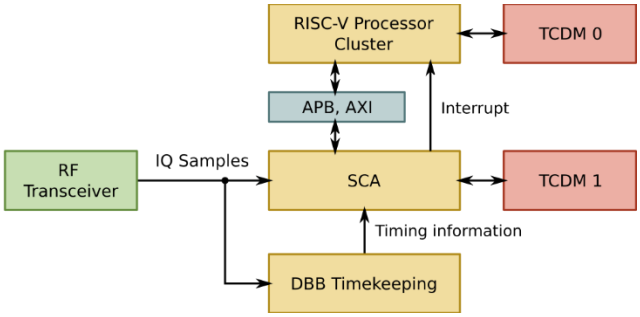


Figure 3.6.5.3-1: SCA in the DBB controlled over APB from a processor cluster.

3.6.5.4 Block Diagram

Figure 3.6.5.4-1 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 received samples to the frequency domain. After the correlation is computed using multiplication in the frequency domain, the FFT is used again to transform the correlation results back to time domain. A sample buffer at the input stores the incoming samples and absorbs the uneven consumption rate of the correlator. The final accumulation block decimates the time-domain correlation results and accumulates them in the TCDM.

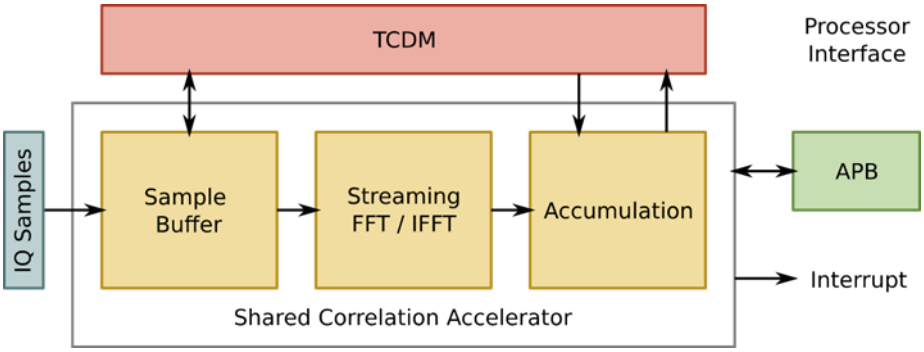


Figure 3.6.5.4-1: SCA architecture with sample buffer, streaming FFT and accumulator

3.6.5.5 Clocking Strategy

The SCA and its coupled memory use a single synchronous clock provided by the DBB.

3.6.5.6 Reset Strategy

An asynchronous reset is used to initialize the SCA after power on. A separate, synchronous clear can be used to return to the initial state. In the initial state, the SCA waits to be configured by the processor software.

3.6.5.7 Power Management Strategy

The SCA utilizes clock gating to reduce power consumption and the SCA can be powered down together with the rest of the DBB when no data reception is required.

3.6.5.8 Verification Strategy

The SCA is primarily verified against a bit-true software model in a stand-alone testbench, which enables full insight into the block. Additional observability in-system is provided by exposing the internal state using APB registers.

4 Conclusion

This deliverable provided initial architecture definitions for the hardware modules and extensions to be developed within WP3 (Accelerators and Extensions) of the ISOLDE project, building on the previously defined initial demonstrator and hardware module requirements (Deliverable D1.1 and D1.2). The initial architecture definitions include details about each component's purpose and preliminary information about its design (placement in the system, functional description, interfaces, and strategies for clocking, resetting, power management, and debugging). The deliverable structure follows the tasks defined in WP3 to make it easier to relate the presented content and technical progress with the project proposal.

Based on this document, the related work packages WP2 (providing the foundational cores), WP4 (developing the necessary software support for using the hardware extensions), and WP5 (combining the foundational cores with selected hardware extensions creating diverse demonstrators) gain more insight about the developed hardware extensions and modules. Hence, this deliverable is crucial for further collaboration between these work packages and ISOLDE's goal to create high-performance processing systems.

Regarding WP3, this deliverable will act as a starting point for subsequent deliverables covering the prototype and final implementations of the hardware extensions (D3.2, D3.3 in M24 and D3.4, D3.5 in M33).

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
CNN	Convolutional Neural Network
CORDIC	Coordinate Rotation Digital Computer
COP	Call-Oriented Programming
CPU	Central Processing Unit
CPS	Cyber-Physical Systems
CSR	Control and Status Register

CTM	Cryptographically Tagged Memory
CV-X-IF	Core-V eXtension Interface
DBB	Digital Base Band
DDR (SDRAM)	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
ReCo	Row Column
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

6 References

[Abella2023] J. Abella, F. J. Cazorla, S. Alcaide, M. Paulitsch, Y. Peng and I. P. Gouveia, "Envisioning a Safety Island to Enable HPC Devices in Safety-Critical Domains," 2023.

[arm2024] arm, "Armv8.5-A Memory Tagging Extension," 2024. [Online]. Available: https://developer.arm.com/-/media/Arm%20Developer%20Community/PDF/Arm_Memory_Tagging_Extension_Whitepaper.pdf.

[Avanzi2023] R. Avanzi, S. Banik, O. Dunkelman, M. Eichlseder, S. Ghosh, M. Nageler and F. Regazzoni, "The QARMAv2 Family of Tweakable Block Ciphers," 2023.

[Baldi2014] Baldi, Marco. QC-LDPC code-based cryptography. Springer Science & Business, 2014.

[Bernstein1999] D. J. Bernstein, "djbfft - an extremely fast library for floating-point convolution," 1999. [Online]. Available: https://cr.yp.to/djbfft.html.

[Bernstein2001] D. J. Bernstein, "Multidigit multiplication for mathematicians," *Advances in Applied Mathematics*, p. 1–19, 2001.

[Bernstein2007] D. J. Bernstein, "The tangent FFT," in *International Symposium on Applied Algebra, Algebraic Algorithms, and Error-Correcting Codes*, 2007.

[Božilov2020] D. Božilov, M. Eichlseder, M. Kneževic, B. Lambin, G. Leander, T. Moos, V. Nikov, S. Rasoolzadeh, Y. Todo and F. Wiemer, "PRINCEv2 - More Security for (Almost) No Overhead," 2020.

[Buhren2021] R. Buhren, H.-N. Jacob, T. Krachenfels and J.-P. Seifert, "One Glitch to Rule Them All: Fault Injection Attacks Against AMD's Secure Encrypted Virtualization," in *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security*, New York, NY, USA, 2021.

[Cass2022] S. Cass, "Top Programming Languages 2022," 2022. [Online]. Available: https://spectrum.ieee.org/top-programming-languages-2022.

[Castryck2022] W. Castryck and T. Decru, "An efficient key recovery attack on SIDH," 2022.

[Cavalcante2019] M. Cavalcante, F. Schuiki, F. Zaruba, M. Schaffner and L. Benini, "Ara: A 1-GHz+ scalable and energy-efficient RISC-V vector processor with multiprecision floating-point support in 22-nm FD-SOI," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems,* vol. 28, p. 530–543, 2019.

[Cherubin2020] Stefano Cherubin, Daniele Cattaneo, Michele Chiari, and Giovanni Agosta. 2020. Dynamic Precision Autotuning with TAFFO. ACM Trans. Archit. Code Optim. 17, 2, Article 10 (June 2020), 26 pages. https://doi.org/10.1145/3388785

[Ciobanu2013] C. B. Ciobanu, "Customizable Register Files for Multidimensional SIMD Architectures," 2013.

[Ciobanu2018] C. B. Ciobanu, G. Stramondo, C. de Laat and A. L. Varbanescu, "MAX-PolyMem: High-Bandwidth Polymorphic Parallel Memories for DFEs," *2018 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, Vancouver, BC, Canada, 2018, pp. 107-114, doi:10.1109/IPDPSW.2018.00025.

[Cooley1965] J. W. Cooley and J. W. Tukey, "An algorithm for the machine calculation of complex Fourier series," *Mathematics of computation,* vol. 19, p. 297–301, 1965.

[Dobraunig2016] C. Dobraunig, M. Eichlseder, F. Mendel and M. Schläffer, "Ascon v1. 2," *Submission to the CAESAR Competition*, vol. 5, p. 7, 2016.

[Donahue2024] P. Donahue and T. Newsome, "RISC-V Debug Specification Version 1.0," February 2024. [Online]. Available: https://github.com/riscv/riscv-debug-spec/releases/download/1.0.0-rc1-asciidoc/riscv-debug-specification.pdf.

[Duong-Ngoc2022] P. Duong-Ngoc, S. Kwon, D. Yoo and H. Lee, "Area-efficient number theoretic transform architecture for homomorphic encryption," *IEEE Transactions on Circuits and Systems I: Regular Papers,* vol. 70, p. 1270–1283, 2022.

[Franchetti2011] F. Franchetti and M. Püschel, "Fast Fourier Transform," in *Encyclopedia of Parallel Computing*, Springer, 2011.

[Fritzmann2020] T. Fritzmann, G. Sigl and J. Sepúlveda, "RISQ-V: Tightly coupled RISC-V accelerators for post-quantum cryptography," *IACR Transactions on Cryptographic Hardware and Embedded Systems,* p. 239–280, 2020.

[Fuguet2023] Fuguet, César. “HPDcache: Open-Source High-Performance L1 Data Cache for RISC-V Cores.” In Proceedings of the 20th ACM International Conference on Computing Frontiers, 377–78. Bologna Italy: ACM, 2023. https://doi.org/10.1145/3587135.3591413.

[Gerlin2022] N. Gerlin, E. Kaja, M. Bora, K. Devarajegowda, D. Stoffel, W. Kunz and W. Ecker, "Design of a Tightly-Coupled RISC-V Physical Memory Protection Unit for Online Error Detection," in *2022 IFIP/IEEE 30th International Conference on Very Large Scale Integration (VLSI-SoC)*, 2022.

[Gerlin2023] N. Gerlin, E. Kaja, F. Vargas, L. Lu, A. Breitenreiter, J. Chen, M. Ulbricht, M. Gomez, A. Tahiraga, S. Prebeck, E. Jentzsch, M. Krstić and W. Ecker, "Bits, Flips and RISCs," in *2023 26th International Symposium on Design and Diagnostics of Electronic Circuits and Systems (DDECS)*, 2023.

[GoogleProjectZero2024] Google Project Zero, "0day "In the Wild"," 2024. [Online]. Available: https://docs.google.com/spreadsheets/d/1lkNJ0uQwbeC1ZTRrxdtuPLCIl7mlUreoKfSIgajnSyY/edit#gid=1746868651.

[Han2022] Z. Han, G. Rutsch, D. Wang, B. Li, S. S. Prebeck, D. S. Lopera, K. Devarajegowda and W. Ecker, "Transformative Hardware Design Following the Model-Driven Architecture Vision," in *VLSI-SoC: Technology Advancement on SoC Design*, Cham, 2022.

[Hazy1997] L. Hazy and M. El-Tanany, "Synchronization of OFDM systems over frequency selective fading channels," in *1997 IEEE 47th Vehicular Technology Conference. Technology in Motion*, 1997.

[Intel2021] Intel, "Intel Hardware Shield - Intel Total Memory Encryption," May 2021. [Online]. Available: https://www.intel.com/content/dam/www/central-libraries/us/en/documents/white-paper-intel-tme.pdf.

[Kaja2021] E. Kaja, N. Gerlin, M. Vaddeboina, L. Rivas, S. Prebeck, Z. Han, K. Devarajegowda and W. Ecker, "Towards Fault Simulation at Mixed Register-Transfer/Gate-Level Models," in *2021 IEEE International Symposium on Defect and Fault Tolerance in VLSI and Nanotechnology Systems (DFT)*, 2021.

[Kaplan2021] D. Kaplan, J. Powell and T. Woller, "AMD Memory Encryption," October 2021. [Online]. Available: https://www.amd.com/system/files/TechDocs/memory-encryption-white-paper.pdf.

[Kim2014] Y. Kim, R. Daly, J. Kim, C. Fallin, J. H. Lee, D. Lee, C. Wilkerson, K. Lai and O. Mutlu, "Flipping bits in memory without accessing them: an experimental study of DRAM disturbance errors," *SIGARCH Comput. Archit. News*, vol. 42, p. 361–372, June 2014.

[Kim2020] S. Kim, W. Jung, J. Park and J. H. Ahn, "Accelerating number theoretic transformations for bootstrappable homomorphic encryption on gpus," in *2020 IEEE International Symposium on Workload Characterization (IISWC)*, 2020.

[Kohlbrenner2020] D. Lee, D. Kohlbrenner, S. Shinde, K. Asanovic and D. Song, “Keystone: An Open Framework for Architecting Trusted Execution Environments,” in *Proceedings of the Fifteenth European Conference on Computer Systems*, 2020.

[Koleci2023] K. Koleci, P. Mazzetti, M. Martina and G. Masera, "A Flexible NTT-Based Multiplier for Post-Quantum Cryptography," *IEEE Access,* vol. 11, p. 3338–3351, 2023.

[Kroll2017] H. Kroll, M. Korb, B. Weber, S. Willi and Q. Huang, "Maximum-Likelihood Detection for Energy-Efficient Timing Acquisition in NB-IoT," in *2017 IEEE Wireless Communications and Networking Conference Workshops (WCNCW)*, 2017.

[Kuzmanov2006] G. Kuzmanov, G. Gaydadjiev, and S. Vassiliadis. Multimedia rectangularly addressable memory. IEEE Transactions on Multimedia, pages 315–322, April 2006.

[Kwong2012] J. Kwong and M. Goel, "A high performance split-radix FFT with constant geometry architecture," in *2012 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, 2012.

[Kwong2013] J. Y. Kwong and M. Goel, "Constant geometry split radix fft". Patent US 2013/0066932 A1, 2013.

[Lee2023] J. Lee, W. Kim and J.-H. Kim, "A Programmable Crypto-Processor for National Institute of Standards and Technology Post-Quantum Cryptography Standardization Based on the RISC-V Architecture," *Sensors*, vol. 23, 2023.

[Liang2020] Z. Liang, S. Shen, Y. Shi, D. Sun, C. Zhang, G. Zhang, Y. Zhao and Z. Zhao, "Number Theoretic Transform: Generalization, Optimization, Concrete Analysis and Applications," in *Information Security and Cryptology: 16th International Conference, Inscrypt 2020, Guangzhou, China, December 11–14, 2020, Revised Selected Papers*, Berlin, 2020.

[Liang2022] Z. Liang and Y. Zhao, "Number Theoretic Transform and Its Applications in Lattice-based Cryptosystems: A Survey," 2022.

[Liljestrand2021] H. Liljestrand, T. Nyman, L. J. Gunn, J.-E. Ekberg and N. Asokan, "{PACStack}: an authenticated call stack," in *30th USENIX Security Symposium (USENIX Security 21)*, 2021.

[Lippuner2020] S. Lippuner, M. Salomon, M. Korb, M. Gautschi, T. Dellsperger, S. Altorfer, J. Rogin, S. Willi, D. Tschopp, B. Weber and Q. Huang, "A Triple-Mode Cellular IoT SoC Achieving −136.8-dBm eMTC Sensitivity," *IEEE Solid-State Circuits Letters,* vol. 3, pp. 418-421, 2020.

[Longa2016] P. Longa and M. Naehrig, "Speeding up the number theoretic transform for faster ideal lattice-based cryptography," in *Cryptology and Network Security: 15th International Conference, CANS 2016, Milan, Italy, November 14-16, 2016, Proceedings 15*, 2016.

[Nannipieri2021] P. Nannipieri, S. Di Matteo, L. Zulberti, F. Albicocchi, S. Saponara and L. Fanucci, "A RISC-V post quantum cryptography instruction set extension for number theoretic transform to speed-up CRYSTALS algorithms," *IEEE Access,* vol. 9, p. 150798–150808, 2021.

[Nasahl2021] P. Nasahl, R. Schilling, M. Werner, J. Hoogerbrugge, M. Medwed and S. Mangard, "CrypTag: Thwarting physical and logical memory vulnerabilities using cryptographically colored memory," in *Proceedings of the 2021 ACM Asia Conference on Computer and Communications Security*, 2021.

[Niederreiter1986] Niederreiter, Harald. "Knapsack-type cryptosystems and algebraic coding theory." Prob. Contr. Inform. Theory 15.2 (1986): 157-166.

[OpenHWGroup2021] OpenHW Group, "CORE-V Extension Interface," 2021. [Online]. Available: https://docs.openhwgroup.org/projects/openhw-group-core-v-xif/en/v0.2.0/x_ext.html.

[OpenHWGroup2022] OpenHW Group, "Introduction to the core-v extension interface," Apr. 2022. [Online]. Available: https://docs.openhwgroup.org/projects/ openhw-group-core-v-xif/en/latest/intro.html

[OpenHWGroup2023] OpenHW Group, "CVFPU repository," 2023. [Online]. Available: https://github.com/openhwgroup/cvfpu.

[OpenHWGroup2024] OpenHW Group, "CVA6 repository," 2024. [Online]. Available: https://github.com/openhwgroup/cva6.

[Ozcan2019] E. Ozcan and A. Aysu, "High-level synthesis of number-theoretic transform: A case study for future cryptosystems," *IEEE Embedded Systems Letters,* vol. 12, p. 133–136, 2019.

[Perotti2022] M. Perotti, M. Cavalcante, N. Wistoff, R. Andri, L. Cavigelli and L. Benini, "A “New Ara” for vector computing: an open source highly efficient RISC-V V 1.0 vector processor design," in *2022 IEEE 33rd International Conference on Application-specific Systems, Architectures and Processors (ASAP)*, 2022.

[Nashimoto2021] S. Nashimoto, D. Suzuki, R. Ueno and N. Homma, "Bypassing Isolated Execution on RISC-V using Side-Channel-Assisted Fault-Injection and Its Countermeasure," *IACR Transactions on Cryptographic Hardware and Embedded Systems*, vol. 2022, p. 28–68, November 2021.

[Roscian2013] C. Roscian, A. Sarafianos, J.-M. Dutertre and A. Tria, "Fault Model Analysis of Laser-Induced Faults in SRAM Memory Cells," in *2013 Workshop on Fault Diagnosis and Tolerance in Cryptography*, 2013.

[RVI2021] RISC-V International, "RISC-V V Vector Extension (Version 1.0)," 2021. [Online]. Available: https://github.com/riscv/riscv-v-spec/releases/download/v1.0/riscv-v-spec-1.0.pdf.

[RVI2024] RISC-V International, "RISC-V ELF psABI Document," 2024. [Online]. Available: https://github.com/riscv-non-isa/riscv-elf-psabi-doc/.

[RV-SS-LP-TG2023] RISC-V Shadow-stack and Landing-pads Task Group, "Shadow Stack and Landing Pads (v0.3.1)," 2023. [Online]. Available: https://github.com/riscv/riscv-cfi/.

[RV-SS-LP-TG2024] RISC-V Shadow-stack and Landing-pads Task Group, "Shadow Stack and Landing Pads (0036ff2)," 2024. [Online]. Available: https://github.com/riscv/riscv-cfi/.

[Sangiovanni-Vincentelli2012] A. Sangiovanni-Vincentelli, W. Damm and R. Passerone, "Taming Dr. Frankenstein: Contract-Based Design for Cyber-Physical Systems*," *European Journal of Control,* vol. 18, pp. 217-238, 2012.

[Satriawan2023] A. Satriawan, I. Syafalni, R. Mareta, I. Anshori, W. Shalannanda and A. Barra, "Conceptual Review on Number Theoretic Transform and Comprehensive Review on Its Implementations," *IEEE Access,* 2023.

[Tran2020] D. D. Tran, K. Grüttner, F. Oppenheimer and W. Nebel, "Timing Contracts and Monitors for Safety Relevant Controller Design in IEC 61499," in *2020 25th IEEE International Conference on Emerging Technologies and Factory Automation (ETFA)*, 2020.

[VE-VIDES] "VE-VIDES - Designmethoden und HW/SW-Co-Verifikation für die eindeutige Identifizierbarkeit von Elektronikkomponenten," [Online]. Available: https://www.elektronikforschung.de/projekte/ve-vides (German only).

[Vizcaino2023] P. Vizcaino, F. Mantovani, R. Ferrer and J. Labarta, "Acceleration with long vector architectures: Implementation and evaluation of the FFT kernel on NEC SX-Aurora and RISC-V vector extension," *Concurrency and Computation: Practice and Experience,* vol. 35, p. e7424, 2023.

[Walther2000] J. S. Walther, "The Story of Unified Cordic," *Journal of VLSI signal processing systems for signal, image and video technology,* vol. 25, pp. 107-112, 2000.

[Waterman2021] A. Waterman, Asanovic and J. Hauser, "The RISC-V Instruction Set Manual, Volume II: Privileged Architecture, Document Version 20211203," December 2021. [Online]. Available: https://drive.google.com/file/d/1EMip5dZlnypTk7pt4WWUKmtjUKTOkBqh/view?usp=drive_link.