



| | |
|---|---|
| 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 | **D3.2** |
| Title | **Safety & Security modules prototype implementation** |
| Type of deliverable | Report |
| Deadline | 30.04.2025 |
| Creation Date | 31.01.2025 |
| Dissemiation Level | ISOLDE - public |

Authors
Marcus Borrmann, Erik Kraft, Marcel Medwed - NXP-AT
Daniel Gracia Pérez, Sylvain Girbal - TRT
Stefan Wallentowitz - HM
Holger Blasum, Darshak Sheladiya - 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, Daniel Cebrián - FENTISS
Wolfgang Ecker, Endri Kaja, Ares Tahiraga, Jad Al Halabi - IFX
Mladen Berekovic, Rainer Buchty, Amrit Sharma Poudel, Nguyen-Hoang-Hai Pham, Saleh Mulhem - UzL
Luigi Ghionda, Emanuele Parisi, Yvan Tortorella - UNIBO
Andrea Galimberti. D.Zoni, W.Fornaciari - POLIMI
Alexandru Pușcașu, Cătălin Ciobanu, Mihai Gologanu - IMT
Sven Mehlhop, Jörg Walter - 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, George Suciu - BEIA

Michael Gautschi, Reza Ghanaatian, Stefan Lippuner - ACP

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)

Coordinator: Sylvain Girbal, TRT, sylvain.girbal@thalesgroup.com
Reviewers: Francesco Conti, UNIBO, f.conti@unibo.it
Jan Kaštil, Codasip, jan.kastil@codasip.com

# 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 documents refine the architecture and describe the prototype implementations of the hardware modules related to safety and security, previously introduced in Deliverable D3.1. These modules have been developed within the work package WP3 “Accelerators and Extensions” of the ISOLDE project, more particularly in tasks T3.1 and T3.3.

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

1. Extensions enhancing the safety and security of RISC-V systems (T3.1)
2. Extensions that allow monitoring of the foundational core and accelerators (T3.3)

For each safety / security extension, this document contains refined architecture description since deliverable D3.1, reminding the purpose of the extension, where in the system is it integrated, and which other systems it is connected to. We then provide a more detailed description on the hardware module internals, describing the prototype that allow the hardware module to be evaluated / 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” is providing 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 deliverable covering the final implementations of the extensions (D3.4, D3.5 in M33). The components described in this deliverable are aiming at different maturity levels and aiming for different certifiability.

# 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 D3.2, titled "Safety & Security modules prototype implementation", is to refine the architecture description of each safety / security hardware module presented in Deliverable D3.1, and provide implementation details and early results prior to the integration in ISOLDE use-cases in Work-package WP5.

This document is intended for public release and includes the hardware extensions' updated architecture and design specifications. These updated architecture and design specifications encapsulate the extensions' technical research , implementation and developmental progress.

## 2.2 Purpose and Scope

This document is an intermediary deliverable prior to Deliverable D3.4 that will report the final implementation of safety and security hardware modules, due in early 2026 and finalizing the bottom-up approach to address all ongoing safety / security activities comprehensively.

WP3 organizes the project into distinct tasks covering different domains, with various partners contributing to each. Section 3 focuses on Task T3.1: Safety & Security Modules, and Task T3.3: Monitoring Infrastructure that relates to either safety or cyber-security hardware IPs.

Each subsection of Section 3 relates to one of such tasks, describing the ecosystem and the participating partners. The remainder of these subsections contains in-depth technical information about each specific extension providing:

- an **IP card** of the hardware module summarizing technical information.
- **general information** about the context the hardware module will be used in, providing a scope beyond the specific module itself.
- The **purpose and scope** of this specific hardware module, providing a brief high-level description of the module purpose.
- A **refined architecture description** pointing out the updates since the initial architecture description provided in Deliverable D3.1, and covering its placement in the system with block diagrams.
- A list of the both control and data **interfaces** to the other IPs this specific hardware component is connected to, as well as dependencies to existing IPs.
- A description of any **instruction set architecture (ISA) specialization** associated with the hardware module, as well as potential hardware abstraction layers (HAL) or high-level API.
- Finally, the **evaluation prototype** that will report the implementation and developmental progress related to the safety / security module, as well as strategies for clocking, resetting, power management, verification and debugging, possibly including a test-bench prototype

and early results prior to the integration to WP5 use-cases.

Within WP5 and its associated deliverables, we will merge these safety / security hardware modules with selected core features from WP2 and software IPs from WP4 providing software support.

# 3 Safety and Security extensions

## 3.1 Safety and Security Modules

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.

| IP | Lead Partner | Domain | Dependencies | Licensing |
|---|---|---|---|---|
| IEE (3.1.1) | NXP-AT | Security | None | Proprietary closed source |
| BCFI (3.1.2) | NXP-AT | Security | RV32I processor, IEE (3.1.1) | Proprietary closed source |
| CTM (3.1.3) | NXP-AT | Security | RV32I processor, IEE (3.1.1) | Proprietary closed source |
| EMI (3.1.4) | NXP-AT | Security | RV32I processor, IEE (3.1.1) | Proprietary closed source |
| FCFI (3.1.5) | NXP-AT | Security | RV32I processor | Proprietary closed source |
| CA-PMC (3.1.6) | TRT | Safety | CA-PMC-IF (3.2.1) | Proprietary closed source |
| Memory support for Bytecode VMs (3.1.7) | HM | Security | None | Permissive open source |
| SafeTI (3.1.8) | BSC | Safety | None | Permissive open source |
| Safety & Security Control Unit (3.1.9) | IFX | Safety, Security | None | Permissive open source |
| Safety Island (3.1.10) | UZL | Safety | CVA6 core | Permissive open source |
| RoT (3.1.11) | UNIBO | Security | CVA6, OpenTitan | Permissive open source |
| High-Perf. Cache Analysis (3.1.12.1) | SYSGO | Security | CVA6, CEA cache (Tristan) | N/A |

Table 3.1: Overview of Task 3.1 contributions

### 3.1.1 Inline Encryption Engine (IEE) – NXP-AT

#### 3.1.1.1 IP Card

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

#### 3.1.1.2 General Information

Traditionally, isolation of different workloads and their associated sensitive assets is achieved using the processor's privilege mode and logical memory isolation (e.g., using memory protection or management units). The logical memory isolation primitive limits access triggered by specific

privilege modes and is configured by a trusted software running in a higher privilege mode, for example, the firmware running in machine mode or the operating system running in supervisor mode. However, research has shown that this logical isolation is insufficient against a range of physical attacks techniques (e.g., probing the memory bus, injecting faults flipping bits in memory [25, 19], or injecting faults affecting the memory protection configuration by the operating system [22]).

Adding a memory encryption engine to the system, which encrypts all data before it reaches the external bus, helps to mitigate some of these attacks. The resulting ciphertexts usually depend on the physical address (used as tweak) in addition to the secret key to mitigate attacks exchanging the encrypted data words. If a pure encryption scheme is used as a primitive for memory encryption, then the confidentiality of data on the external bus and in memory is protected, and controlled modifications are harder to achieve. If the verification of the data integrity is required, an authenticated encryption scheme must be used, but then also the memory overhead will be higher. Memory encryption engines help mitigate many physical attacks, but they also impose considerable area and latency overhead, while not increasing resilience against logical attacks.

As this deliverable provides updated information about the IP state since Deliverable D 3.1, please refer to Section 3.1.1 in Deliverable D 3.1 for additional information about the Inline Encryption Engine (IEE) module.

#### 3.1.1.3 Purpose and Scope

The IEE module adds a tweakable memory encryption engine based on a low-latency cipher to the base 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 extensions contributed by NXP-AT, which add defense mechanisms against various logical attacks. For more information about these extensions, please refer to the description of the Backward-Edge Control Flow Integrity (BCFI; see Section 3.1.2), Cryptographically Tagged Memory (CTM; see Section 3.1.3), and Enclave Memory Isolation (EMI; see Section 3.1.4) extension provided by NXP-AT. Systems that already including a tweakable memory encryption engine especially benefit from this approach, as adding the mentioned defenses against logical attacks comes at little additional cost.

#### 3.1.1.4 Refined architecture description

As we were not able to find an integration opportunity for our IPs within the official ISOLDE demonstrators, we instead collaborate with the TRISTAN project, integrating our features into the NFC_FPGA demonstrator (WI6.4.1). Hence, the architecture of this extension was updated to match the new integration target where an extended CV32E40S RISC-V core will be used.

In Figure 3.1, the parts of the CV32E40S core modified by the IEE extension are highlighted in orange. In the following paragraphs, the added and modified modules are described in more detail.

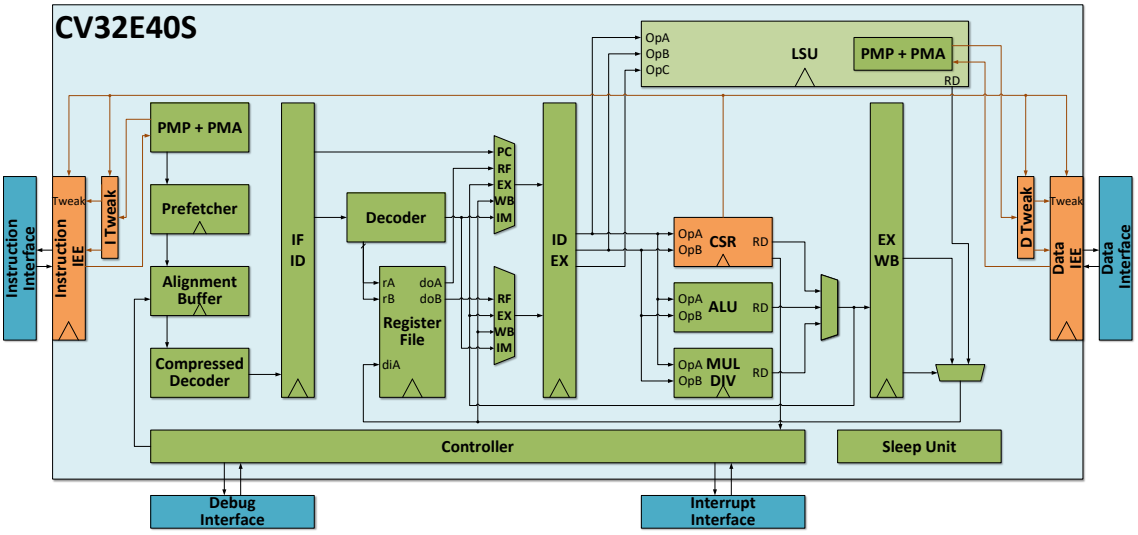


Figure 3.1: Architecture of the IEE extension

### CSR

The IEE extension adds CSRs representing the encryption key (`ieekey0` - `ieekey3`) and the end of the encrypted memory region (`mieeencend` and its read-only alias `uieeencend`) to the existing CSR unit. Since these CSRs provide the encryption key and define the end of the encrypted memory region, they influence the memory encryption. Therefore, it is essential to ensure that the pipeline is flushed when they are written. Otherwise, instruction fetches or regular memory access in previous pipeline stages may use a wrong configuration. Therefore, the IEE extension also adapts the logic in the CSR unit to ensure the pipeline is flushed when the introduced CSRs are written to.

### I Tweak

The I Tweak module provides the active encryption tweak to the Instruction IEE module. It is connected to the core's OBI instruction interface and the CSRs, so that it has access to all information required to derive the encryption tweak. The encryption tweak generated by the I Tweak module consists of the access address, access privilege level and the enclave tweak provided by NXP-AT's EMI extension.

### Instruction IEE

The Instruction IEE module is attached to the core's OBI instruction interface and provides a OBI instruction interface to external components. Additionally, it is connected to the CSRs introduced by the IEE extension and the I Tweak module providing the required encryption tweak. It decrypts all instruction fetches from the encrypted memory region, encryption is not required as the the

core never issues write requests through the OBI instruction interface.

The design of the Instruction IEE module is detailed in Figure 3.2. At its core, it consists of a low-latency 32-bit decryption function. The decryption function relies on a 32-bit keyed unscrambling primitive (inspired by the SRAM scrambler primitive from openTitan) and the KeystreamGen module deriving the required key stream. The KeystreamGen module derives the key for the unscrambling primitive from the processor's inline encryption key (stored in the `ieekey0` - `ieekey3` CSRs) and the tweak provided by the I Tweak module. During the address phase of a new OBI request (`obi_instr_if.s_req.req` asserted), the key required for decryption is derived and latched together with the request address for later use in the response phase. When the availability of the response is indicated and the last request addresses was within the encrypted memory region (defined by the design parameter `IEE_ENCRYPTION_BASE` and the `mieeencend` CSR), then the `rdata` signals of the payload are replaced with the unscrambled version. All other signals of the external OBI interface connected to the memory (`obi_instr_ext_if`) are passed through to the core's OBI interface (`obi_instr_if`). Since the key derived by the KeystreamGen module is always ready during the response phase and the unscrambling is purely combinatorial, the cycle count for instruction fetch requests remains the same regardless of the usage of the Instruction IEE module. However, the additional logic increases the delay of the critical path. If the achievable frequency fails to meet requirements, more pipeline stages must be added. To reduce stalls in this scenario, caused by waiting for unscrambling results, a cache with prefetching can be added. This allows the cache subsystem to fetch, unscramble, and buffer instructions that are likely to be requested by the processor in the future.

Figure 3.3 details the design of the KeystreamGen module. It consists of two 64-bit PRINCEv2 [5] instances with a 1-stage pipeline. The PRINCEv2 Sbox was replaced with the ORTHROS [2] Sbox to reduce the latency of PRINCEv2. The tweak provided by the I Tweak module is split and routed to the data input of the two PRINCEv2 instances. The PRINCEv2 instances derive the 128-bit key required for the unscrambling primitive from the inline encryption engine key and the provided tweak. This key is divided into four round keys, which are XORed with the data before and after the initial and final inverse substitution-permutation network ($\text{SPN}^{-1}$) rounds of the unscrambling primitive, as illustrated in Figure 3.4. Finally, Figure 3.5 details the construction of the inverse SPN. It consists of a key addition layer, followed by the inverse SKINNY [4] MixColumns operation, the inverse SKINNY ShiftRows operation, the inverse PRINCEv2 S-box, and finally another key addition layer.

The custom low-latency 32-bit encryption and decryption function should be considered as extensive scrambling for now, since a thorough security analysis has yet to be conducted. However, further security improvements are investigated, and a paper conducting a thorough security analysis of the scheme is currently in progress.

#### D Tweak

The D Tweak module provides the active encryption tweak to the Data IEE module. It is connected to the OBI instruction interface and the CSRs, so that it has access to all information required to derive the encryption tweak. The encryption tweak generated by the D Tweak module consists of the access address, access privilege level, the BCFI access identifier provided by NXP-AT's BCFI extension, the pointer color provided by NXP-AT's CTM extension and the enclave tweak provided by NXP-AT's EMI extension.

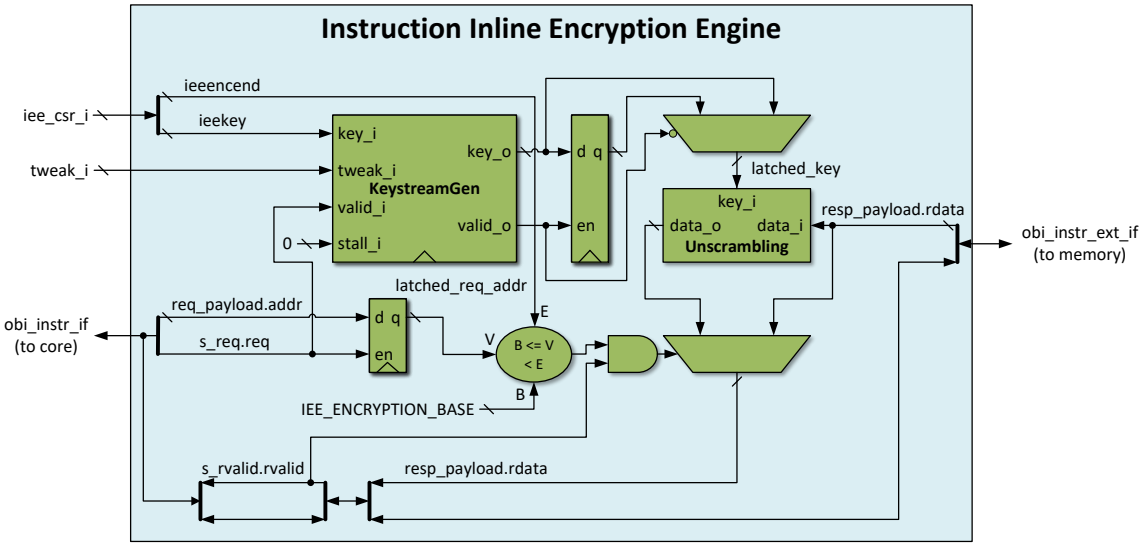


Figure 3.2: Architecture of the Instruction IEE Module

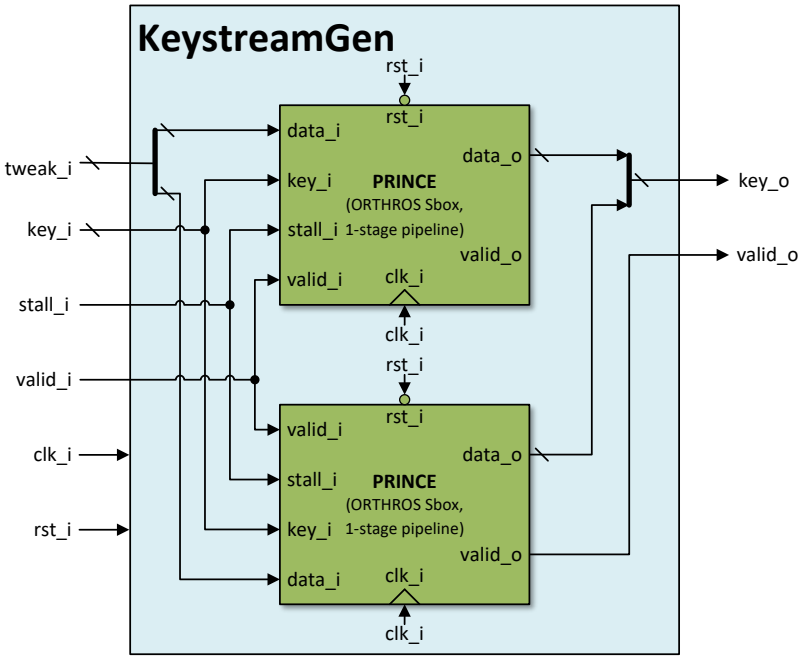


Figure 3.3: Architecture of the IEE KeystreamGen module

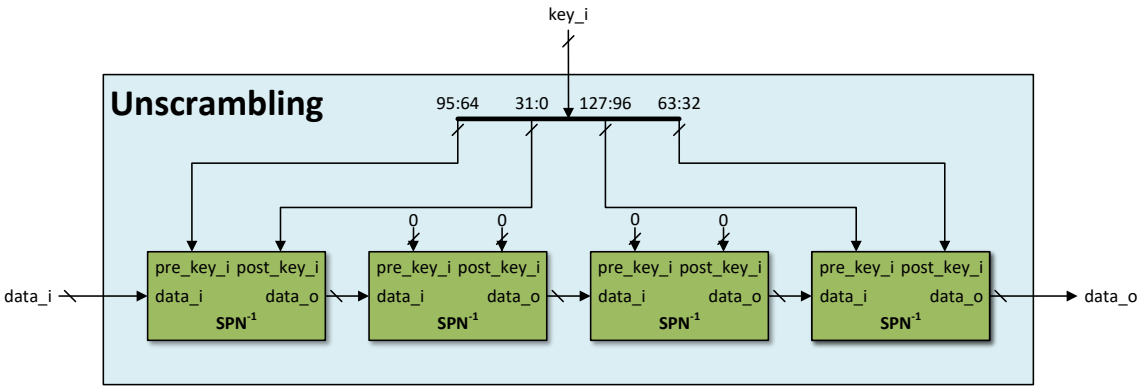


Figure 3.4: Architecture of the IEE Unscrambling module

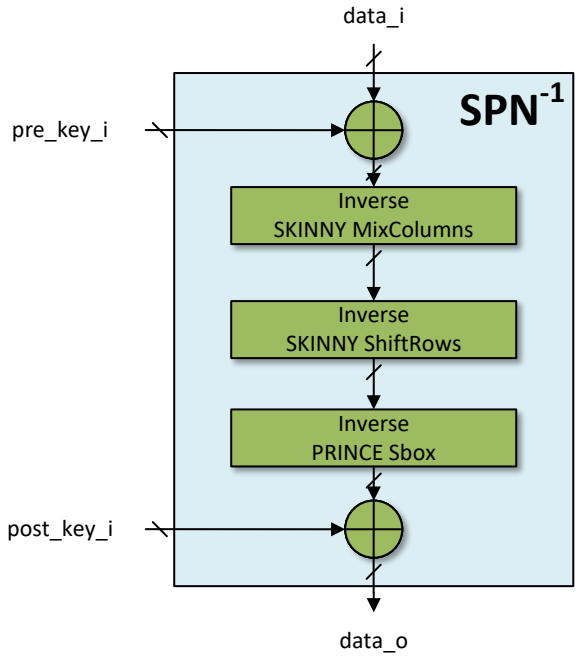


Figure 3.5: Architecture of the IEE inverse substitution–permutation network ($\text{SPN}^{-1}$)

### Data IEE

The Data IEE module is attached to the core's OBI data interface and provides a OBI data interface to external components. Additionally, it is connected to the CSRs introduced by the IEE extension and the D Tweak module providing the required encryption tweak. It encrypts data to be written to the encrypted memory region before passing it to the external OBI data interface. Similarly, it decrypts data read from the encrypted memory region attached to the external OBI data interface

before passing it to the core. Unlike the Instruction IEE, where the latency and throughput of fetch requests are unaffected by the inline encryption, memory requests issued through the Data IEE may require a few additional cycles in the worst case (depending on the the duration of the OBI transactions and the memory access sequence). These additional cycles result from the Data IEE module not being able to issue another request via the OBI data interface, as it is still occupied with a request related to the more complex handling of stores: For fullword stores, the key from the KeystreamGen module must already be ready during the address phase of the OBI transfer as it is required for the encryption of the write data. For subword stores, an additional read access is required before the store can be performed. This is due to the low-latency cipher's fixed block size of 32 bits, which requires the corresponding word to be read and decrypted, the affected bytes to be updated, and the updated word to be re-encrypted before issuing the store request.

The Data IEE module incorporates a more complex control logic and a pipelined data path to efficiently manage the more complex handling of stores to the encrypted region. The Data IEE module requires the low-latency 32-bit decryption function described in the Instruction IEE paragraph, along with its encryption counterpart. The difference between the encryption and decryption functions is that the encryption function uses a scrambling primitive instead of an unscrambling one. The scrambling primitive employs the regular SPN instead of the inverse (i.e., inverted operations in reversed order), and the order of the round keys is reversed.

#### 3.1.1.5 Interfaces

The IEE extension integrates modules between the CV32E40S core's OBI interfaces (one for fetching instructions and another for accessing data) and the external memory, enabling encryption or decryption of data before it is written to external memory or passed to the core. Additionally, it extends the ISA of the CV32E40S core with new CSRs that store the encryption key and the boundary of the encrypted memory region. Finally, as mentioned in Section 3.1.1.3, the IEE module can optionally be connected to the other security extensions provided by NXP-AT to increase protection against logical attacks.

#### 3.1.1.6 ISA specialization

The specification of the ISA extension did not change since deliverable D3.1. The IEE extension adds CSRs representing the encryption key (`ieekey0` - `ieekey3`) and the end of the encrypted memory region (`mieeencend` and its read-only alias `uieeencend`). The base address of the encrypted memory region can be configured using the `IEE_ENCRYPTION_BASE` design parameter.

The assigned CSR addresses are still temporary, we will provide the final assignment in deliverable D3.4 after the demonstrator integration is finished.

#### 3.1.1.7 Evaluation prototype

The evaluation prototype consists of a CV32E40S core enhanced with NXP-AT's security extensions, including the here described IEE extension, embedded in one of the following evaluation environments:

- *CORE-V Verification Environment*
  The enhanced core replaces stock core in the CV32E40S testbench. The resulting testbench can be evaluated using RTL simulation (tested with the Xcelium simulator).

- *CORE-V MCU*
  The enhanced core replaces the CV32E40P core in the CORE-V MCU design. The resulting design can be synthesized to a Nexys A7-100T or Genesys 2 FPGA board and evaluated there (tested with Genesys 2).

A suite of self-checking tests has been written for the IEE extension. These tests can be and were executed in both of the supported environments.

The evaluation prototype was tested on the Genesys 2 FPGA board (featuring a XC7K325T-2FFG900C FPGA IC). The design was synthesized using Vivado v2024.1, targeting a frequency of 50 MHz. The IEE extension increased the usage of lookup tables by 65.15 % (absolute: 5901) and registers by 47.13 % (absolute: 1623) compared to the base CV32E40S core.

Initial PPA analysis using a 16 nm process with worst timing, targeting a clock frequency of 500 MHz (the highest frequency before area increases significantly), shows a 77.79 % (absolute: 37295 GE) area increase caused by the IEE extension compared to the base CV32E40S core. The IEE extension increased the combinatorial delay of the critical path by 16 % compared to the base CV32E40S core.

While these increases are significant, memory encryption is often required for security-sensitive products.

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

#### 3.1.2.1 IP Card

| | | |
|---|---|---|
| **Basic Info** | | |
| *IP name* | Backward-Edge Control Flow Integrity (BCFI) | |
| *License* | Proprietary Closed Source | |
| *Repository* | N/A | |
| **Architecture** | | |
| *Clock* | Number of clock domains | 1 |
| | Synchronous with system | Y |
| | Clock generated internally | N |
| *Ctrl Interface* | ISA extension? | Y (complies partly with Zicfiss extension [23]) |
| | Memory mapped? | N |
| | Protocol | N/A |
| | Address Map | N/A |
| *Initiator Interface* | Protocol | |
| | Cached? | N/A |
| | IOMMU? | |
| *Interrupts* | Interrupts | N |
| **Microarchitecture** | | |
| *Parametrization* | Parametric no. cores? | N |
| | Parameteric config? | Y |
| *Programmability* | Contains programmable cores? | N |
| | ISA | N/A |
| **Software** | | |
| *Compiler* | Requires specialized compiler? | Y |
| | Compiler repository | N/A |
| *Hardware Abstraction Layer* | N/A | |
| *High-level API* | Is there a high-level API/SDK? | N |
| | SDK repository | N/A |
| | Is there a domain-specific compiler? | N |
| **Integration** | | |
| *IP Distribution* | Manifest type (if any) | N |
| | Standalone simulation? | N |
| | (if standalone sim) SW requirements? | N/A |
| | Integration documented / examples? | N/A |
| *Synthesis* | Is the IP synthesizable? | Y |
| | FPGA synthesis example available? | Y |
| | ASIC synthesis example available? | N |
| *Simulation* | Closed-source simulation? | Y (Xcelium) |
| | Open-source simulation? | N |
| *Evaluation* | PPA results available? | Y |

#### 3.1.2.2 General Information

According to Google Project Zero, memory corruption vulnerabilities are the most used path to gain unintended remote control over digital devices [15]. 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 [7], 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 only with 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.

The exploitation of memory safety vulnerabilities may enable an attacker to modify the program behavior and take over control. For example, the attacker could replace function return addresses spilled on the stack with addresses containing attack gadgets. Such attacks, which aim to modify the backward-edge control flow of programs, are among the most common and known ones. A typical memory safety issue is a buffer overflow vulnerability, which occurs when user input is not correctly sanitized. For example, missing bound checks that otherwise ensure the user input has a valid length, could allow an attacker to craft an input that overwrites the return address on the stack when (part of) the input is copied to internal data structures.

Modern systems feature a variety of countermeasures, like stack canaries and Address Space Layout Randomization (ASLR) to mitigate these attacks. However, these countermeasures have weaknesses and can be bypassed. For example, both are vulnerable to information disclosure attacks. A more recent countermeasure, not suffering from these weaknesses, are shadow stacks. In a system with this countermeasure, return addresses are stored on an isolated shadow stack in addition to the regular stack. The isolation guarantees that only special instructions can access the shadow stack while regular memory operations cannot access it. Therefore, an attacker can only overwrite return addresses on the regular stack, but not the duplicates 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 to detect attacks. RISC-V International ratified a shadow stack design for RISC-V, the *Zicfiss* extension [24].

As this deliverable provides updated information about the IP state since Deliverable D 3.1, please refer to Section 3.1.2 in Deliverable D 3.1 for additional information about the Backward-Edge Control Flow Integrity (BCFI) module.

#### 3.1.2.3 Purpose and Scope

While the Zicfiss extension requires a MMU to isolate the shadow stack, our BCFI extension isolates the shadow stack cryptographically using a tweakable memory encryption engine like NXP-AT's IEE extension (see Section 3.1.1). The cryptographical isolation is achieved by using distinct encryption tweaks for memory accesses resulting from shadow stack operations, differentiating them from regular memory accesses. Further, a hash value uniquely identifying the current return address chain is stored in an isolated CSR. The previous (intermediate) hash values are spilled to the shadow stack instead of the actual return addresses, preventing the replay of encrypted return addresses (the topmost hash value depends on all previous ones). When returning from a function, the current and previous hash value are passed to the inverse hash function to reconstruct the return address. For more information about the concept of our BCFI extension, which is based on the PACStack paper [20], see the detailed description in Section 3.1.2.2 of Deliverable D3.1.

The BCFI extension can be added with little overhead to systems already including a tweakable memory encryption engine (i.e., needed to fulfill the security requirements) and does not rely on the presence of an MMU. Further, it offers better protection against physical attacks.

#### 3.1.2.4 Refined architecture description

As we were not able to find an integration opportunity for our IPs within the official ISOLDE demonstrators, we instead collaborate with the TRISTAN project, integrating our features into the NFC_FPGA demonstrator (WI6.4.1). Hence, the architecture of this extension was updated to match the new integration target where an extended CV32E40S RISC-V core will be used.

Figure 3.6 illustrates a CV32E40S core enhanced with the IEE module from Section 3.1.1 and the BCFI extension. The components of the core that have been modified by the BCFI extension are highlighted in orange. In the following paragraphs, the added and modified modules are described in more detail.

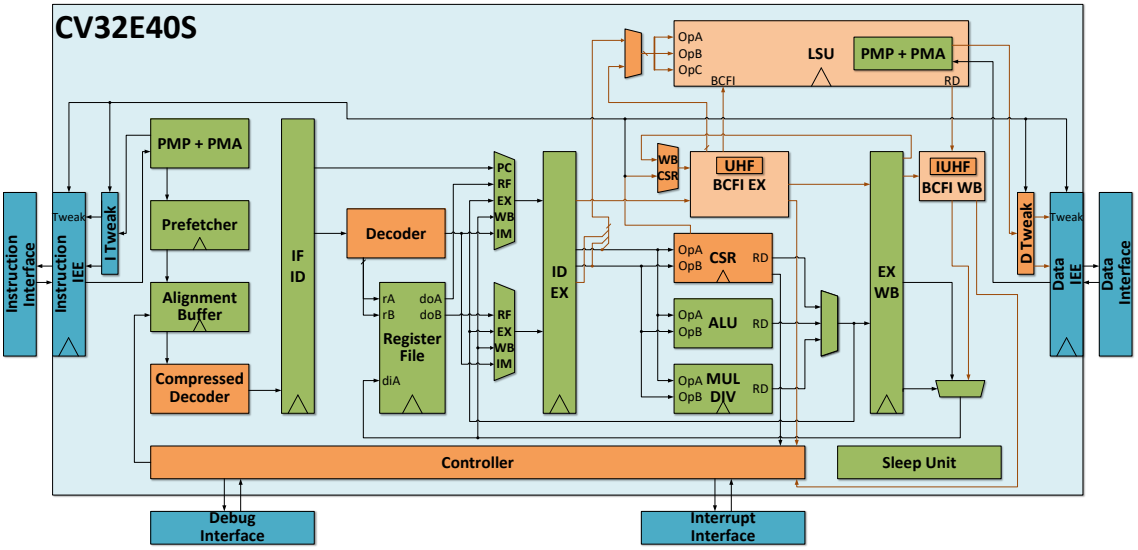


Figure 3.6: Architecture of the BCFI extension

#### Decoders

The BCFI extension introduces new instructions for manipulating the shadow stack (see Section 3.1.2.5 of Deliverable D3.1 for more details). Hence, the extension adapts the existing compressed and regular decoder.

#### CSR

The configuration and state of the BCFI extension are stored in CSRs. Hence, the extension adds the required CSRs or CSR fields (see Section 3.1.2.5 of Deliverable D3.1 for more details) to the

CSR unit. Note that the `senvcfg` CSR can be ignored as CV32E40S processor does not support supervisor mode. The added CSRs are implicitly accessed by the BCFI execution units in the execute and writeback stage. Additionally, they may be explicitly accessed by CSR read or write instructions. The BCFI extension adjusts the forwarding logic to ensure that results for implicit or explicit reads are passed from the writeback stage when necessary.

**BCFI execution unit (execute)**

Most of the functionality of the BCFI extension is implemented as a new execution unit in the execute stage. However, computing the inverse hash using the IUHF module (needed to restore the return address during executing `sslw` and `sspopchk`) can only be done in the writeback stage, as it requires the load of the previous hash value to be completed. The BCFI execution unit in the execute stage receives information from the ID-EX pipeline register and the current BCFI state from the corresponding CSRs (either directly or forwarded), performs the supported operations, and outputs an updated state and further control signals (e.g., security violations passed to the controller). As some BCFI operations (i.e., `sspush`, `sslw` or `sspopchk`) need to interact with memory, the execution unit is also connected to the LSU. To update the hash value representing the return address chain (done during executing `sspush`), a universal hash function (UHF) module is included in the BCFI execution unit. The UHF is a scaled down version (i.e., computations are performed over a smaller binary field) of the GHASH function [9, Sec. 6.4] used in the GCM encryption mode.

**BCFI execution unit (writeback)**

As mentioned in the previous paragraph, the inverse hash calculation can only be performed in the writeback stage as it requires the LSU to provide the result of the necessary memory read. If the BCFI execution unit in the writeback stage is active (only for the `sslw` and `sspopchk` instructions), then it uses the data from the EX-WB pipeline register and the output from the LSU to compute the inverse hash representing the reconstructed return address. Next, the result is either passed as destination register content (in case of `sslw`) or it is compared with the provided return address (in case of `sspopchk`). If the comparisons fails, then a security violation is passed to the controller. Finally, the execution unit roles back the topmost hash value to the previous one by setting the `sstca` CSR to the data received from the LSU with the help of the CSR unit.

**LSU**

Some BCFI operations (i.e., `sspush`, `sslw` or `sspopchk`) need to interact with the memory. Hence, the extension modifies the logic in the execute stage so that the memory access address and write data can be routed from the BCFI execution unit to the LSU. Further, the BCFI unit passes another signal to the LSU that indicates if the memory access request is a shadow stack transaction. This signal is passed on by the LSU to the D Tweak unit as part of the internal OBI data request signals.

**D Tweak**

The D Tweak module provides the active encryption tweak to the Data IEE module (see Section 3.1.1). It is connected to the OBI data interface and the CSRs, so that it has access to all information needed to derive the required encryption tweak. The CTM extension incorporates logic into the D Tweak unit to determine if an OBI data request is a shadow stack access by inspecting the corresponding signal added by the LSU to the request. The D Tweak module

encodes this information as one bit of the encryption tweak passed to the Data IEE module.

**Controller**

The extension adapts the existing controller implementation so that a software check exception is raised if one of the BCFI execution units report a security violation. Additionally, it extends the forwarding logic as explained in the CSR paragraph.

#### 3.1.2.5 Interfaces

The BCFI extension is tightly integrated in the CV32E40S processor and requires NXP-AT's IEE extension.

#### 3.1.2.6 ISA specialization

The specification of the ISA extension did not change since Deliverable D3.1. The BCFI extension adds instructions to manipulate data on the shadow stack allowing to push (`sspush`, `c.sspush`), pop (`sspopchk`, `c.sspopchk`), load (`sslw`), and atomically swap words (`ssamoswap`). Further, it adds instructions for receiving (`ssrdp`) and increasing the shadow stack pointer value (`ssincp`, `c.ssincp`). To enable the configuration of the extension, the `mseccfg` and `menvcfg` CSRs are modified. The universal hash function is parametrized with the `ssuhfx` and `ssuhfxinv` CSRs. The shadow stack pointer is stored in the `ssp` and the topmost hash value in the `sstca` CSR.

The assigned CSR addresses are still temporary, we will provide the final assignment in deliverable D3.4 after the demonstrator integration is finished.

#### 3.1.2.7 Evaluation prototype

The evaluation prototype and available evaluation environments are the same as described in Section 3.1.1.7 for NXP-AT's IEE module.

A suite of self-checking tests has been written for the BCFI extension. These tests can be and were executed in both of the supported environments.

The evaluation prototype was tested on the Genesys 2 FPGA board (featuring a XC7K325T-2FFG900C FPGA IC). The design was synthesized using Vivado v2024.1, targeting a frequency of 50 MHz. The BCFI extension increased the usage of lookup tables by 8.08 % (absolute: 732) and registers by 7.98 % (absolute: 275) compared to the base CV32E40S core.

Initial PPA analysis using a 16 nm process with worst timing, targeting a processor clock frequency of 500 MHz (the highest frequency before area increases significantly), shows a 9.13 % (absolute: 4377 GE) area increase caused by the BCFI extension compared to the base CV32E40S core.

### 3.1.3 Cryptographically Tagged Memory (CTM) – NXP-AT

#### 3.1.3.1 IP Card

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

#### 3.1.3.2 General Information

As this deliverable provides updated information about the IP state since Deliverable D 3.1, please refer to Section 3.1.4 in Deliverable D 3.1 for additional information about the Cryptographically Tagged Memory (CTM) module.

#### 3.1.3.3 Purpose and Scope

As explained in Section 3.1.2, memory corruption vulnerabilities are the most used path to gain remote control over computing devices. To mitigate attacks exploiting memory safety issues to modify function return addresses spilled on the stack, we introduced the BCFI extension in Section 3.1.2. Further, a device may isolate different tasks logically (i.e., using memory protection or management units) or cryptographically (using memory encryption engines, see Section 3.1.1 for more information). However, there exist other fine-grained assets within a task's stack, heap, or 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

These sensitive assets can not be sufficiently protected with the mentioned technologies and hence an additional mechanism is required.

Memory tagging can be used to mitigate this security 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).

Note that the colors must be stored in addition to the rest of the data, increasing the overall memory usage. Therefore, existing memory tagging implementations, like Armv8.5-A MTE [1], allocate only few bits for the colors, making them unsuitable for security purposes. Cryptographically tagged memory [21] 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, assuming a memory encryption engine is available, this technique allows to use more bits for the color without increasing the memory overhead. The major behavioral difference to the original scheme is that an attacker can still misuse pointers to access sensitive memory blocks. However, it is much harder to perform a successful attack as access triggered from a misused pointer will not be useful:

- Suppose misusing a pointer triggers an out-of-bounds read of a memory location associated with a different color. 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 to a memory location associated with a different color. Then, the color of the genuine pointer will not match the one of the misused pointer. Hence, the attacker cannot modify the data in a controlled way.

Until now, cryptographically tagged memory has only been implemented on 64-bit processors, where unused bits in the virtual address space are utilized to store the tags. The CTM extension adds cryptographically tagged memory to 32-bit processors without requiring a memory management unit. The smaller address space poses additional challenges as usually no unused bits in pointers are available, and therefore, a different strategy to embed the colors in tagged pointers is needed.

#### 3.1.3.4 Refined architecture description

As we were not able to find an integration opportunity for our IPs within the official ISOLDE demonstrators, we instead collaborate with the TRISTAN project, integrating our features into the NFC_FPGA demonstrator (WI6.4.1). Hence, the architecture of this extension was updated to match the new integration target where an extended CV32E40S RISC-V core will be used.

Figure 3.7 illustrates a CV32E40S core enhanced with the IEE module from Section 3.1.1 and the CTM extension. The components of the core that have been modified by the CTM extension are highlighted in orange. In the following paragraphs, the added and modified modules are described in more detail.

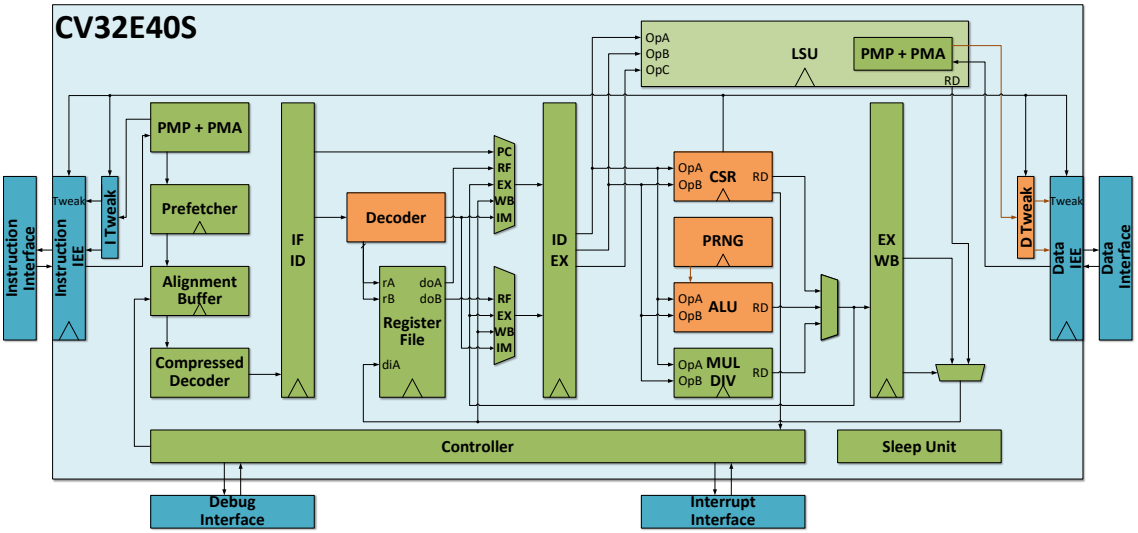


Figure 3.7: Architecture of the CTM extension

#### Decoder

The CTM extension introduces two new instructions to the decoder for tagging (`ctmtag`) and untagging (`ctmuntag`) pointers (see Section 3.1.4.4 in Deliverable D 3.1 for more information).

#### CSR

The extension adds the `CTME` field to the `mseccfg` CSR, allowing software to disable CTM (see Section 3.1.4.4 in Deliverable D 3.1 for more information).

#### ALU

The extension introduces two new operations to the ALU. The `ctmtag` operation adds a random color generated by a pseudorandom number generator (PRNG) to the pointer in the source register, while the `ctmuntag` operation removes the color from the pointer in the source register.

#### PRNG

The CTM extension adds a hardware PRNG to the core, or reuses an existing one. The PRNG is needed for generating the colors used to tag pointers. The entropy of the used PRNG should at least match the color field size so that the probability of neighboring memory blocks with matching colors is as low as possible. Depending on the threat model different PRNG designs can be used:

- If the assumed attacker can only inject faults, then a PRNG based on a linear-feedback shift register (LFSR) is sufficient, as the attacker cannot observe its outputs or state.
- If the assumed attacker can only perform logical attacks (allowing the attacker to observe the PRNG outputs), then a continuously-clocked LFSR with a sufficiently large cycle could be used. An attacker would then need to predict how often the LFSR will have been clocked at the next color generation to leak the next color, which is hard for a logical attacker and complex systems.
- Finally, if the assumed attacker is very advanced, then a cryptographically secure PRNG (CSPRNG) design must be used. For example, designs described by the NIST SP 800-90A Rev. 1 standard [3] could be used.

#### D Tweak

The D Tweak module provides the active encryption tweak to the Data IEE module (see Section 3.1.1). It is connected to the OBI data interface and the CSRs, so that it has access to all information needed to derive the required encryption tweak. The CTM extension adds logic to the D Tweak unit for extracting the color from tagged pointers, which is passed to the Data IEE module as part of the encryption tweak. Figure 3.8 shows the operation of the color extraction logic. The color extraction logic separates the color if CTM is enabled and the pointer is tagged (indicated by the bit at position `CTM_EXPLICIT_SELECTION_BIT`). Furthermore, if the pointer is tagged, the color is removed from the address, and an offset is added to the result before it is passed on as the access address to the Data IEE. This addressing scheme allows the system integrator to configure a region in the address space that can be protected using CTM, provided the MSB of addresses is not needed and can be utilized to indicate tagged pointers. The maximum size of the protected region depends on the number of bits used for the color.

### 3.1.3.5 Interfaces

The CTM extension is tightly integrated in the CV32E40S processor and requires NXP-AT's IEE extension.

### 3.1.3.6 ISA specialization

The specification of the ISA extension did not change since Deliverable D3.1. The CTM extension adds two instructions for tagging (`ctmtag`) and untagging (`ctmuntag`) pointers. Further, it adds the `CTME` field to the `mseccfg` CSR, allowing software to disable CTM.

### 3.1.3.7 Evaluation prototype

The evaluation prototype and available evaluation environments are the same as described in Section 3.1.1.3 for NXP-AT's IEE module.

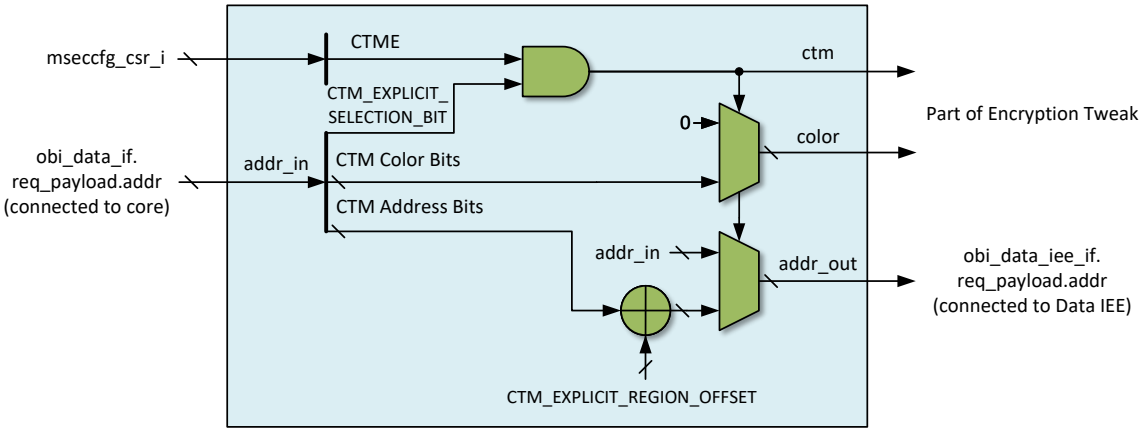


Figure 3.8: CTM color extraction logic

A suite of self-checking tests has been written for the CTM extension. These tests can be and were executed in both of the supported environments.

The evaluation prototype was tested on the Genesys 2 FPGA board (featuring a XC7K325T-2FFG900C FPGA IC). The design was synthesized using Vivado v2024.1, targeting a frequency of 50 MHz. The CTM extension increased the usage of lookup tables by 0.99 % (absolute: 90) and registers by 1.02 % (absolute: 35) compared to the base CV32E40S core.

Initial PPA analysis using a 16 nm process with worst timing, targeting a processor clock frequency of 500 MHz (the highest frequency before area increases significantly), shows a 0.03 % (absolute: 14 GE) area increase caused by the CTM extension compared to the base CV32E40S core.

### 3.1.4 Enclave Memory Isolation (EMI) – NXP-AT

#### 3.1.4.1 IP Card

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

#### 3.1.4.2 General Information

Modern systems often run a range of workloads on one physical general-purpose processor as it is cost-efficient to reuse existing infrastructure. These workloads may include sensitive assets and their vendors may not trust each other. Hence, to guarantee the confidentiality and integrity

of the assets, it is necessary to isolate the workloads from each other. Traditionally, this isolation is realized by dedicated hardware components (memory protection and management units) that enforce that tasks in lower privilege modes can only access allowed regions. The operating system, which runs in a higher privilege mode, sets up the memory regions and configures suitable access permissions for each task.

However, research has shown that this logical isolation is insufficient against a range of physical attacks techniques (e.g., probing the memory bus, injecting faults flipping bits in memory [25, 19], or injecting faults affecting the memory protection configuration by the operating system [22]). 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.

#### 3.1.4.3 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 Inline Encryption Engine (IEE) module (see Section 3.1.1). Further, the design of the EMI module mitigates the impact of fault attacks including skipping instructions during the context switch. 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 physical attacks). These stronger guarantees benefit use cases requiring strong isolation between workloads like trusted execution environments.

#### 3.1.4.4 Refined architecture description

As we were not able to find an integration opportunity for our IPs within the official ISOLDE demonstrators, we instead collaborate with the TRISTAN project, integrating our features into the NFC_FPGA demonstrator (WI6.4.1). Hence, the architecture of this extension was updated to match the new integration target where an extended CV32E40S RISC-V core will be used.

Figure 3.9 illustrates a CV32E40S core enhanced with the IEE module from Section 3.1.1 and the EMI extension. The components of the core that have been modified by the EMI extension are highlighted in orange. In the following paragraphs, the added and modified modules are described in more detail.

##### CSR

The EMI extension includes new CSRs, which are combined to form an enclave tweak (see Figure 3.10). The extension adds these CSRs to the existing CSR unit. As these CSRs influence memory encryption, it must be ensured that the pipeline is flushed when they are written. Otherwise, instruction fetches or regular memory access in previous pipeline stages may use a wrong tweak leading to wrongly decrypted data. Therefore, the EMI extension also extends the existing logic in the CSR unit that determines for which CSR writes the pipeline should be flushed.

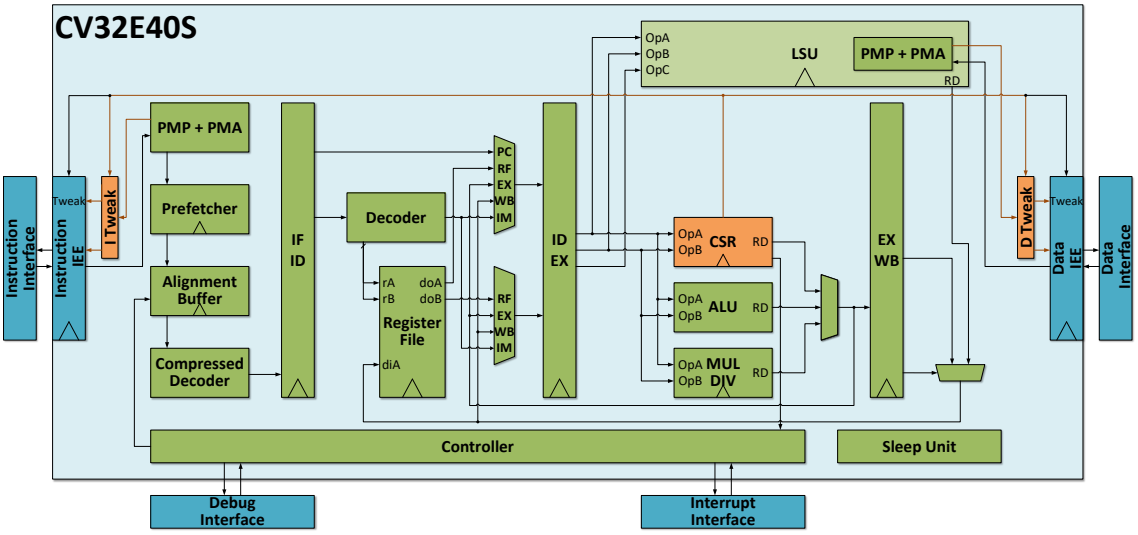


Figure 3.9: Architecture of the EMI extension

#### I Tweak and D Tweak

The enclave tweaks, which are passed to NXP-AT's IEE modules and influence the memory encryption, are constructed as detailed in the previous deliverable and illustrated in Figure 3.10. The I Tweak and D Tweak modules provide the active encryption tweak to the Instruction and Data IEE module (see Section 3.1.1). They are connected to the core's OBI interfaces and the CSRs, so that they have access to all information needed to derive the required encryption tweak. As the CV32E40S core does not support S mode and does not include caches, the S mode modifier CSRs are not needed. The EMI extension adds logic to the I Tweak and D Tweak unit combining the modifiers stored in CSRs to form enclave tweaks, which are passed to the IEE modules as part of the encryption tweak.

### 3.1.4.5 Interfaces

The EMI extension is tightly integrated in the CV32E40S processor and internally connected to NXP-AT's IEE modules.

### 3.1.4.6 ISA specialization

The specification of the ISA extension did not change since Deliverable D3.1. The EMI extension adds multiple CSRs (`mfetchmod{0,1}`, `ufetchmod{0,1}`, `mloadmod{0,1}`, `uloadmod{0,1}`, `mstoremod{0,1}`, `ustoremod{0,1}`) representing modifiers that are combined to an enclave tweak depending on the current privilege level and access type (load, store, fetch), as shown in Figure 3.10.

The assigned CSR addresses are still temporary, we will provide the final assignment in Deliverable D3.4 after the demonstrator integration is finished.

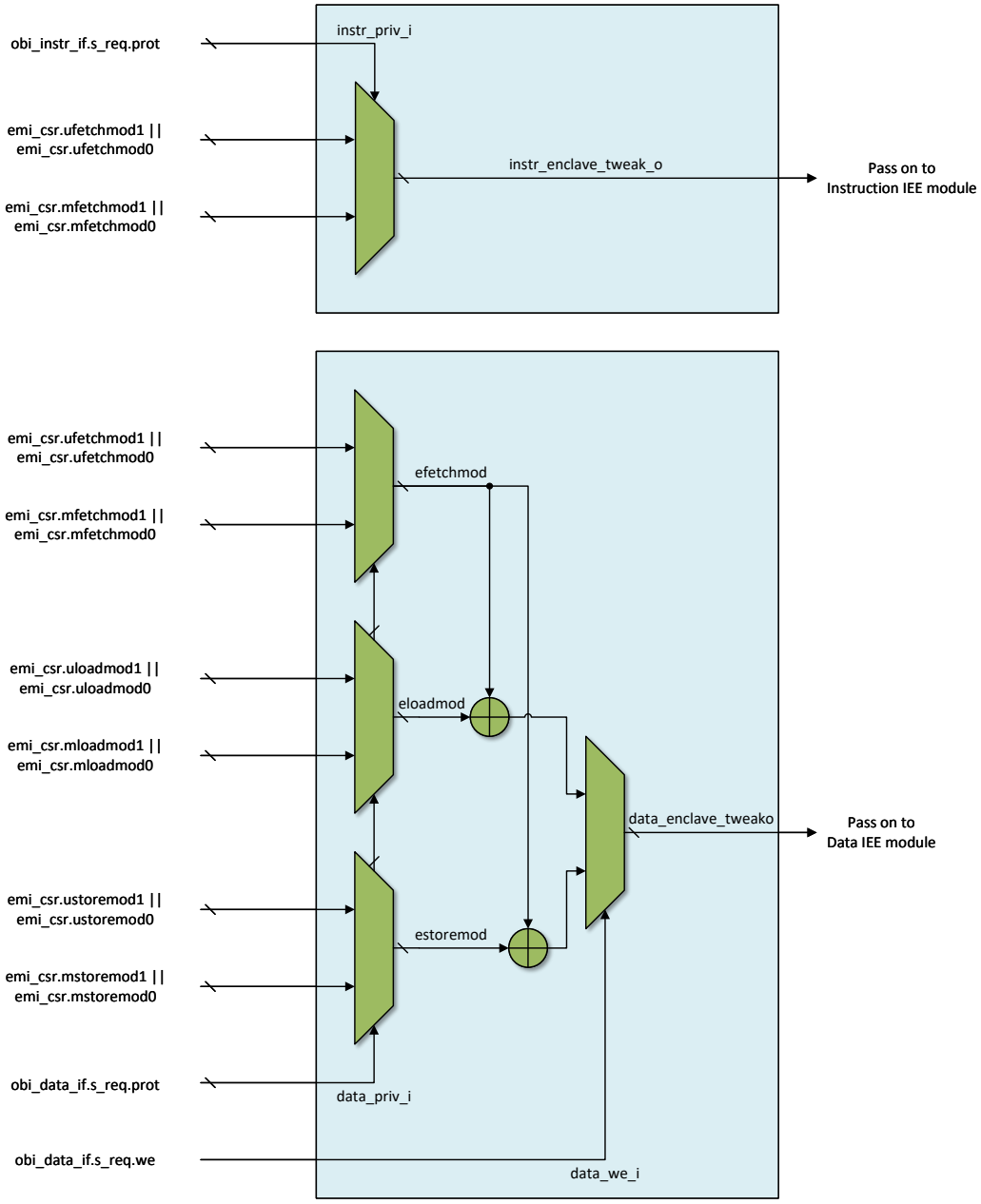


Figure 3.10: EMI enclave tweak formation logic

### 3.1.4.7 Evaluation prototype

The evaluation prototype and available evaluation environments are the same as described in Section 3.1.1.3 for NXP-AT's IEE module.

A suite of self-checking tests has been written for the EMI extension. These tests can be executed in both of the supported environments.

The evaluation prototype was tested on the Genesys 2 FPGA board (featuring a XC7K325T-2FFG900C FPGA IC). The design was synthesized using Vivado v2024.1, targeting a frequency of 50 MHz. The EMI extension increased the usage of lookup tables by 3.60 % (absolute: 326) and registers by 10.74 % (absolute: 370) compared to the base CV32E40S core.

Initial PPA analysis using a 16 nm process with worst timing, targeting a processor clock frequency of 500 MHz (the highest frequency before area increases significantly), shows a 7.91 % (absolute: 3764 GE) area increase caused by the EMI extension compared to the base CV32E40S core.

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

#### 3.1.5.1 IP Card

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

#### 3.1.5.2 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. The injected disturbances can cause different effects in a pro-

cessor depending on the affected logic and form of the fault signal. For example, the processor may skip instructions or execute altered instructions. Attackers exploit these effects to bypass security measures, e.g., by skipping their configuration or enablement. Research demonstrated various successful attacks exploiting instruction skips introduced by fault attacks. Among those are bypassing signature verification to load malicious firmware [6] or skipping the reconfiguration of memory protection units to gain access to protected data [22].

#### 3.1.5.3 Purpose and Scope

The FCFI extension 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 check exception is raised. Hence, the FCFI extension enables the detection of fault injection attacks that aim to modify the instruction stream.

Even with sophisticated fault injection attacks, such as single-spot laser fault injection, it is challenging to arbitrarily alter multiple instructions. The potential outcomes are probabilistic and their probabilities depend on the configuration of the fault injection setup and the technology node used to manufacture the IC. Further, considering a single-spot laser fault injection setup, an attacker would need to realign the laser to prepare the next fault injection, which is not feasible before the next checksum check. Consequently, even for advanced attackers it is hard to hide introduced errors by trying to modify subsequent instructions before the next checksum check occurs.

Further, the FCFI extension also includes instructions that allow implementing landing pads. Hence, it can also detect modifications of forward-edge control flow transfers (indirect calls) by logical attacks. Note that also the *Zicfilp* RISC-V extension offers similar landing pads, but it cannot be efficiently combined with FCFI's instruction stream integrity feature.

However, the FCFI extension cannot ensure the integrity of the data memory. Hence, it also cannot mitigate attacks aiming to modify backward-edge control flow transfers (i.e., altering function return addresses on the stack). To reduce the attack surface further, it can be combined with NXP-AT's BCFI module described in Section 3.1.2.

#### 3.1.5.4 Refined architecture description

The FCFI extension extends the ISA of the base processor to enable two high-level features:

- Instruction stream integrity (mitigating physical fault injection attacks)
- Landing pads for forward-edge control flow integrity (mitigating logical attacks aiming to overwrite function pointers)

After reset, both features are disabled. Software must activate them using dedicated instructions (`csjal`, `csjalr`) or CSR fields (`ucsstate.LPE`). To better understand the necessary changes in the CV32E40S core, first the functionality of the high-level features is described.

The instruction stream integrity feature requires a 16-bit CRC module allowing to update a CRC value incrementally. Terminator instructions, which end a basic block (a sequence of instructions that executes sequentially), initialize the CRC value to the PC value of the next basic block. In the context of the FCFI extension, the following instructions of the rv32ic architecture and the

extension itself are terminating instructions: `b{eq,ne,lt,ge,ltu,geu}`, `c.b{eqz,nez}`, `jal`, `jalr`, `c.j`, `c.jr`, `c.jal`, `c.jalr`, `csb{eq,ne,lt,ge,ltu,geu}`, `cscheck`, `c.cscheck`, `csjal`, and `csjalr`. If further extensions present in the target processor include instructions that change the control flow, then support for them needs to be added to the FCFI extension.

During the execution of a basic block, the CRC value is continuously updated with the encoding of the instruction to be executed next. Instead of updating the CRC value using instruction encodings, the decoder output signals could have been used. This alternative implementation would have the advantage of also ensuring the integrity of the decoder signals. However, it would also require the compiler to know details about the microarchitecture, leading to significant disadvantages (maintenance, IP protection).

When a terminator instruction is executed, then the FCFI extension compares the current CRC value with a reference value computed by a software tool. Dedicated instructions (`csref`, `c.csref`) allow to pass the bits of the reference CRC values to the FCFI extension during the execution of a basic block. It is assumed that code sections cannot be modified by an attacker, preventing the direct modification of reference values. When the computed CRC value does not match the reference value or any other of the checks detailed in Section 3.1.5.6 fails, then a software check exception is raised. The minimum required security level (i.e., the minimum number of reference bits compared at CRC checks) can be enforced using the `ucscontrol.NEED11` or `ucscontrol.NEED5` CSR fields.

During trapping to machine mode, the processor suspends the FCFI extension by setting the `ucsstate.EXCP` bit. Suspending the extension allows software to save the FCFI state of the interrupted task and restore the state of a different task. While the FCFI extension is suspended, no updates of its state are performed. When returning from machine to user mode (during execution of the `mret` instruction), the `ucsstate.EXCP` bit is cleared. The FCFI extension is also suspended during debug mode (independently of the `ucsstate.EXCP` bit), so that it does not interfere with debug implementations using the processor's pipeline (i.e., execution-based debug implementations).

Additionally, the FCFI extension implements a watchdog to ensure that CRC checks occur after a configurable number of retired instructions. This watchdog can be optionally activated (`ucswdog.WDE`). If activated, then the compiler must insert dedicated instructions (`cscheck`) in long basic blocks to avoid the expiry of the watchdog. The watchdog's counter is decremented for every non-terminating instruction and a software check exception is raised if the counter reaches zero.

The landing pad functionality can operate together or independent of the instruction stream integrity feature. In the first case, during an indirect call instruction, the CRC value is initialized with a label value (provided using `cslabel` or `c.cslabel`) instead of the target address. The label provided by the compiler should be derived from the function signature of genuine target functions and the software tool computing the reference CRCs must take the changed initialization into account. Then, the CRC values are influenced by the label and the next CRC check will fail if an attacker diverted the control flow to a function with a wrong signature. If a function is invoked both directly and indirectly, distinct entry points should be used for each type of call to ensure compatibility.

When the landing pad functionality operates independently of the instruction stream integrity feature (i.e., `ucsstate.LPE` is set), then the first instruction after indirect calls must be a landing pad (`cscheck` or `c.cscheck`). Otherwise, a software check exception is raised. Optionally, finer-grained protection can be achieved by enforcing that labels are provided before indirect calls using the `cslabel` or `c.cslabel` instruction (`ucscontrol.NEEDLPLABEL`). The reference labels must then be provided with the landing pad instructions (`cscheck` or `c.cscheck`), which check if the current label matches the reference label.

In Figure 3.11, the parts of the CV32E40S core modified by the FCFI extension are highlighted in orange. In the following paragraphs, the added and modified modules are described in more detail.

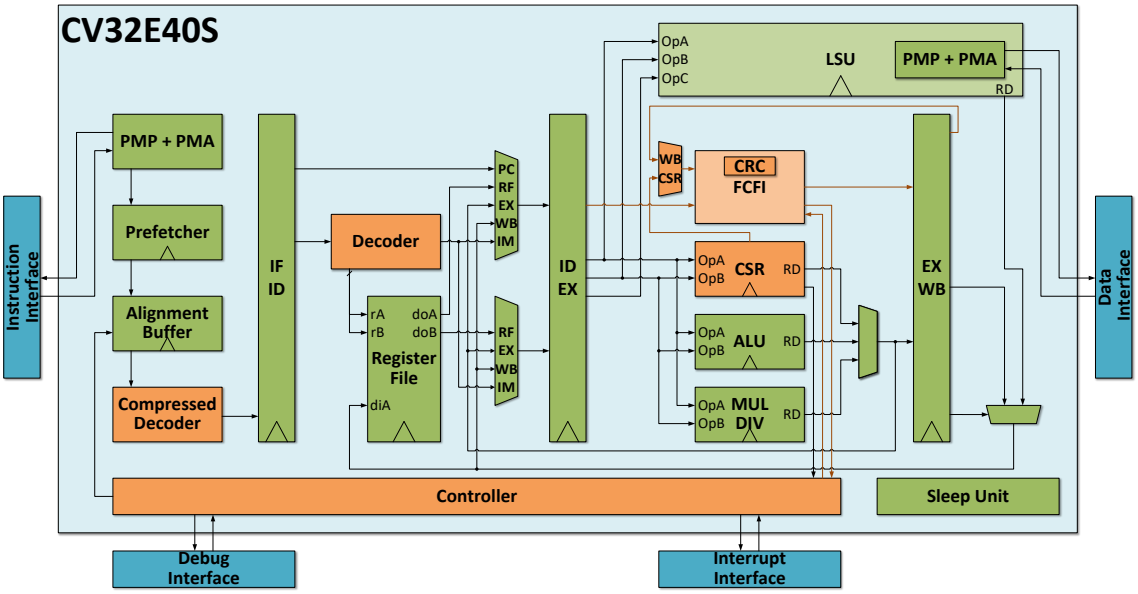


Figure 3.11: Architecture of the FCFI extension

### Decoders

The FCFI extension introduces new instructions and modifies the behavior of existing instructions, as detailed in Section 3.1.5.6. Hence, the extension adapts the existing compressed and regular decoder. The introduced branch (`csb{eq,ne,lt,ge,ltu,geu}`) and call (`csjal, csjalr`) instructions use a new instruction format. Therefore, the FCFI extension also modifies the unit computing the target PC of instructions changing the control flow and the destination register selection in the instruction decode stage.

### CSR

The configuration and state of the FCFI extension is stored in user-accessible CSRs. Hence, the extension adds the necessary CSRs (see Section 3.1.5.6) to the CSR unit. These CSRs are

implicitly accessed by the FCFI execution unit. Additionally, they may be explicitly accessed by CSR read or write instructions. The extension adjusts the forwarding logic to ensure that results for implicit or explicit reads are passed from the writeback stage when necessary. If explicit and implicit writes occur in parallel (e.g., there is a CSR write instruction in the execute stage and the FCFI execution unit outputs an updated CRC value), then the writes are merged in the execute stage.

**FCFI execution unit**

The core functionality of the FCFI extension is implemented as a new execution unit in the execute stage. This unit receives information about the instruction to be executed from the ID-EX pipeline register and the current FCFI state from the corresponding CSRs (either directly or forwarded), performs the necessary operations (e.g., updating the CRC value or decrementing the watchdog counter), and outputs an updated state and further control signals (e.g., security violations passed to the controller).

**Controller**

The extension adapts the existing controller implementation so that an exception is raised if the FCFI execution unit reports a security violation. Additionally, it extends the forwarding logic as explained in the CSR paragraph. Further, the modified controller triggers the setting of the `ucsstate.EXCP` bit during trap entry, which suspends FCFI operations. Finally, the modified controller suspends the FCFI extension during debug entry and debug mode. This is necessary as debug operations are executed using the CV32E40S's pipeline (i.e., execution-based debug implementation). Hence, suspending FCFI ensures that debug mode does not interfere with the FCFI state, and that the instruction stream integrity feature does not interfere with debug operations. As an authenticated debug user has anyway full system access, suspending FCFI does not impact security.

Given that one of the FCFI extension's objectives is to mitigate fault attacks, the implementation of the FCFI module and its critical components must be security-hardened. This includes duplicating CSRs that contain the state and configuration, as well as related signals and the next PC logic.

#### 3.1.5.5 Interfaces

The FCFI extension is tightly integrated in the CV32E40S processor.

#### 3.1.5.6 ISA specialization

The FCFI extension modifies the ISA of the CV32E40S core. In the following, the modified and new design parameters, CSRs, instructions, and exceptions are introduced.

**Design Parameters**
Table 3.2 introduces the design parameters impacting the FCFI extension.

**CSRs**
In Table 3.3, the CSRs added by the FCFI extensions are described. The introduced CSRs are accessible by all privilege modes and are zero initialized at reset. The final addresses will be

| Parameter | Default Value | Function |
|---|---|---|
| FCFI_ENABLE | 1 | Determines if the FCFI extension is included in the design. |

Table 3.2: Design parameters of the FCFI extension

assigned once the integration with the processor used in the targeted demonstrator is finished and provided with Deliverable D3.4.

| CSR | Index | Field | Bits | Function |
|---|---|---|---|---|
| ucsstate | TBD | EXCP | 25 | Set upon entering a trap and cleared upon exiting (suspends FCFI) |
| | | LPE | 18 | Landing pads are enabled |
| | | LPX | 17 | Landing pad is expected |
| | | CSE | 16 | Instruction stream integrity is enabled |
| | | CURCRC | 15...0 | Current CRC value |
| ucsrefcrc | TBD | VALID4 | 18 | Bits 3...0 of ucsrefcrc.VALUE are valid |
| | | VALID5 | 17 | Bits 4...0 of ucsrefcrc.VALUE are valid |
| | | VALID11 | 16 | Bits 15...5 of ucsrefcrc.VALUE are valid |
| | | VALUE | 15...0 | Collected reference CRC value |
| ucslabel | TBD | VALID5 | 17 | Bits 4...0 of ucslabel.VALUE are valid |
| | | VALID11 | 16 | Bits 15..5 of ucslabel.VALUE are valid |
| | | VALUE | 15...0 | Label value |
| ucswdog | TBD | WDE | 16 | Watchdog for ensuring regular CRC checks is enabled |
| | | INIT | 15...8 | Initial value for watchdog |
| | | CUR | 7...0 | Current value of watchdog |
| ucsinitmask | TBD | MASK | 15..0 | XOR-mask for initializing the current CRC |
| ucscontrol | TBD | NEEDLPLABEL | 5 | Enforce usage of labels for landing pads |
| | | NEED11 | 1 | Need at least 11 reference bits |
| | | NEED5 | 0 | Need at least 5 reference bits |

Table 3.3: CSRs introduced by the FCFI extension

### Instructions

Table 3.4 outlines the added and modified instructions by the FCFI extension. In the context of the FCFI extension, the terms 'active' and 'enabled' are not used synonymously. For example, FCFI can be enabled but suspended (ucsstate.EXCP bit set) during the execution of a trap handler.

The distinction between jumps, calls and returns relies on the RISC-V ABI.

Table 3.4: Instructions added and modified by the FCFI extension

| Instruction | FCFI-related action |
|---|---|

| | |
|---|---|
| `csref` (16 reference bits)<br>`c.csref` (11 reference bits)<br>`cslabel` (16 label bit)<br>`c.cslabel` (11 label bits) | These new instructions provide reference or label bits. When the instruction stream integrity feature is active, the current CRC value is updated with the instruction's encoding (excluding the reference or label bits). |
| `b{eq,ne,lt,ge,ltu,geu}`,<br>`jal (rd == x0)`,<br>`jalr (rs1 != x1\|x5, rd == x0)` | FCFI extends the existing RISC-V branches and jumps to update and verify the current CRC value when the instruction stream integrity feature is active. If the CRC check passes, the CRC value is reset to the target address of the control transfer XORed with `ucsinitmask`, and the watchdog is reset. If the CRC check fails, a software check exception is raised. |
| `csb{eq,ne,lt,ge,ltu,geu}` | These new branch instructions provide 5 reference bits in addition to the previously described behavior of regular branches. To encode the reference bits, the branch range has been reduced to $\pm 512\,\mathrm{B}$. Depending on the enforced security level (`ucscontrol.NEEDx`), these instructions can load both the required reference bits and perform the branch, thereby reducing code size overhead. |
| `jal (rd != x0)` | FCFI extends existing RISC-V direct calls to update and verify the CRC value, as previously described for branches. Additionally, regular direct calls indicate control transfers to unprotected functions. Hence, if the instruction stream integrity feature is active, the `ucsstate.CSE` bit is backed up and cleared. |
| `jalr (rd != x0)` | FCFI extends existing RISC-V indirect calls, similar to the direct calls described above, but additionally requires a landing pad instruction at the target address when the landing pad feature is enabled. |
| `csjal, csjalr` | These new instructions perform calls to protected functions. They provide 4 reference bits, update, verify, and reset the CRC value when the instruction stream integrity feature is active, as described earlier for branches. If the label value is valid, then the CRC value is reset to the label instead of the target address. Additionally, they backup the current `ucsstate.CSE` bit when the instruction stream integrity feature is active and then set the bit. The destination register selection is limited to `x1` or `x5` to free up encoding space for the reference bits. |
| `jalr (rs1 == x1\|x5, rd == x0)` | FCFI extends existing RISC-V returns so that they update and verify the CRC value when the instruction stream integrity feature is active. Additionally, they restore the `ucsstate.CSE` bit if FCFI is not suspended. Finally, if the instruction stream integrity feature is active after restoring the state, the CRC value is reset to the return address XORed with `ucsinitmask`. |

| | |
|---|---|
| `cscheck` (16 reference bits) `c.cscheck` (11 reference bits) | When the instruction stream integrity feature is active, then these new instructions provide 16 or 11 reference bits, update, verify, and reset the CRC value to the address of the next instruction XORed with `ucsinitmask`. Otherwise, when only the landing pad feature is active, they use the reference bits as expected label value, verify the current label value, and raise a software check exception if there is a mismatch. |
| *Machine mode trap* | FCFI modifies the trap entry behavior to suspend its operation (sets `ucsstate.EXCP`), allowing its state to be saved and restored by privileged software. |
| `mret` | FCFI extends RISC-V trap return instructions to clear the `ucsstate.EXCP` bit, resuming its operation. |
| *Any other instruction* | When the instruction stream integrity feature is active, FCFI updates the current CRC value using the encoding of each instruction to be executed. |
| Additionally, for any instruction that is not a landing pad (all except `c.cscheck` and `cscheck`) | FCFI raises a software check exception when the landing pad feature is enabled and a landing pad is expected (i.e., after an indirect call). |
| *Additionally, for any instruction that is not a terminating instruction* | When the instruction stream integrity feature is active, FCFI raises a software check exception if the watchdog value is zero. Otherwise, it decreases the watchdog value. |
| *Additionally, for any instruction introduced by FCFI (`cs*`)* | FCFI raises a software check exception if it is suspended and an FCFI instruction is executed, indicating an attempt to bypass FCFI by suspending it. |

**Exceptions**
In Table 3.5, the exceptions extended by the FCFI extension are described.

#### 3.1.5.7 Evaluation prototype

The evaluation prototype and available evaluation environments are the same as described in Section 3.1.1.3 for NXP-AT's IEE module.

A suite of self-checking tests has been written for the FCFI extension. These tests can be and were executed in both of the supported environments.

The evaluation prototype was tested on the Genesys 2 FPGA board (featuring a XC7K325T-2FFG900C FPGA IC). The design was synthesized using Vivado v2024.1, targeting a frequency of 50 MHz. The FCFI extension increased the usage of lookup tables by 8.15 % (absolute: 738) and registers by 11.88 % (absolute: 409) compared to the base CV32E40S core.

Initial PPA analysis using a 16 nm process with worst timing, targeting a processor clock frequency of 500 MHz (the highest frequency before area increases significantly), shows a 10.64 % (absolute: 5102 GE) area increase caused by the FCFI extension compared to the base CV32E40S core.

| Exception | Code | Description |
|---|---|---|
| `SoftwareCheckException` | 18 | Synchronous exception which is triggered when there are violations of checks and assertions regarding the integrity of software assets. The exact cause can be determined by examining the `xTVAL` register:<br>• 20: FCFI detected an instruction stream integrity violation (CRC mismatch).<br>• 21: The FCFI watchdog expired as no check was performed within the configured timeout.<br>• 22: FCFI encountered reference bits although it is disabled and not suspended.<br>• 23: FCFI could not perform the instruction stream integrity check due to missing reference bits.<br>• 24: FCFI could not perform the instruction stream integrity check due to insufficient reference bits.<br>• 25: FCFI expected a landing pad, but there was none.<br>• 26: FCFI expected a landing pad with an explicit label, but there was no label.<br>• 27: FCFI detected a landing pad size mismatch.<br>• 28: FCFI detected a landing pad value mismatch.<br>• 29: FCFI encountered an unexpected `cscheck` instruction.<br>• 30: The processor attempted to execute a FCFI instruction while FCFI is suspened. |

Table 3.5: Exceptions extended by the FCFI extension

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

#### 3.1.6.1 IP Card

| | | |
|---|---|---|
| **Basic Info** | | |
| *IP name* | Context-Aware Performance Monitor Counter (CA-PMC) | |
| *License* | Open-source (SolderPad Hardware License v0.51) | |
| *Repository* | https://github.com/ThalesGroup/cva6-context-aware-monitoring.git | |
| **Architecture** | | |
| *Clock* | Number of clock domains | 1 |
| | Synchronous with system | Y |
| | Clock generated internally | N |
| *Ctrl Interface* | ISA extension? | N |
| | Memory mapped? | Y (see also CA-PMC-IF in Section 3.2.1.5) |
| | Protocol | AXI or AXI Lite |
| | Address Map | Y (see Section 3.2.1.5) |
| *Initiator Interface* | Protocol | N/A |
| | Cached? | N |
| | IOMMU? | N |
| *Interrupts* | Interrupts | Y (generates 1 interrupt) |
| **Microarchitecture** | | |
| *Parametrization* | Parametric no. cores? | N |
| | Parameteric config? | Y |
| *Programmability* | Contains programmable cores? | N/A |
| | ISA | N/A |
| **Software** | | |
| *Compiler* | Requires specialized compiler? | N |
| | Compiler repository | - |
| *Hardware Abstraction Layer* | N/A | |
| *High-level API* | Is there a high-level API/SDK? | N |
| | SDK repository | N/A |
| | Is there a domain-specific compiler? | N |
| **Integration** | | |
| *IP Distribution* | Manifest type (if any) | (to be defined) |
| | Standalone simulation? | N |
| | (if standalone sim) SW requirements? | - |
| | Integration documented / examples? | Y |
| *Synthesis* | Is the IP synthesizable? | Y |
| | FPGA synthesis example available? | Y (as component in CVA6 based design, to be provided) |
| | ASIC synthesis example available? | N |
| *Simulation* | Closed-source simulation? | Y (QuestaSim) |
| | Open-source simulation? | Y (Verilator) |
| *Evaluation* | PPA results available? | N |

#### 3.1.6.2 General Information

As already presented in Section 3.3.1 of Deliverable D3.1, the Context Aware Monitoring framework is a set of hardware IPs to enrich monitoring with software context information such as the

currently running virtual machine in the hypervisor, the currently running process in the operating system or the currently running thread/function at application level.

The Context Aware Monitoring framework is composed of 4 different hardware IPs defined in both WP2 and WP3:

1. The **CA-CORE extension** extends the CVA6 [29] core design with an extra CSR register storing the software context information, and providing both a software and a hardware interface to setup and access the context information.
2. The **CA-BUS extension** decorates memory requests with information related to the software context, both at core-level up to the L1 cache and at NoC level for DDR accesses on the AXI bus.
3. The **CA-PMC extension** implements context-based performance monitor collection, filtering collected hardware events based on the software context, counting event only for a specific process of the operating system, or specific cryptographic function in the application.
4. The **CA-PMC-IF extension** (Section 3.2.1) provides a memory-mapped interface to the software context and the monitored core performance monitor counters (PMCs) enabling remote out-of-core monitoring.

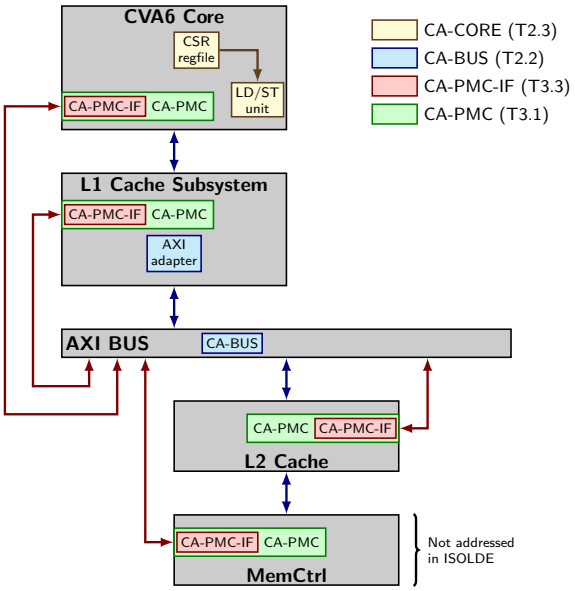


Figure 3.12: Context Aware Monitoring architecture concept

Figure 3.12 shows an example of an architecture using the 4 IPs provided by the Context Aware Monitoring framework.

Prior to ISOLDE, we developed METrICS [13], a software-level Measurement Environment for multi-core Time Critical Systems dedicated at non-intrusively collecting performance monitoring information. Such information is currently used to characterize the behavior of the software on the hardware in terms of hardware resource usage, and anticipate the timing interference issue inherent to multi-core systems.

We then developed the Cyber BlackBox [14] as a way to couple this monitoring information with machine learning to learn the expected behavior of a critical software, and detect deviations from this expected behavior as anomalies indicator of either safety-related failures or security-related cyber-attacks.

A drawback of a monitoring framework at software level, is to provide the user with a new attack surface (the monitoring framework itself) introducing new potential vulnerabilities and the ability to spy on the critical application.

The ISOLDE project and the open hardware ecosystem is an opportunity to implement such a monitoring infrastructure at hardware level, with dedicated memories not accessible from the software layer. The safety/security extensions presented above will allow us to collect context-aware monitoring information at hardware level from a core dedicated to the monitoring the applicative cores.

#### 3.1.6.3 Purpose and Scope

The purpose of the CA-PMC IP is to implement the counters associated to the monitored IP, providing a memory-mapped bus interface to control and read the counters, enabling the counters access to a monitoring core.

#### 3.1.6.4 Refined architecture description

The architecture of the CA-PMC IP depends on the IP it is monitoring, e.g. in Figure 3.12 the core will have a different CA-PMC module, aka the CA-PMC Core module, than the one in the memory controller, aka the CA-PMC Memory Controller module. To simplify the design of the different CA-PMC modules the CA-PMC-IF module has been introduced. The CA-PMC-IF module provides an uniform interface to all the CA-PMC modules. The CA-PMC-IF provides storage for the counters and their management, and a memory-mapped bus interface, so all the CA-PMC modules can provide the same interface to the monitoring core(s).

With the CA-PMC-IF module instantiated the only task the CA-PMC module needs to do is to convert the monitoring element signals into events that the CA-PMC-IF can use to update the counters it manages. For example a CA-PMC Core module might generate a number of instructions committed each cycle from the signals on the Core Scoreboard. The CA-PMC input signals and the signal to event translation logic are highly dependent on the module the CA-PMC is associated with and their description is out of the scope for this document. Figure 3.13 presents a high-level view of the CA-PMC architecture.

The CA-PMC is to be included/instantiated in the module it is providing counters for, like depicted in Figure 3.13, or as an external module connected to the module it is providing counters for, like depicted in Figure 3.14. The second is the approach followed in the ISOLDE project, but both are possible. To simplify our diagrams we typically represent the CA-PMC module as a submodule of the module its is providing counters for, as show in Figure 3.12.

Finally, as shown in Figures 3.13 and 3.14 the CA-PMC module routes the interrupt signal and the AXI Slave provided by the CA-PMC-IF instance, that respectively are forwarded typically to an interrupt controller and to an AXI bus.

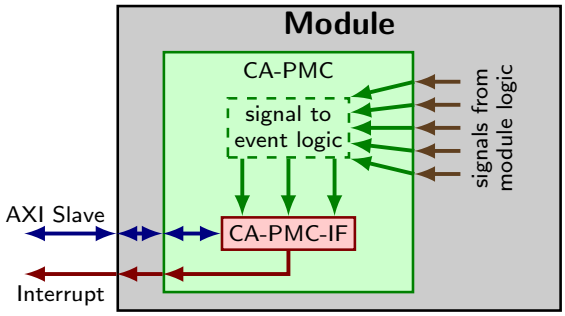


Figure 3.13: CA-PMC module architecture and example of instantiation in module

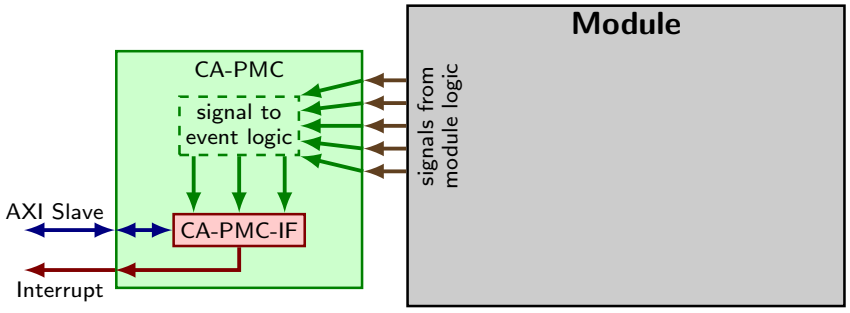


Figure 3.14: Alternative CA-PMC module instantiation

#### 3.1.6.5 Interfaces

As described in Section 3.1.6.4, the CA-PMC module provides three interfaces:

1. Signals from the monitored module (input): Signals from the monitored module allowing the generation of events that can be counted by the CA-PMC module (through the support provided by the CA-PMC-IF module). The signals also contain the context information generated by the CA-CORE module and forwarded by intermediate modules (e.g. bus, caches, etc.).
2. AXI Slave interface (input/output): An addressable slave interface, i.e. will only respond to requests, to control the performance monitoring counters operation (e.g. set the event the counter is tracking, enable a counter, enable a counter to generate overflow, etc.) and read-/write the counters. The actual implementation of the AXI Slave interface and its memory mapping is done by the CA-PMC-IF module instantiated by the CA-PMC and the later simply forwards the interface. For a complete description of the interface and its memory map refer to the CA-PMC-IF description in Section 3.2.1.
3. Interrupt (output): An interrupt signaling that one of the counters has overflowed. This signal is actually generated by the CA-PMC-IF module instantiated by the CA-PMC and simply forwarded by the later. For a more complete description of the conditions on which an interrupt can be generated refer to the CA-PMC-IF description in Section 3.2.1.

#### 3.1.6.6 ISA specialization

The CA-PMC module has no impact on the ISA.

#### 3.1.6.7 Evaluation prototype

A CA-PMC prototype of the CVA6 Core has been implemented. We refer to this IP as the *CVA6 CA-PMC*. It provides a parameterizable number of counters (from 1 to 256) and all the events available in the CVA6 version 5.01 can be mapped to any of the counters. The CVA6 CA-PMC module follows the design of the CVA6 core that integrates the first-level cache into the core.

The CVA6 CA-PMC module has been instantiated in two different designs. The first design instantiates the CVA6 CA-PMC alongside a single-core design of the CV32A6, as depicted in Figure 3.15. This design allows to validate the functionalities of the counters provided by the CVA6 CA-PMC module. Currently all the fonctionalites have been validated without the context awareness capability, as the context information generation is still under test. This design is particularly used to:

- initial platform to debug the design of the CVA6 CA-PMC and its integration in a SoC;
- validate the address mapping and access to the CVA6 CA-PMC module from the CVA6 core through the system bus;
- validate the counters counting operation by comparing them to regular performance monitoring counters provided by the regular CVA6 core;
- and validate the counter overflow interrupts.

This single-core design has been mostly validated in simulations under Verilator. It has also been succesfully sinthesized for the Genesys 2 board (Xilinx Kintex-7 FPGA).

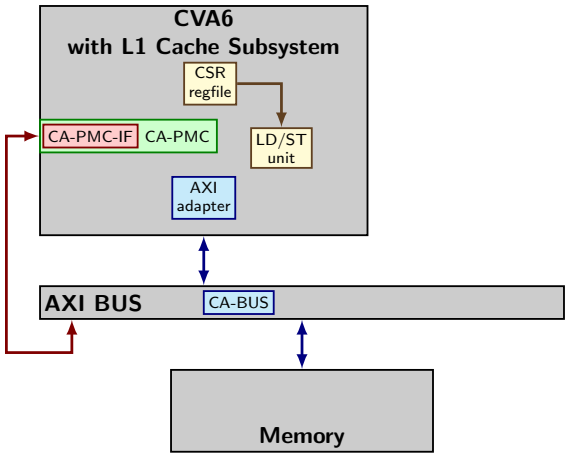


Figure 3.15: Single-core prototype of the Context Aware Monitoring framework

The second design is a multi-core design with two, four or eight CV32A6 cores, each with its own level 1 cache, connected to the system bus. Each core has its own CA-PMC Core module, i.e. the design contains as many CA-PMC Core modules as cores, and the CA-PMC core modules are

connected to the system bus allowing one core to access the performance counters of the other core through memory requests (i.e. load and store instructions). This design as the first one does not have a level 2 cache. Figure 3.16 shows an instantiation of the design with two cores, i.e. dual-core. This design allows to validate the access of the counters from anywhere in the system (as long as the core and the accessed CA-PMC share the same address space). Various tests have succesfully been performed, but as in the first design the context awareness has not been tested and in this case neither the counter overflow interrupts. This design has only been tested using simulation, the synthetizable model being developed at the time of this writing.

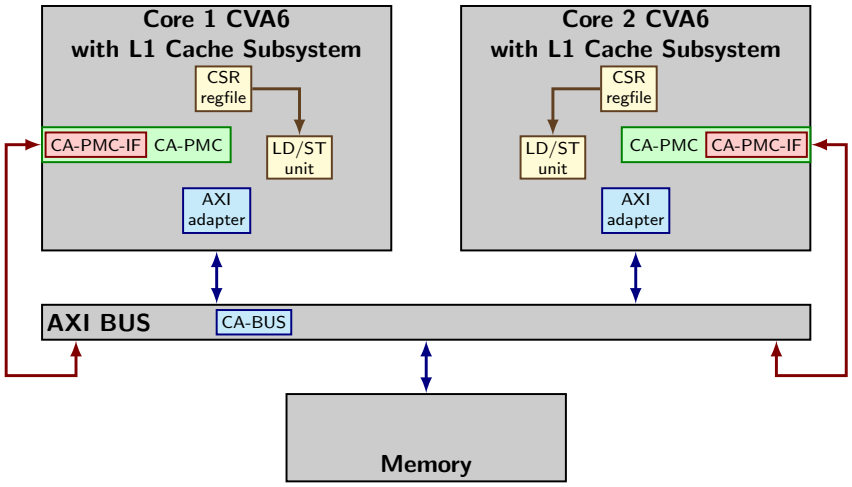


Figure 3.16: Dual-core prototype of the Context Aware Monitoring framework

Listing 3.1 provides the output of one of the bare metal tests on the dual-core design shown in Figure 3.16. For this test each CA-PMC Core module has 8 counters available. In the test the two cores synchronize (not showed in the listing) before starting the CA-PMC modules test. After synchronization the two cores start to execute the same program simultaneously:

- First each core checks the capabilities (see Section 3.2.1) of the CA-PMC Core module attached to it. We can observe that each core detects that their CA-PMC Core module has 8 counters available, line 6 in Listing 3.1 for core 1 and line 5 for core 0.
- Then each core initializes each counter to predetermined events (e.g. counter 0 is set to event 0 that corresponds to the cycle event, counter 1 to event 1 that corresponds to the instruction executed event, etc.), lines 14 and 13 in the listing for core 0 and 1 respectively.
- Then each core enables all the counters in the CA-PMC Core modules, lines 16 and 15 in the listing for core 0 and 1 respectively. At this moment the counters will start to count the events each one is associated with.
- Then each core start to perform some matrix computations. The purpose of these computations is to generate events and make the counters count.
- After finishing the matrix computations each core disables the counters of their associated CA-PMC Core modules, lines 20 and 31 in the listing for core 0 and 1 respectively. At this point the CA-PMC Core counters will stop counting when their associated events when those occur, i.e. they retain the value they had before the being disabled.

- Then each core dumps the values and associated events of each counter in their associated CA-PMC Core module, lines 21, 22, and 24-29 for core 0 in the listing and lines 32-39 for core 1. After dumping the counters each core displays the `Goodbye` message, lines 30 and 40 in the listing for core 0 and core 1 respectively.
- After displaying the `Goodbye` message each core checks if the other one has finished, i.e. already displayed the `Goodbye` message. If the other core has finished the core simply enters and infinite loop without doing anything. In Listing 3.1 we can observe that is core 0 that finishes first as there is no more output from that core after displaying the `Goodbye` message. The core that finishes the latest, core 1 in the listing, then dumps all counters values and associated events for each of the CA-PMC Core modules in the design, lines 41 to 56 in the listing. We can observe that the information displayed matches the information previously displayed by each of the cores.

The same bare metal test and others were successfully run on simulations of the two, four and eight cores instantiations of the multi-core design.

Listing 3.1: Output of baremetal test in dual-core design

```
[C1] Checking mapped PMC support
[C0] Checking mapped PMC support
[C1] capabilities.version = 1
[C0] capabilities.version = 1
[C1] capabilities.ncounters = 8
[C0] capabilities.ncounters = 8
[C1] capabilities.nevents = 24
[C0] capabilities.nevents = 24
[C1] capabilities.custom = 0
[C0] capabilities.custom = 0
[C1] capabilities.ctxlen = 8
[C0] capabilities.ctxlen = 8
[C1] Setting counters
[C0] Setting counters
[C1] Enabling counters
[C0] Enabling counters
[C1] Filling matrix
[C0] Filling matrix
[C0] Reducing matrix
[C0] Disabling counters
[C0] Counter 0 event 0 = 0x000000000000cf98
[C0] Counter 1 event 1 = 0x0000000000008fbe
[C1] Reducing matrix
[C0] Counter 2 event 6 = 0x000000000000120f
[C0] Counter 3 event 7 = 0x0000000000001183
[C0] Counter 4 event 2 = 0x000000000000003f
[C0] Counter 5 event 3 = 0x0000000000000409
[C0] Counter 6 event 10 = 0x00000000000024a7
[C0] Counter 7 event 15 = 0x0000000000000020
[C0] Goodbye
[C1] Disabling counters
[C1] Counter 0 event 0 = 0x000000000001361f
[C1] Counter 1 event 1 = 0x0000000000009196
[C1] Counter 2 event 6 = 0x0000000000001222
[C1] Counter 3 event 7 = 0x00000000000011bb
[C1] Counter 4 event 2 = 0x000000000000003d
[C1] Counter 5 event 3 = 0x000000000000040a
[C1] Counter 6 event 10 = 0x0000000000002516
[C1] Counter 7 event 15 = 0x0000000000000032
[C1] Goodbye
[C1] Core 0 counter 0 event 0 = 0x000000000000cf98
[C1] Core 0 counter 1 event 1 = 0x0000000000008fbe
[C1] Core 0 counter 2 event 6 = 0x000000000000120f
[C1] Core 0 counter 3 event 7 = 0x0000000000001183
[C1] Core 0 counter 4 event 2 = 0x000000000000003f
[C1] Core 0 counter 5 event 3 = 0x0000000000000409
[C1] Core 0 counter 6 event 10 = 0x00000000000024a7
[C1] Core 0 counter 7 event 15 = 0x0000000000000020
[C1] Core 1 counter 0 event 0 = 0x000000000001361f
[C1] Core 1 counter 1 event 1 = 0x0000000000009196
[C1] Core 1 counter 2 event 6 = 0x0000000000001222
[C1] Core 1 counter 3 event 7 = 0x00000000000011bb
[C1] Core 1 counter 4 event 2 = 0x000000000000003d
[C1] Core 1 counter 5 event 3 = 0x000000000000040a
[C1] Core 1 counter 6 event 10 = 0x0000000000002516
[C1] Core 1 counter 7 event 15 = 0x0000000000000032
```

### 3.1.7 Memory Subsystem Support for Bytecode VMs – HM

#### 3.1.7.1 IP Card

| | | |
|---|---|---|
| **Basic Info** | | |
| *IP name* | Bytecode VM Memory Subsystem | |
| *License* | Open-source (Apache2 License) | |
| *Repository* | https://github.com/hm-aemy/wasm-hwstack (pending publication) | |
| **Architecture** | | |
| *Clock* | Number of clock domains | 1 |
| | Synchronous with system | Y |
| | Clock generated internally | N |
| *Ctrl Interface* | ISA extension? | Y (CSR) |
| | Memory mapped? | N |
| | Protocol | N/A |
| | Address Map | N/A |
| *Initiator Interface* | Protocol | Simplified bus, AXI Wrapper |
| | Cached? | N |
| | IOMMU? | N |
| *Interrupts* | Interrupts | N |
| **Microarchitecture** | | |
| *Parametrization* | Parametric no. units? | N |
| | Parameteric config? | Y |
| *Programmability* | Contains programmable cores? | N |
| | ISA | - |
| **Software** | | |
| *Compiler* | Requires specialized compiler? | N |
| | Compiler repository | - |
| *Hardware Abstraction Layer* | N/A | |
| *High-level API* | Is there a high-level API/SDK? | N |
| | SDK repository | N/A |
| | Is there a domain-specific compiler? | N |
| **Integration** | | |
| *IP Distribution* | Manifest type (if any) | N |
| | Standalone simulation? | Y (cocotb verification) |
| | (if standalone sim) SW requirements? | N |
| | Integration documented / examples? | Y |
| *Synthesis* | Is the IP synthesizable? | Y |
| | FPGA synthesis scripts/example available? | N |
| | ASIC synthesis scripts/example available? | N |
| *Simulation* | Closed-source simulation? | N |
| | Open-source simulation? | Y (Verilator, Icarus) |
| *Evaluation* | PPA results available? | N |

#### 3.1.7.2 General Information

The memory subsystem support for bytecode VMs is a standalone HW module that serves as a hardware-assisted stack. Software is adopted with CSR accesses to substitute memory ac-

cesses, essentially offloading the entire stack machine handling into hardware, with the goal of single-cycle accesses to the stack.

#### 3.1.7.3 Purpose and Scope

While the stack of the virtual machine is stored in memory, this module virtualizes the access with push and pop access via CSRs. The module caches the top of the stack for faster access and asynchronously moves data between the internal cache and main memory. Besides the caching effect, this offloads all address calculations from software.

The module is integrated with CVA6 and other cores from the same family.

#### 3.1.7.4 Place in the System

The module is tightly integrated into the core via the CSR interface, extending the CSR map with custom extension CSRs. As a second interface, the module has a simple bursting memory interface that can trivially be mapped to AXI4Lite. A wrapper is available.

The configuration parameters are:

`STACK_SIZE` Size of the cached stack
`STACK_BURST_SIZE` Size of bursts, which are asynchronously fetched and stored.

#### 3.1.7.5 Architecture

The architecture is internally organized into four components:

**CSR logic** serves requests for the CSRs of this module, which are split into configuration and runtime registers. On both register types stall cycles can appear, which limits the integration to cores that allow for multi-cycle CSR accesses.

**Stack** manages a local cache of the current top elements of the stack. It exposes the top of the thread to the CSR logic and signals its fill state.

**Memory logic** transfers data in bursts between the local cache and main memory. In cases the local stack cache runs empty, it refills with the next elements from memory, and in cases it runs full stores the data to the memory.

**Control logic** Synchronizes between the modules, handles configuration accesses and signals wait states.

#### 3.1.7.6 Interfaces

**Bus/AXI4Lite interface**

The bus interface is a simple standard bus interface that only handles one outstanding memory transaction at a time. It is trivially mapped to AXI4Lite and other bus protocols.

**CSR interface**

The CSR interface is defined the following table:

| CSR | Index | Field | Bits | Function |
|---|---|---|---|---|
| stackctrl | TBD | FILL | 31..8 | Current fill state (read only) |
| | | EN | 0 | Enabled. On transition from 1 to 0, the module flushes the stack content to memory. |
| stackbase | TBD | ADDR | XLEN-1..0 | Base address of stack in memory |
| stackptr | TBD | ADDR | XLEN-1..0 | Current top address of stack in memory (for context switches) |
| stackend | TBD | ADDR | XLEN-1..0 | End address of stack in memory |
| stackpush | TBD | DATA | XLEN-1..0 | Push data item to stack |
| stackpop | TBD | DATA | XLEN-1..0 | Pop data item from stack |

**Software Interface**

The bytecode virtual machine interpreter (or other stack-based software) accesses the module along the lifecycle as listed below

```
// Configure the currently running application
csr_write(stackbase, <base-address>);
csr_write(stackptr, <base-address>);
csr_write(stacktop, <end-address>);
csr_write(stackctrl, 1);

// During runtime: Push an element, example constants
csr_write(stackpush, 2);
csr_write(stackpush, 3);

// During runtime: Pop elements, push result, example add
p0 = csr_read(stackpop);
p1 = csr_read(stackpop);
csr_write(stackpush, p0+p1);

// Context switch between two applications
csr_write(stackctrl, 0);
csr_swap(stackbase, ctx0.stackbase, ctx1.stackbase);
csr_swap(stackptr, ctx0.stackptr, ctx1.stackptr);
csr_swap(stackend, ctx0.stackend, ctx1.stackend);
csr_write(stackctrl, 1);
```

## 3.1.8 Safety-Related Traffic Injector (SafeTI) – BSC

### 3.1.8.1 IP Card

| | | |
|---|---|---|
| **Basic Info** | | |
| *IP name* | SafeTI | |
| *License* | Open-source (MIT License) | |
| *Repository* | https://github.com/bsc-loca/SafeTI/tree/21678221e4bb9c9e51be7d135f0644213726fe7e | |
| **Architecture** | | |
| *Clock* | Number of clock domains | 1 |
| | Synchronous with system | Y |
| | Clock generated internally | N |
| *Ctrl Interface* | ISA extension? | N |
| | Memory mapped? | Y |
| | Protocol | APB 32b |
| | Address Map | Base (0xfc085000) + 0x100 |
| *Initiator Interface* | Protocol | APB 32b |
| | Cached? | N |
| | IOMMU? | N |
| *Interrupts* | Interrupts | N |
| **Microarchitecture** | | |
| *Parametrization* | Parametric no. units? | N |
| | Parameteric config? | Y |
| *Programmability* | Contains programmable cores? | Y, but using its own traffic pattern descriptors |
| | ISA | - |
| **Software** | | |
| *Compiler* | Requires specialized compiler? | N |
| | Compiler repository | - |
| *Hardware Abstraction Layer* | N/A | |
| *High-level API* | Is there a high-level API/SDK? | Y |
| | SDK repository | https://github.com/bsc-loca/SafeTI/tree/21678221e4bb9c9e51be7d135f0644213726fe7e/sw |
| | Is there a domain-specific compiler? | N |
| **Integration** | | |
| *IP Distribution* | Manifest type (if any) | Y (Readme.md) |
| | Standalone simulation? | Y |
| | (if standalone sim) SW requirements? | QuestaSim and Xilinx Vivado 2020.2 |
| | Integration documented / examples? | https://gitlab.com/selene-riscv-platform/selene-hardware/-/tree/ISOLDE/grlib/software/noelv/BSC_tests/injector_tests_template |
| *Synthesis* | Is the IP synthesizable? | Y |
| | FPGA synthesis scripts/example available? | Y |
| | ASIC synthesis scripts/example available? | N |
| *Simulation* | Closed-source simulation? | Y (QuestaSim) |
| | Open-source simulation? | N |
| *Evaluation* | PPA results available? | https://doi.org/10.1145/3703910 |

### 3.1.8.2 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.

#### 3.1.8.3 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.

#### 3.1.8.4 Place in the System

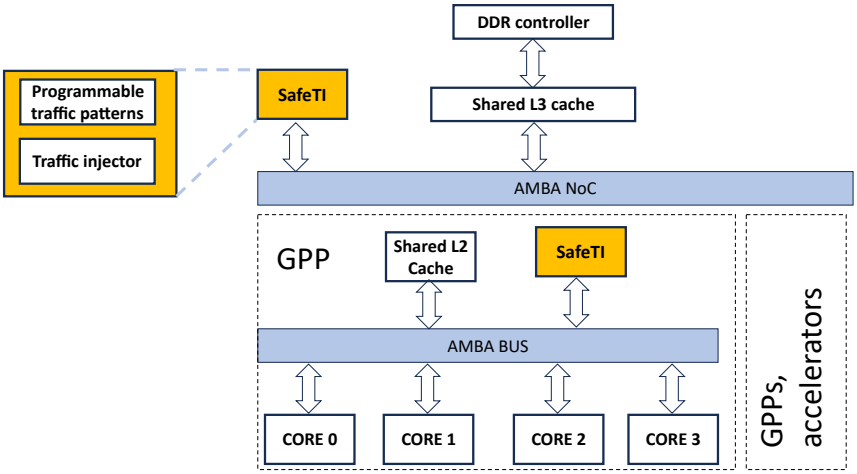


Figure 3.17: SafeTI system integration

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 Figure3.17. 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) and will be compliant with AMBA AXI in the future.

#### 3.1.8.5 Architecture

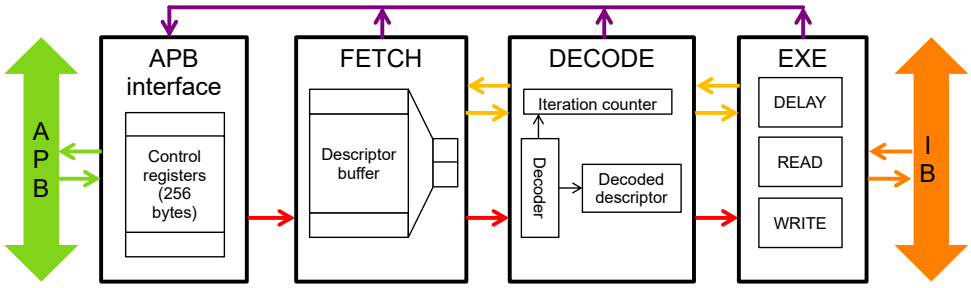


Figure 3.18: SafeTI architecture block diagram

The SafeTI has a set of control registers programmed through an APB or AXI 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 3.18), 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.

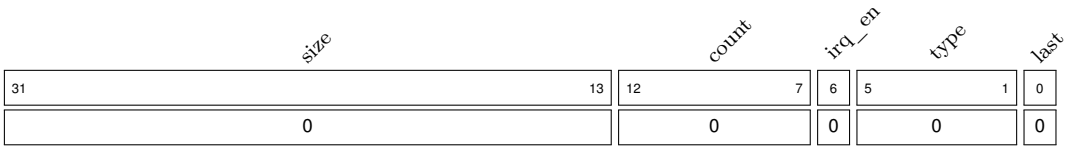


Figure 3.19: SafeTI as microprogrammed and memory-mapped module using its own descriptor format

The SafeTI is a microprogrammed and memory-mapped module using its own descriptor format, which we illustrate in Figure 3.19. 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 a lighter implementation. Changes to the first descriptor word fields are considered in future descriptor type expansions. Field 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

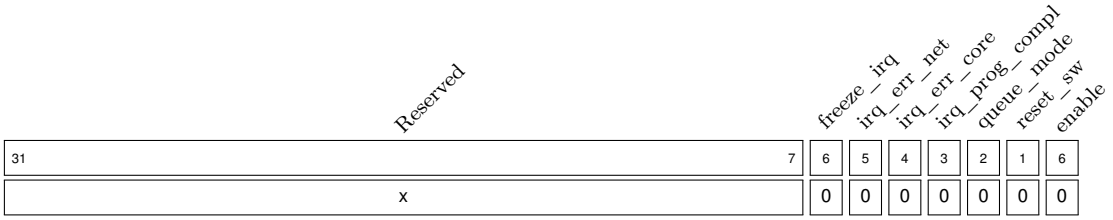


Figure 3.20: SafeTI Hardware Interface

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 3.20, 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 under 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 Next Generation, also known as XNG, in the particular case of the SafeTI integration in ISOLDE) is in charge of managing privileges to allow only specific partitions to update 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. XNG 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.

For allowing a partition of the XNG hypervisor to manage the SafeTI device from a high-level perspective, a driver at hypervisor level is being developed and integrated by FENTISS, which exposes to the partitions a new set of hypercalls. Hypercalls are services or system calls provided by the hypervisor to the partitions, and requesting them causes a privilege escalation from the partition running at user mode to the core running at system mode. These new hypercalls implemented, specified in Table 3.6 allow managing the SafeTI device by a partition which has SafeTI access permissions.

In order to define whether or not a partition has access to the SafeTI device, a new attribute in the Xtratum Configuration Files (XCF) has been added. If a partition's XCF includes the "safeti"' attribute, that partition is granted with permissions to access SafeTI registers and use the provided hypercalls without restriction. Otherwise, if the attribute is not present, any attempt to the partition to invoke a SafeTI hypercall will result in an error code indicating an invalid configuration.

Finally, in order to receive from a partition the interrupts generated by the SafeTI device, the XCF of the partition needs to be configured to delegate them. It is also important to note that the partition should allow the interrupts reception, unmask the desired interrupts and install the desired handlers to be triggered when an interrupt is received, which can be done through other hypercalls implemented within XNG.

| Hypercall | Description |
|---|---|
| `retCode XSafeTiProgramDescriptor( desc_type, size, attack_addr, count, last, int_en)` | Configures and writes an individual descriptor into the FIFO queue by writing to the descriptor memory feed register. |
| `retCode XSafeTiConfig( en, queue_en, int_prog_compl, int_error, int_net_error, freeze_int)` | Programs the configuration register of the SafeTI injector. |
| `retCode XSafeTiReset()` | Resets the injector and clears current configuration. |
| `retCode XSafeTiGetCounterVal( counter, *value)` | Reads a SafeTI counter (interrupts or accesses). |
| `retCode XSafeTiCountersReset()` | Resets the SafeTI counters. |
| `retCode XSafeTiIsRunning( *value)` | Returns whether or not the SafeTI module is running. |

Table 3.6: Hypercall table implemented within XNG for SafeTI device support

#### 3.1.8.7 Evaluation Prototype

To evaluate the SafeTI, it has been integrated in the SELENE SoC and implemented on a Xilinx FPGA operating at 100MHz as seen in Figure 3.17. One instance is of the SafeTI is integrated in the AMBA AHB Bus, SafeTI_AHB, while the other instance is connected to the AMB AXI Bus, SafeTI_AXI. To evaluate their correct operation, different injection patterns will be set using the AXI descriptors. The SafeSUs attached to the same bus will report whether the traffic injected is correct as expected by the descriptors employed or not.

### 3.1.9 Safety and Security Control Unit – IFX

#### 3.1.9.1 IP Card

| | | |
|---|---|---|
| **Basic Info** | | |
| *IP name* | Safety and Security Control Unit | |
| *License* | Open Source; Details to be defined | |
| *Repository* | At the moment local. Will be released later publicly | |
| **Architecture** | | |
| *Clock* | Number of clock domains | 1 |
| | Synchronous with system | Y |
| | Clock generated internally | N |
| *Ctrl Interface* | ISA extension? | Custom CSRs required |
| | Memory mapped? | N |
| | Protocol | Initially AHB; Current implementation CSR |
| | Address Map | Custom to Platform |
| *Initiator Interface* | Protocol | CSR instructions |
| | Cached? | N |
| | IOMMU? | N |
| *Interrupts* | Interrupts | Y |
| **Microarchitecture** | | |
| *Parametrization* | Parametric no. cores? | N |
| | Parameteric config? | RTL from a parametrized generator |
| *Programmability* | Contains programmable cores? | N |
| | ISA | RV32I + Custom CSRs |
| **Software** | | |
| *Compiler* | Requires specialized compiler? | N |
| | Compiler repository | N/A |
| *Hardware Abstraction Layer* | N/A | |
| *High-level API* | Is there a high-level API/SDK? | N |
| | SDK repository | N/A |
| | Is there a domain-specific compiler? | N |
| **Integration** | | |
| *IP Distribution* | Manifest type (if any) | Planned; Format to be defined |
| | Standalone simulation? | Y |
| | (if standalone sim) SW requirements? | N |
| | Integration documented / examples? | Will be released together with the IP |
| *Synthesis* | Is the IP synthesizable? | Y |
| | FPGA synthesis example available? | Planned |
| | ASIC synthesis example available? | N |
| *Simulation* | Closed-source simulation? | Y (Xcelium) |
| | Open-source simulation? | Planned |
| *Evaluation* | PPA results available? | N |

#### 3.1.9.2 General Information

The Safety and Security Control Unit (SSCU) serves as a centralized control unit. Its role is to collect all error signals coming from various sources on the chip, analyze and process them, and in turn alarm the system when a critical state is reached.

This implementation of the SSCU is architected to be tightly integrated with a RISC-V CPU core, enabling a highly efficient configuration for systems with a single processing core. In such systems, where the RISC-V core serves as the sole initiator for accessing the SSCU, it becomes unnecessary to connect the SSCU to the shared bus interconnect. Unlike multi-core designs or systems with multiple initiators, there is no requirement for other components to access the SSCU's registers. Consequently, leveraging RISC-V CSRs as the primary interface for the SSCU becomes a more optimal solution.

By decoupling the SSCU from the bus interconnect, the overall complexity of the bus infrastructure is reduced. Notably, this eliminates the need for address decoding logic specific to the SSCU, thereby streamlining the datapath and reducing the critical timing path associated with bus operations. Furthermore, accessing the SSCU via CSR instructions bypasses the latency and contention typically associated with shared bus transactions, offering significantly lower access times. The RISC-V CSR mechanism inherently allows for efficient read and write operations, as they are tightly coupled to the instruction pipeline of the CPU core.

Additionally, the tight physical coupling between the SSCU and the RISC-V core introduces further architectural advantages. By situating the SSCU in close proximity to the processor core, wire lengths are minimized, which in turn reduces signal propagation delays and mitigates power dissipation associated with long interconnects. This proximity optimizes both timing and energy efficiency, contributing to improved system performance.

Figure 3.21 provides an example of one possible integration of the SSCU in a RISC-V core. The error signals that are external to the core perspective would be configured as an asynchronous interrupts, meanwhile the error signals internal to the core would be configured as a custom synchronous exception to enable the quickest way to react.

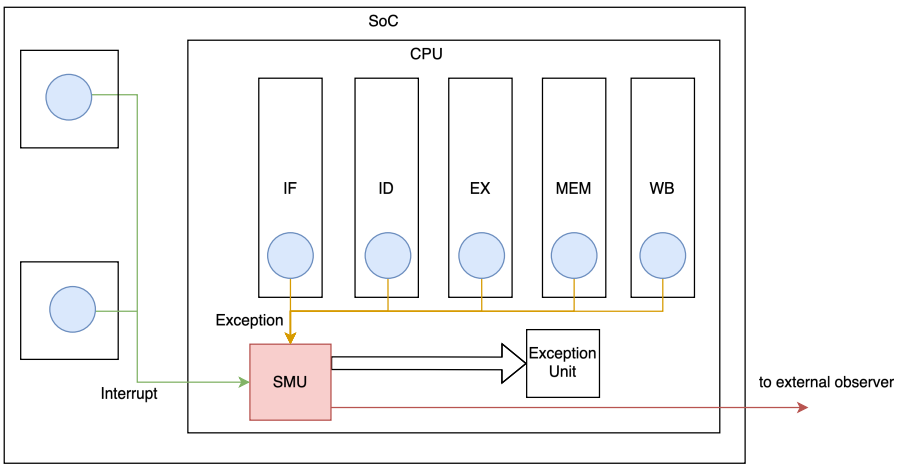


Figure 3.21: Example of SSCU integration in a 5-stage RISCV core.

#### 3.1.9.3 Purpose and Scope

The SSCU is designed as a universal module capable of integration into a wide range of safety-critical platforms and applications. Its primary role is to manage the handling of Single Event faults,

ensuring robust system operation in environments prone to transient errors caused by radiation or high-energy particles. However, its application is not limited only to this, making it a versatile solution suitable for broader fault-handling requirements in safety-critical systems.

The SSCU is focused exclusively on fault handling. It does not directly participate in the detection or correction of such faults. These tasks—detection and correction—are delegated to the safety modules implemented within the chip itself. This design decision ensures that the SSCU remains flexible and agnostic to specific fault models, enabling seamless integration into diverse platforms and applications. The SSCU operates at a higher conceptual level, dealing only with error management, regardless of the underlying mechanisms used for fault detection and correction.

We refer to these hardening mechanisms simply as safety nodes. Each node must have the following attributes:

- *identifier* : unique identifier for each node within the system.
- *function* : specific original functionality performed by the node such as error correction or error detection.
- *error detection/correction mechanisms* : the implementation-specific method used by the node to detect or correct errors.
- *trap specifier* : additional specifier indicates which trap handler of the CPU is responsible to handle the particular error generated from this node.

Different safety nodes have varying levels of severity and importance regarding system functional safety. To address this, nodes are categorized into groups based on the severity and handling requirements of the errors they detect. These groups require distinct logical management. By centralizing all error signals and processing them in group units, the SSCU reduces the number of individual error notifications by generating a single alarm for each group. Each alarm can be configured to trigger internal actions or notify external systems of faults handling via a fault signaling protocol.

Currently, the supported errors can be categorized based on their status:

- *uncorrected errors* : errors detected but not corrected by the node, which will propagate in the next cycle and possibly contaminate the hardware state.
- *corrected errors* : errors detected and corrected by the node, preventing them from affecting subsequent cycles

Figure 3.22 shows an example design of a node with error detection and correction capabilities. This node has the ability to correct errors using spatial redundancy. It features an Encoder, Register Bank, and Decoder.

The Encoder processes input data *Data_in* and an enable signal (Enable), then forwards it to the Register Bank, which includes three protected registers (Reg0, Reg1, Reg2) implementing TMR. The Decoder uses a voter mechanism to correct errors and outputs the corrected data *Data_corrected out*. Additionally, a non-equivalence comparator Decoder generates an *alarm_out* signal if discrepancies are detected. This *alarm_out* signal is wired toward the SSCU and whenever an alarm is detected, specific processing to ensure robust error management of the error is performed.

The described logic is not the only type of logic used for error detection or correction, it is just shown for illustration. Any other mechanism used also works as long as it is able to alarm the

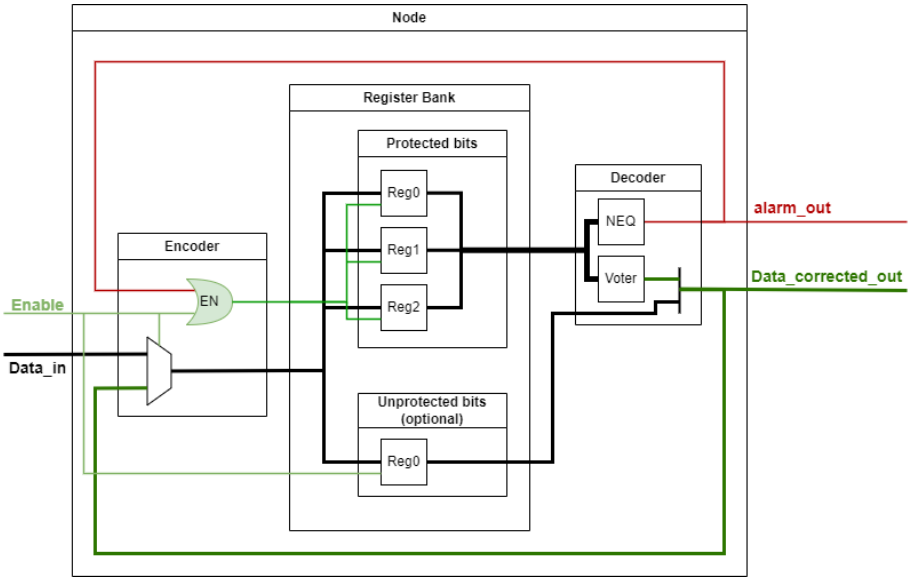


Figure 3.22: Node with Error Correction (TMR).

SSCU accordingly.

#### 3.1.9.4 Refined architecture description

**Error Groups**

As mentioned above, incoming errors towards the SSCU are categorized into groups. Each group has one dedicated alarm. However, the logic that triggers this alarm varies.

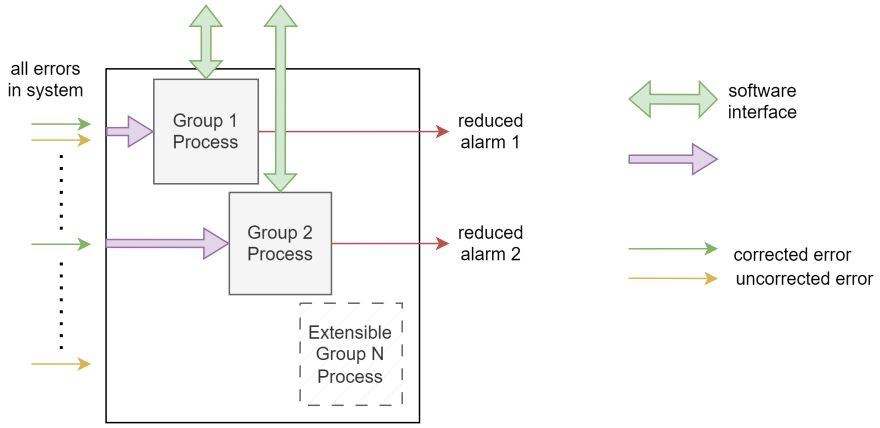


Figure 3.23: SSCU Consisting of Multiple Error Groups.

Figure 3.23 illustrates the concept of error groups within the SSCU. Nodes are organized into

different groups based on the severity and/or type of errors they generate. Each group is processed separately to ensure locally prioritized error handling. All errors are then routed to their respective group processing units. Group 1 Processing handles a subset of errors with specific characteristics, related to certain traps. This group's output generates alarm 1, which signals an error condition based on the aggregated errors from nodes within this group. Group 2 Processing manages another subset of errors, possibly with different severity or different handling requirements. The output of this group generates alarm 2, similarly indicating an error condition for its respective nodes. The SSCU aggregates these error signals and interfaces with software, allowing for real-time monitoring and control. The software interface can read and write CSRs corresponding to each error group.

**Implementing an Error Group**

The nodes within a group are processed in batches, and a so-called batch unit is designed to support a subset of the nodes in the group. This error batch process unit outputs node ID and also the 2-bit of error type (uncorrected, corrected overflow, or uncorrected overflow). Each batch unit produces one batch alarm that feeds into the next stage where a group alarm is generated based on batch alarms. In addition, batch ID is computed to contain information on which batch has fired alarms.

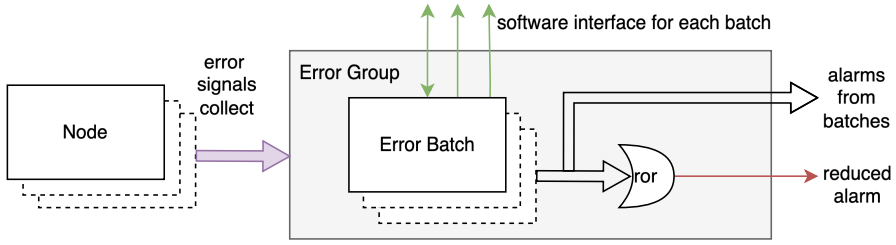


Figure 3.24: Separation of Group in Batches.

Error batch process unit is the module where most of the logic to handle errors is located and also the module that interacts with the software. This unit provides 3 key features: *error prioritization*, *alarm flood prevention* and *diagnostic ability* which are in turn explained below.

First, it is crucial to implement a mechanism that prioritizes different types of errors. This ensures that critical errors are addressed promptly while less severe issues are managed appropriately. Additionally, the unit must prevent alarm flooding by avoiding the continuous sending of alarms for the same cause, which can overwhelm the system and obscure significant new errors. Furthermore, the error batch processing unit needs to have mechanisms for enabling and disabling error reporting based on the system's operational requirements. This flexibility is vital for maintenance and testing purposes. Another requirement is the storage of error information for software diagnostics.

The overview of the error batch processing unit is depicted in Figure 3.25. Alarm flood prevention is achieved through a report control mechanism and an error counter and compare mechanism, which reduces the frequency of servicing the same error. Alarms are prioritized using a priority encoder, and an alarm encoding is produced and stored in the control status register for diagnostic purposes.

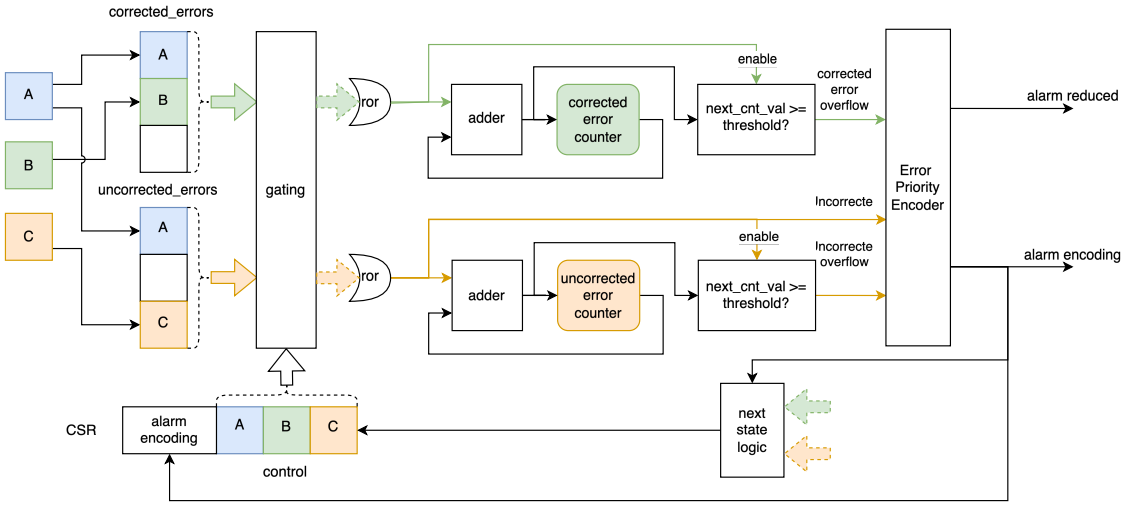


Figure 3.25: High-Level Schematic of Error Batch Process Unit.

There are three main categories of alarms that the SSCU can generate:

- *uncorrected error alarm* : This alarm is triggered when an uncorrected error is detected. Uncorrected errors indicate serious faults that cannot be automatically resolved by the local safety mechanism, requiring system measures such as triggering a CPU exception to step in.
- *uncorrected overflow alarm* : This alarm is activated when the counter for uncorrected errors overflows. It occurs when the number of uncorrected errors exceeds the configured threshold, indicating that the system itself is experiencing catastrophic issues where external action needs to be taken.
- *corrected overflow alarm* : This alarm is triggered when the counter for corrected errors overflows. The purpose of this alarm is to notify that the system is experiencing a high rate of correctable errors, which, while not immediately critical, may indicate underlying issues that could escalate if not addressed.

**Error Diagnosis**

In the presence of an error, whether corrected or uncorrected, the software can diagnose the issue by leveraging the hierarchical structure of the SSCU. The diagnosis process begins with the fact that each error group will trigger a different trap. Each error group corresponds to a specific set of nodes categorized based on the severity and type of errors they generate and by pinpointing the exact error group, the system narrows down the possible sources of the error.

Once the error group is identified, the next step is to determine the specific batch group within that error group that has triggered the alarm. Each error group consists of multiple batch units, and the alarm generated by the error group indicates which batch is responsible. The software reads the CSR of the batch group to further diagnose the issue. The CSR contains detailed information

about the error signals and their origins, including the node error report control bits for each node within the batch. If a node error report control bit is disabled, it indicates that the corresponding node is the source of the alarm. The software can then use the alarm encoding stored in the CSR to identify the type of error—whether it was a corrected error, an uncorrected error, or an overflow condition.

#### 3.1.9.5 Interfaces

As shown in Figure 3.21, the SSCU needs to interact with the Exception Unit of the RISCV core. The implementations of the Exception Unit might vary from core to core, but the idea is to configure different alarms that SSCU generates as custom traps. Once these alarms are generated, the Exception Unit will be responsible to force the core into executing the relevant trap handling routine which in turn will diagnose the SSCU to detect the error and then take further actions.

Relating to the SW interface, the Core needs only to perform the following actions in relation to the SSCU:

- *Error Controls* : which allows the system to enable or disable error reporting based on specific conditions or thresholds. It prevents the system from being overwhelmed by continuous alarms from repetitive or non-critical errors.
- *Error Diagnosis* : which allows the system to identify the root cause of the error and evaluate its criticality so it can take appropriate measures.

To support these actions, the CPU where the SSCU is integrated, must support custom CSRs that have a specific structure. The addresses of the CSRs can vary from platform to platform, based on the availability of the addresses and the number of groups implemented in the SSCU. However, they must be within the ranges reserved for *Custom read/write Machine-Level CSRs*, as defined in the official RISCV specification.

Each group has one CSR that is used to configure this group. The two Least Significant bits of this CSR provide information on the alarm type. All alarm types can be asserted simultaneously. Since uncorrected errors have the potential to propagate and contaminate the circuit state, they need to be given higher priority. Therefore, alarm encoding is introduced to indicate the type of error that needs to be processed. The exact encoding is shown in Table 5.2.

| Uncorrected Error | Uncorrected Overflow | Corrected Overflow | Encoding |
|---|---|---|---|
| 1 | 1 | X | 11 |
| 1 | 0 | X | 10 |
| 0 | 0 | 1 | 01 |
| 0 | 0 | 0 | 00 |

Table 3.7: Alarm Encoding Table.

The rest of the CSR is composed of single bitfields, with each bitfield dedicated to one error node. When cleared, the bitfield disables the error reporting for that specific error node. When SSCU generates an alarm, it will automatically clear the bitfield of the node responsible for the alarm. This way, the SW can simply detect the node that raised the error by simply identifying the cleared bitfield in the CSR. This structuring, limits the number of nodes one group can have to 30.

#### 3.1.9.6 ISA specialization

This unit does not necessitate the introduction of a specialized ISA extension for its operation. Instead, it integrates seamlessly with the existing RISC-V architecture through the addition of a minimal set of CSRs. These custom CSRs are allocated specific addresses within the designated range reserved for User-Defined CSRs, as per the RISC-V specification.

#### 3.1.9.7 Evaluation prototype

The Safety Unit will be integrated into a basic RV32IMC RISC-V core, which will be augmented with specific hardening mechanisms applied to selected registers. These hardening mechanisms are designed to improve system fault tolerance by detecting and mitigating potential errors [12]. Furthermore, the core will include a fault injection mechanism, which introduces faults into the hardened registers at randomized intervals. When faults occur, the hardening mechanisms will detect the errors and trigger a signal to the Safety Unit. Upon receiving this signal, the Safety Unit will initiate a controlled response by forcing the core to execute specific trap handlers. These trap handlers are responsible for executing predefined recovery actions based on the type of fault detected. This entire flow will be simulated and checks will be performed on runtime to evaluate the correct functioning of the Safety Unit.

To further evaluate the system, the design will also be implemented on a basic FPGA platform for hardware-based testing [18]. Fault injection in the FPGA prototype can be manually triggered via external inputs, such as push buttons, allowing for deterministic testing of fault scenarios. The system's response to these faults will be visualized using LED indicators, providing a clear representation of the fault detection and recovery processes in real time.

### 3.1.10 Safety Island - Interface Definition – UzL

UzL has moved the effort to the development and analysis of Floating-Point Unit in T3.2, as reported in D3.1.

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

#### 3.1.11.1 IP Card

| | | |
|---|---|---|
| **Basic Info** | | |
| *IP name* | Root-of-Trust (RoT) | |
| *License* | Open-source (SolderPad Hardware License v0.51) | |
| *Repository* | https://github.com/pulp-platform/opentitan/tree/mc/astral | |
| **Architecture** | | |
| *Clock* | Number of clock domains<br>Synchronous with system<br>Clock generated internally | 1<br>Y<br>N |
| *Ctrl Interface* | ISA extension?<br>Memory mapped?<br>Protocol<br>Address Map | N<br>N<br>SCMI<br>N.A. |
| *Initiator Interface* | Protocol<br>Cached?<br>IOMMU? | AXI4<br>N<br>N |
| *Interrupts* | Interrupts | Y |
| **Microarchitecture** | | |
| *Parametrization* | Parametric no. cores?<br>Parameteric config? | N (1)<br>Y |
| *Programmability* | Contains programmable cores?<br>ISA | Y<br>RISC-V (CV32E) |
| **Software** | | |
| *Compiler* | Requires specialized compiler? | N |
| *Hardware Abstraction Layer* | N/A | |
| *High-level API* | Is there a high-level API/SDK?<br>SDK repository<br>Is there a domain-specific compiler? | Y<br>sw subfolder of main repository<br>N |
| **Integration** | | |
| *IP Distribution* | Manifest type (if any)<br>Standalone simulation?<br>(if standalone sim) SW requirements?<br>Integration documented / examples? | Bender.yml<br>Y<br>Bazel + Python + QuestaSim<br>Example in tree/lg/isolde |
| *Synthesis* | Is the IP synthesizable?<br>FPGA synthesis example available?<br>ASIC synthesis example available? | Y<br>Y<br>N |
| *Simulation* | Closed-source simulation?<br>Open-source simulation? | Y (QuestaSim)<br>N |
| *Evaluation* | PPA results available? | N |

#### 3.1.11.2 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. It includes a secure microcontroller, on-board private memory, cryptographic

hardware accelerators, and communication I/O interfaces. It is based on lowRISC's OpenTitan Earl Grey architecture, which is refactored and extended to turn it into a silicon-secure element easily integrable within larger designs.

#### 3.1.11.3 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, protecting sensitive data and systems against hardware attacks, tampering, and counterfeiting. 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). It enables a multi-stage secure boot process by ensuring each component is authenticated by its predecessor, leveraging hardware features and cryptographic checks to protect against unauthorized modifications or executions. UNIBO's RoT is meant to be a ready-to-integrate silicon IP able to act as a RoT.

#### 3.1.11.4 Refined architecture description

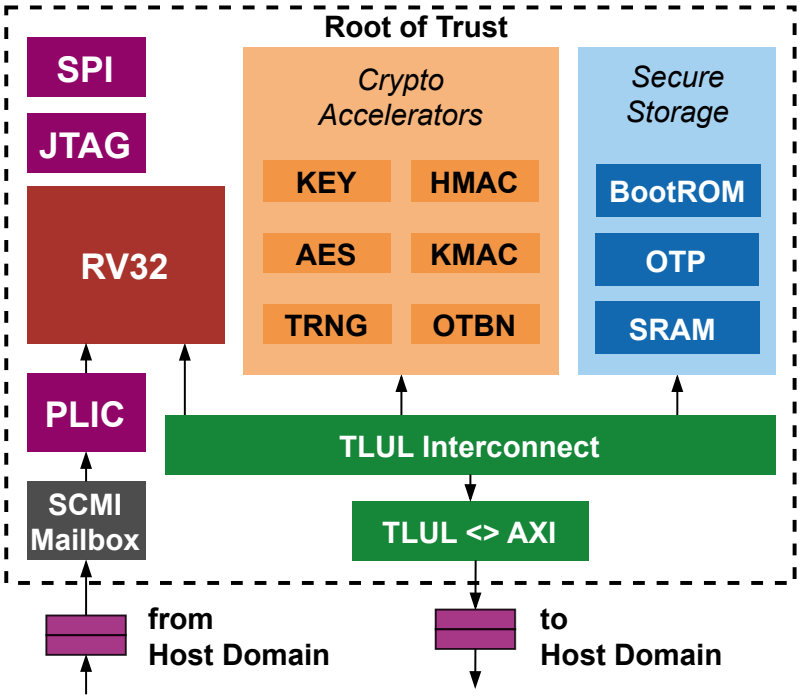


Figure 3.26: Root-of-Trust architecture.

The architecture of the Root-of-Trust (RoT) unit is depicted in Figure 3.26. It is centered around a 32-bit RISC-V core, specifically designed to support embedded and power-efficient applications, with an emphasis on minimal area usage. At the system level, interconnection is facilitated

through the TileLink Uncached Lightweight (TLUL) protocol, ensuring streamlined and secure communication between internal components.

To enhance cryptographic performance, RoT unit incorporates dedicated hardware accelerators (*Crypto Accelerators*) capable of executing essential cryptographic algorithms. These include AES, SHA-256, SHA-3, HMAC, and KMAC, which are fundamental to the silicon Root-of-Trust's security features. In addition, the OpenTitan Big Number (OTBN) accelerator provides hardware support for asymmetric encryption algorithms that underpin key exchange protocols and digital signature schemes.

The memory subsystem (*Secure Storage*) of RoT comprises a scratchpad SRAM and an embedded ROM, each managed by dedicated controllers to optimize performance and reliability. Critical cryptographic and scrambling keys are generated by specialized hardware accelerators and securely stored within a tamper-proof one-time-programmable (OTP) memory region. This secure memory pairs with the key manager, which governs the handling of hardware identities and root keys, while safeguarding sensitive assets against potential software-based threats.

To enable the application domain to request specific services from the Root-of-Trust (RoT)—such as Trusted Execution Environment (TEE) primitives or cryptographic operations—a secure communication channel has been established between the host and the unit. This channel avoids exposing any direct target port that could be driven by the host, thereby maintaining the integrity of the RoT boundary.

The communication mechanism is implemented using a shared mailbox, which adheres to the ARM System Control and Management Interface (SCMI) standard—a protocol widely adopted in heterogeneous architectures. Interactions between the primary processor and the secure module are restricted to message-based exchanges, conducted through an interrupt-driven mailbox system. In accordance with the SCMI specification, this mailbox provides two dedicated interrupt lines, each connected to the Platform Level Interrupt Controllers (PLICs) of the application processor and RoT, respectively. Notably, the external interrupt generated via this mechanism remains the sole pathway through which the host can directly interact with OpenTitan, ensuring strict control over the communication interface and preserving the security guarantees of the RoT.

Finally, it includes a peripheral subsystem that enables external communication through standard interfaces such as SPI and JTAG, thereby facilitating integration into a wide range of embedded environments.

#### 3.1.11.5 Interfaces

To enable integration of the Root-of-Trust (RoT) unit into broader SoC architectures, an initiator port from OpenTitan's TLUL interconnect is exposed and connected via a dedicated TLUL-to-AXI4 bridge, ensuring compatibility with host platforms using the AXI4 protocol.

This setup allows OpenTitan to access the host's memory map, which is essential for delivering TEE primitives and performing real-time integrity checks. The initiator port connects directly to the SoC's main interconnect, granting the RoT full system visibility while preserving security boundaries.

To safeguard sensitive information within secure subsystems and prevent unauthorized host-core modifications, OpenTitan is deliberately integrated without exposing a target port access to the

host. Consequently, interaction from the host to OpenTitan is strictly limited to external interrupts through mailboxes, ensuring rigorous control of the communication interface and upholding robust security guarantees inherent to the RoT.

Finally, the RoT is architecturally designed to interface with the external environment via standard off-chip communication peripherals, such as SPI and JTAG interfaces, thereby ensuring compatibility and facilitating external interaction and debugging capabilities.

#### 3.1.11.6 ISA specialization

The Root-of-Trust unit has no impact on the ISA.

#### 3.1.11.7 Evaluation prototype

We implemented RoT unit in synthesizable SystemVerilog HDL. We targeted GlobalFoundries GF12LP+ 12nm technology to evaluate the RoT in terms of area and performance, using Synopsys Design Compiler for synthesis and Cadence Innovus for place-and-route with a frequency target of 300 MHz. The RoT occupies 0.63mm$^2$ of silicon, of which the Secure Storage subsystem is 58%, the Crypto Accelerators cores are 22%, the RISC-V core is 6%, and the peripherals are 5%.

The design has been evaluated also with an FPGA implementation targeting the AMD Xilinx UltraScale+ VCU118 board with a 20 MHz target frequency, using AMD Xilinx Vivado 2020.1. The RoT occupies approximately 250,000 lookup tables (LUTs), 147,000 flip-flops (FFs), 40 36kb blocks of block RAM (BRAM) and 16 unit of UltraRAM (URAM), corresponding to 21%, 6%, 2%, and 2% of the board's available resources, respectively.

### 3.1.12 High-Performance Cache Analysis – SYSGO

#### 3.1.12.1 IP Card

We include an IP card, although in this case the IP was developed by CEA in TRISTAN, we are just using the IP as a demonstrator; here we have answered the questions to the best of our knowledge.

| | | |
|---|---|---|
| **Basic Info** | | |
| *IP name* | CV-Hpdcache | |
| *License* | Open-source (Solderpad Hardware License (version 2.1)) | |
| *Repository* | https://github.com/openhwgroup/cv-hpdcache | |
| **Architecture** | | |
| *Clock* | Number of clock domains | 1 |
| | Synchronous with system | Y |
| | Clock generated internally | N |
| *Ctrl Interface* | ISA extension? | Y |
| | Memory mapped? | Y |
| | Protocol | N |
| | Address Map | Y |
| *Initiator Interface* | Protocol | Cache-Requesters Interface (CRI) |
| | Cached? | Y |
| | IOMMU? | N |
| *Interrupts* | Interrupts | Y |
| **Microarchitecture** | | |
| *Parametrization* | Parametric no. cores? | Y |
| | Parameteric config? | Y |
| *Programmability* | Contains programmable cores? | N |
| | ISA | RISC-V |
| **Software** | | |
| *Compiler* | Requires specialized compiler? | N |
| *Hardware Abstraction Layer* | N/A | |
| *High-level API* | Is there a high-level API/SDK? | N |
| | SDK repository | N |
| | Is there a domain-specific compiler? | N |
| **Integration** | | |
| *IP Distribution* | Manifest type (if any) | Bender.yml |
| | Standalone simulation? | Y |
| | (if standalone sim) SW requirements? | Verilator |
| | Integration documented / examples? | Integration Examples of the HPDCache |
| *Synthesis* | Is the IP synthesizable? | Y |
| | FPGA synthesis example available? | Y |
| | ASIC synthesis example available? | N |
| *Simulation* | Closed-source simulation? | N |
| | Open-source simulation? | Y |
| *Evaluation* | PPA results available? | N |

#### 3.1.12.2 General Information

We analyze the high-performance cache provided by CEA in the TRISTAN project for the ISOLDE demonstrator. The output will be a short analysis that compares the CEA cache with the default cache. We have successfully generated the bitstream for CVA6 with HPC enabled using Vivado, as shown in Figure 3.28. We are currently working on the software setup. We have not done the analysis yet; hence the other content of this section is the same as in D3.1.

#### 3.1.12.3 Purpose and Scope

Advance CVA6 ecosystem by showing the usability of advanced caches. Our general focus is on safety and security, and correct use of caching is an important part, e.g. for information flow and interference control in multicore systems. Hence, we are analyzing the available caches for CVA6, and possibly to shape future implementation of cache partitioning.

#### 3.1.12.4 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.

#### 3.1.12.5 Block Diagram

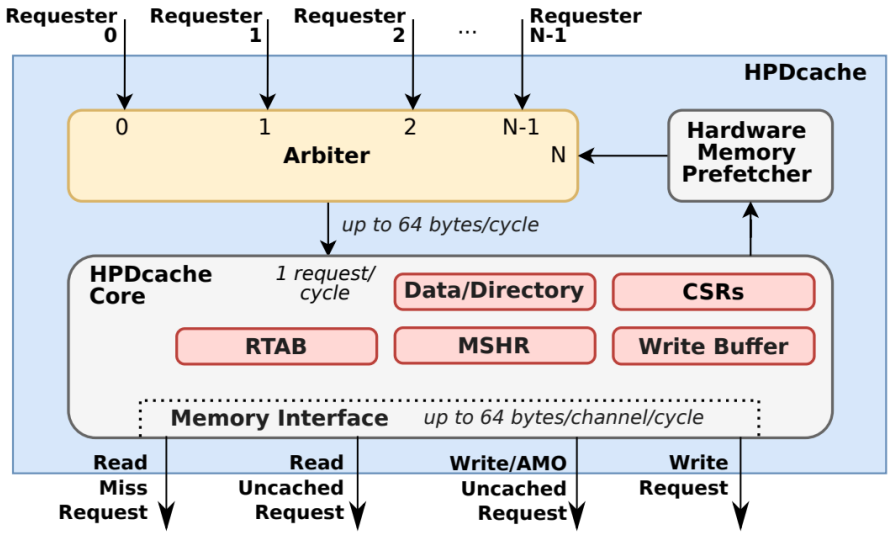


Figure 3.27: High-performance cache block diagram [11]

The HDPcache is shown in Figure 3.27. The HDPcache has a number of request ports, which are managed by an arbiter. The arbiter interacts with the HDPcache core. The cache core has a data directory, a miss status holding directory (MSHR) that handles read misses, a replay table (RTAB), which is used to deal with blocking conditions and allow out-of-order processing, and a write buffer. For more details, see [11].

```
Command: write_cfgmem -format mcs -interface SPIx4 -size 256 -loadbit {up 0x0 wo
Creating config memory files...
Creating bitstream load up from address 0x00000000
Loading bitfile work-fpga/ariane_xilinx.bit
Writing file work-fpga/ariane_xilinx.mcs
Writing log file work-fpga/ariane_xilinx.prm
===================================
Configuration Memory information
===================================
File Format        MCS
Interface          SPIX4
Size               256M
Start Address      0x00000000
End Address        0x0FFFFFFF

Addr1         Addr2         Date                    File(s)
0x00000000    0x00AE9D9B    Apr 28 08:44:28 2025    work-fpga/ariane_xilinx.bit
0 Infos, 0 Warnings, 0 Critical Warnings and 0 Errors encountered.
write_cfgmem completed successfully
```

Figure 3.28: Bit-stream generation using Vivado

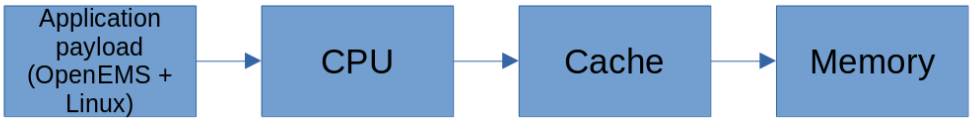


Figure 3.29: High-performance cache block diagram

We will run the application payload on the CPU with cache enabled and we plan to report on results in the future. Figure 3.29 shows a high-level view of the testing setup: an application payload, consisting of OpenEMS and Linux, accesses the memory cached through the cache by CPU instructions that use memory. In particular, we plan to benchmark the time from booting until the first report of energy data in OpenEMS, as it has been set up for WP5 in a scenario without the high-performance caching.

## 3.2 Monitoring Infrastructure

Task 3.3 of WP3 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.

A multicore statistics unit (BSC) is integrated as part of the safety island, while context-aware performance monitoring counters are extended with software 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.

Finally, 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 Partner | Dependencies | Licensing |
|---|---|---|---|
| CA-PMC-IF (3.2.1) | TRT | CA-CORE (WP2), CA-BUS (WP2), CA-PMC (3.1.6) | Permissive open-source |
| RTPM (3.2.2) | POLIMI | Monitored IP | Proprietary |
| SafeSU (3.2.3) | BSC | None | Permissive open-source |
| TCCP (3.2.4) | OFFIS | None | Permissive open-source |

Table 3.8: Overview of Task 3.3 contributions

### 3.2.1 Context-Aware PMC Interface (CA-PMC-IF) – TRT

#### 3.2.1.1 IP Card

| | | |
|---|---|---|
| **Basic Info** | | |
| *IP name* | Context-Aware PMC Interface (CA-PMC-IF) | |
| *License* | Open-source (SolderPad Hardware License v0.51) | |
| *Repository* | https://github.com/ThalesGroup/cva6-context-aware-monitoring.git | |
| **Architecture** | | |
| *Clock* | Number of clock domains | 1 |
| | Synchronous with system | Y |
| | Clock generated internally | N |
| *Ctrl Interface* | ISA extension? | N |
| | Memory mapped? | Y |
| | Protocol | AXI4 |
| | Address Map | Y (see Section 3.2.1.5) |
| *Initiator Interface* | Protocol | N/A |
| | Cached? | N |
| | IOMMU? | N |
| *Interrupts* | Interrupts | Y (generates 1 interrupt) |
| **Microarchitecture** | | |
| *Parametrization* | Parametric no. cores? | N |
| | Parameteric config? | Y |
| *Programmability* | Contains programmable cores? | N/A |
| | N/A | |
| **Software** | | |
| *Compiler* | Requires specialized compiler? | N |
| | Compiler repository | - |
| *Hardware Abstraction Layer* | N/A | |
| *High-level API* | Is there a high-level API/SDK? | N |
| | SDK repository | N |
| | Is there a domain-specific compiler? | N |
| **Integration** | | |
| *IP Distribution* | Manifest type (if any) | (to be defined) |
| | Standalone simulation? | Y |
| | (if standalone sim) SW requirements? | Verilator |
| | Integration documented / examples? | Y |
| *Synthesis* | Is the IP synthesizable? | Y |
| | FPGA synthesis example available? | Y (as component in CVA6 based design, to be provided) |
| | ASIC synthesis example available? | N |
| *Simulation* | Closed-source simulation? | N |
| | Open-source simulation? | Y (Verilator) |
| *Evaluation* | PPA results available? | N |

#### 3.2.1.2 General Information

The CA-PMC-IF (short name for CA-PMC Interface) IP is part of the Context Aware Monitoring framework developed by TRT. For an overview of the CA-PMC-IF IP and its function in the Context

Aware Monitoring framework refer to Section 3.1.6.2 in page 38.

#### 3.2.1.3 Purpose and Scope

The purpose of the CA-PMC-IF IP is to provide a way to program/control instantiations of CA-PMC modules, e.g. the CA-PMC module of the CVA6 [29, 28, 17] RISC-V core, the CA-PMC module of the cache level 2, etc. As such, the CA-PMC-IF is to be instantiated by CA-PMC modules. The CA-PMC-IF module then provides and manages the counters of the CA-PMC module, and it also provides the interfaces to access and configure these counters, leaving the CA-PMC module the task to provide the events that will be counted.

The CA-PMC-IF module currently provides AXI 4 or AXI 4 Lite interfaces, which one is used depends on the targeted design. Other interfaces might be provided/implemented if required. The AXI interface provides an easy and standard way to connect the CA-PMC-IF (and thus the CA-PMC) to an AXI bus so it can be accessed by cores connected to it, e.g. a supervision core. The CA-PMC-IF address map provides means to configure and retrieve the different counters the CA-PMC instantiating it provides.

Finally, the CA-PMC-IF also provides mechanisms to generate and manage interrupts when the counters overflow. A single interrupt output signal is provided. This signal is to be connected to an interrupt controller, e.g. PLIC.

#### 3.2.1.4 Refined architecture description

An initial architecture of the CA-PMC-IF hardware component was presented in the Section 3.3.1 of Deliverable D3.1, and updated in Figure 3.30. Compared to the previous version, the current CA-PMC-IF has integrated the counters logic, leaving only the events generation to the CA-PMC. This simplifies the creation of new CA-PMC modules, the CA-PMC-IF taking care of the programming interface, the counters management and the interrupts generation.

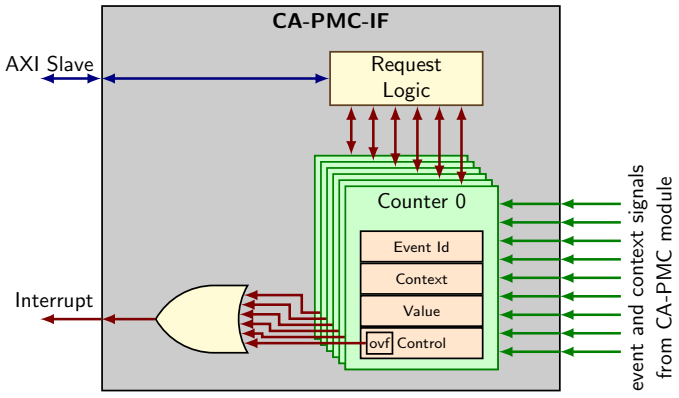


Figure 3.30: CA-PMC-IF architecture

At the core of the CA-PMC-IF are the counters. Each counter contains register to select the event the counter has to count and the value of the counter, i.e. the number of events that have ocurred. Moreover, to control the context information a register with the current context is stored

in the counter. The value field will be only updated when the traced event occurs in the stored counter context. Each counter can generate one interrupt when the value register overflows, this information is stored in a dedicated one-bit field of the counter control register (labelled as “ovf” in Figure 3.30). The control register is used to enable/disable the counter, indicate if counter should consider or not context information, to enable/disable the generation of interrupts when the value register in the counter overflows and to check if the counter produced the overflow interrupt and clean it.

The number of counters in a CA-PMC-IF instantiation is parameterizable at design time, with a limit of 256 in the current design. The overflow interrupt generated by each of the counters is combined into a single output signal by an or gate. To feed the counters with events and the context information, the CA-PMC-IF simply forwards the information received through a dedicated interface from the associated CA-PMC module to the counters.

Finally, the CA-PMC-IF provides an AXI interface to programmatically control the counters by writing and reading them. Section 3.2.1.5 describes the address map provided by CA-PMC-IF for that purpose. The CA-PMC-IF processes the AXI requests through a dedicated logic and reads/writes the counters registers accordingly.

#### 3.2.1.5 Memory-Mapped Interface

As described in Section 3.2.1.4 the main purpose of the CA-PMC-IF module is to provide a unified counter infrastructure for CA-PMC modules and provide its memory-mapped interface to control and read the CA-PMC counters.

**Global CA-PMC-IF memory layout**

Table 3.9 presents the memory layout of the CA-PMC-IF memory-mapped interface.

| Address | Name | Size | Description |
|---|---|---|---|
| BASE + 0x00 | capabilities | 16 Bytes | Register describing CA-PMC-IF instance features |
| BASE + 0x10 | *reserved* | 16 Bytes | Reserved for future usage |
| BASE + 0x20 | overflow | 32 Bytes | Register mapping the counters overflow status |
| BASE + 0x40 | counters | Up to 8192B | Up to 256 memory areas of 32 Byte for each counter |

Table 3.9: CA-PMC-IF memory-mapped interface

The CA-PMC-IF specific memory regions are decomposed into:

- **capabilities** register: This register provides information on the capabilities provided by the specific instance of the CA-PMC-IF IP, such as the supported context width, the available number of counters and so on.
- **overflow** register: This register maps the CA-PMC-IF counters **control** register overflow bit (**counter.control[ovf]**).
- **counters**: This memory area holds memory areas for each counter provided by the CA-PMC-IF instance.

More details about each memory section are presented below.

#### capabilities register layout

The **capabilities** register provides read-only information about the CA-PMC-IF instance. Table 3.10 shows the layout of this register.

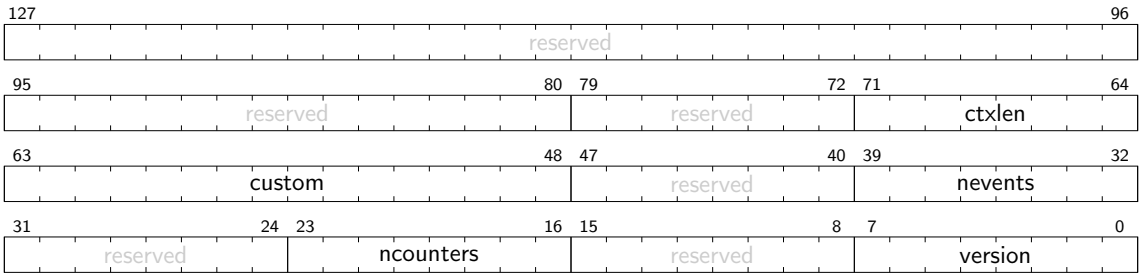


Table 3.10: CA-PMC-IF capabilities memory layout

The whole register is read only, any write is ignored. The register fields are defined as follows:

- **version** (bits 0 to 7): The first byte provides the CA-PMC-IF specification version the CA-PMC-IF instance follows. This document describes the first version of such specifications, i.e. **version** is equal to 1.
- **ncounters** (bits 16 to 23): This field indicates the number of counters supported by the IP.
- **nevents** (bits 32 to 39): This field indicates the number events supported by this IP. This field also limits the event to count that can be set on the counters **event** register (**counter.event**), i.e. the **counter.event** register can be set from 0 to **nevents**-1.
- **custom** (bits 48 to 63): Reserved for custom extensions of the CA-PMC-IF interface. Different than zero if used, otherwise all bit should be set to zero.
- **ctxlen** (bits 64 to 71): Width of the software context value supported by the IP. The maximum value this field can have in the current CA-PMC-IF specification is 32, for a 32-bit wide software context.

Unused bits are reserved and should read as 0, and writes on them are ignored.

#### overflow register

This 32-byte, i.e. 256 bits, register maps the overflow bit field of the counters **control** memory region (**counter.control.ovf**) provided by the IP. Table 3.11 shows its layout. Bit 0 of this register maps counter 0 control register overflow bit (eighth bit of the counter **control** register, **counter.control.ovf**), bit 1 counter 1 overflow bit, bit 2 counter 2, and so on, up to **capabilities.ncounters**. If **capabilities.ncounters** is smaller than 256, bits **capabilities.ncounters** to 255 of the **overflow** register are mapped to the value 0 and can not be modified.

Reading any bit of the **overflow** register gets the current value of the corresponding counter overflow bit (**counter.control.overflow**), or 0 if it is not mapped to any counter. Writing a 0 to any bit of the **overflow** register does not modify the corresponding counter overflow bit, and is ignored if the bit is not mapped to any counter. Writing a 1 to any bit of the **overflow** register does set the

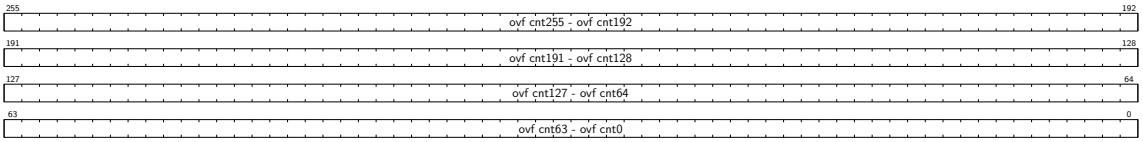


Table 3.11: CA-PMC-IF overflow register layout

corresponding counter overflow bit to 0, and is ignored if the bit is not mapped to any counter.

### Counters memory layout

The **counters** memory region provides the registers of each of the performance counters provided by the CA-PMC-IF, as defined by **capabilities.ncounters**. Each performance counter, **counter**, memory area is organized as shown in Table 3.12.

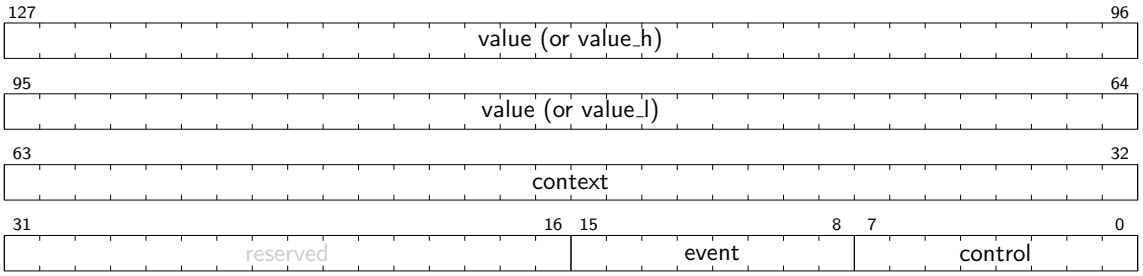


Table 3.12: CA-PMC-IF **counter** memory layout

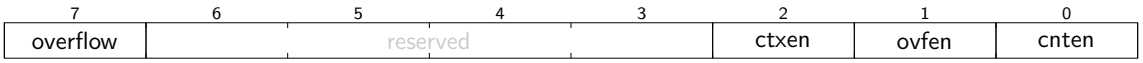


Table 3.13: CA-PMC-IF **counter.control** memory layout

The following are the descriptions of the fields of each counter register:

- **control** register: A R/W register to control performance counter behavior, enabling/disabling counting, overflow or context, and to check and clear the overflow interrupt status. Table 3.13 shows the **counter.control** register layout:
  - **cnten** field (bit 0): Enables (1) or disables (0) event counting.
  - **ovfen** field (bit 1): Enables (1) or disables (0) the generation of overflow interrupt on overflow of the **counter.value** field.
  - **ctxen** field (bit 2): Enables (1) or disables (0) the filtering of events to count based on the context.
  - **overflow** field (bit 7): This a status bit indicating if the counter has produced an overflow interrupt. The counter interrupt signal maps the value in this bit. An overflow interrupt is generated when increasing **counter.value** because of the ocurrence of the monitored

event and resulting **counter.value** is smaller than before. This bit remains set to 1 once and interrupt has been generated. To set this bit to 0, i.e. to acknowledge the interrupt, a write request with a value of 1 on this bit needs to be issued. Writing a 0 to this bit does not modify its status.

- **event** register: A R/W 8-bit register specifying the hardware event to be monitored. Writing a value equal or bigger than **capabilities.nevents** sets this field to an implementation defined value smaller than **capabilities.nevents**.
- **context**: A R/W 32-bit register specifying the context value used restrict hardware event counting to specific software contexts.
- **value**: A R/W 64-bit register increased each time the monitored event (**counter.event**) occurs. This register can be accessed with a 64-bit request or with two 32-bit requests to its subparts: **counter.value_h** for the higher 32-bits and **counter.value_l** for the lower 32-bits. Writing this field is not recommended when the counter is enabled (**counter.control.cnten** equals to 1), specially when writing using 32-bit write requests. When using 64-bit read requests the **counter.value** register can be read on a single 64-bit read request. When using 32-bit read requests the **counter.value_h** and **counter.value_l** registers should be read when the counter is disabled (**counter.control.cnten** equals to 0) or if enabled using the following sequence to ensure the correctness of the read 64-bit value:
  1. read **counter.control.value_h**,
  2. read **counter.control.value_l**,
  3. read **counter.control.value_h**,
  4. if value read in step 1 is different than the value in step 3 then repeat the sequence (i.e. go back to step 1), otherwise the read of the performance counter value is finished.

Reserved bits are set to zero, and write accesses are ignored.

#### 3.2.1.6 Hardware interfaces

The CA-PMC-IF provides three interfaces:

1. An AXI or AXI-Lite bus slave interface (slave): This interface allows to access the CA-PMC-IF and the counters it provides programmatically using the memory map described in Section 3.2.1.5. This interface is parameterizable at design time providing a full AXI interface or a simplified AXI-Lite, and to define the bus characteristics, like address and data width.
2. A output interrupt signal (output): This interface provides a single bit signal indicating that at least one of the CA-PMC-IF counters has produced an overflow interrupt, see Section 3.2.1.4.
3. Monitored module event and context signals (input): This interface allows the monitored CA-PMC module (e.g. a core, a cache module, etc.) to provide to the CA-PMC-IF module the events produced at each cycle and their context. The number of input events the CA-PMC-IF is parameterizable, and each event signal is one width wide, i.e. event occured during a cycle or not. The context signal is currently shared by all the events, i.e. there is a single context signal. The width of the context signal should match the **capabilities.ctxlen** value (see Section 3.2.1.5 *Global CA-PMC-IF memory layout* in page 74).

#### 3.2.1.7 ISA specialization

The CA-PMC-IF IP does not require any modification the ISA. However, to exploit the context monitoring capability IPs generating events should be capable to pass the context information (through the context signal presented in Section 3.2.1.6). Generators of this context information might need ISA modifications to generate it, as proposed by the CA-CORE extension developed in Task T2.3.

#### 3.2.1.8 Evaluation prototype

The CA-PMC-IF IP has been evaluated in a dedicated Verilog testbench to verify its operation. Furthermore, it has been integrated into a CVA6 CA-PMC module to evaluate its integration, synthesis and operation. For further information on that integration refer to Section 3.1.6.7 (page 42).

## 3.2.2 Run-Time Power Monitoring Instrumentation (RTPM) – POLIMI

### 3.2.2.1 IP Card

| | | |
|---|---|---|
| | **Basic Info** | |
| *IP name* | Run-Time Power Monitoring Instrumentation | |
| *License* | Closed-source proprietary | |
| *Repository* | | |
| | **Architecture** | |
| *Clock* | Number of clock domains | 1 |
| | Synchronous with system | Y |
| | Clock generated internally | N |
| *Ctrl Interface* | ISA extension? | N |
| | Memory mapped? | Y |
| | Protocol | AXI |
| | Address Map | TBD |
| *Initiator Interface* | Protocol | AXI |
| | Cached? | N |
| | IOMMU? | N |
| *Interrupts* | Interrupts | No |
| | **Microarchitecture** | |
| *Parametrization* | Parametric no. cores? | N |
| | Parameteric config? | Y |
| *Programmability* | Contains programmable cores? | N |
| | ISA | None |
| | **Software** | |
| *Compiler* | Requires specialized compiler? | N |
| | Compiler repository | |
| *Hardware Abstraction Layer* | N/A? | |
| *High-level API* | Is there a high-level API/SDK? | N |
| | SDK repository | |
| | Is there a domain-specific compiler? | N |
| | **Integration** | |
| *IP Distribution* | Manifest type (if any) | TBD |
| | Standalone simulation? | TBD |
| | (if standalone sim) SW requirements? | TBD |
| | Integration documented / examples? | TBD |
| *Synthesis* | Is the IP synthesizable? | Y |
| | FPGA synthesis example available? | Y |
| | ASIC synthesis example available? | N |
| *Simulation* | Closed-source simulation? | Y (AMD Vivado) |
| | Open-source simulation? | N |
| *Evaluation* | PPA results available? | N |

### 3.2.2.2 General Information

The automatic generation of the run-time power monitoring infrastructure delivers the capability to provide periodic power estimates from the switching activity of a set of signals in the monitored components.

#### 3.2.2.3 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.

#### 3.2.2.4 Refined Architecture Description

The monitoring infrastructure consists of a set of switching activity counters attached to corresponding inputs and output signals of the system components 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.

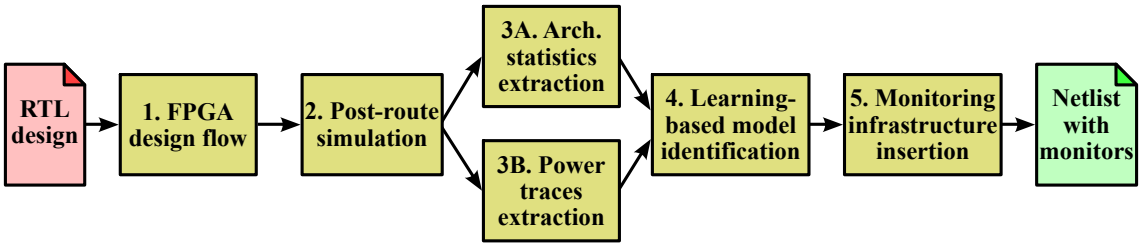


Figure 3.31: Flowchart diagram of the flow for run-time power monitoring.

The automatic instrumentation of a run-time power monitoring infrastructure is carried out by taking the RTL description of the target computing platform and its corresponding testbench as inputs and producing a gate-level netlist enhanced with power monitoring capabilities. This process, depicted in Figure 3.31, can be divided into five sequential phases, each contributing to the final implementation of the monitoring infrastructure.

1. **Target implementation** – The first phase utilizes the standard hardware design flow to process the RTL description of the target platform. This includes synthesis, placement, and routing steps, resulting in a gate-level netlist of the design. This netlist serves as the foundation for all subsequent stages of the framework.
2. **Design simulation** – In the second phase, the post-route gate-level netlist is subjected to a detailed simulation that exercises all parts of the design using a comprehensive input dataset. The goal is to stress the entire computing platform to ensure thorough coverage of its functionality. This phase outputs a value change dump (VCD) file, which contains detailed records of the switching activity of the design, enabling the calculation of power consumption traces. However, the time required for simulation grows significantly with the complexity of the target platform.
3. **Data extraction** – The third phase processes the VCD file to extract relevant information about the switching activity and to compute the power consumption traces associated with the target design. This step is computationally intensive, particularly for large platforms or

when higher temporal resolution is required for the power traces. The resulting power traces and switching activity data form the basis for model identification in the next phase.

4. **Model identification** – The fourth phase involves creating a power model tailored for runtime monitoring. This model identifies a subset of signals from the target platform and assigns them corresponding weights by analyzing the correlation between the extracted switching activity and the computed power traces. The objective is to minimize the difference between the actual power trace and the one generated by the identified power model, while also satisfying constraints such as limiting the number of signals used as model inputs, minimizing the area and power overheads of the corresponding monitors [8], and avoiding side-channel leakage [30].
5. **Monitor implementation** – In the fifth and final phase, the framework automatically instruments the identified power model into the original design. The signals selected by the model are wrapped in additional hardware that periodically samples their switching activity and calculates the corresponding power consumption. This calculated power consumption is made accessible through an architectural register, enabling real-time power monitoring without significant interference with the platform's primary functionality.

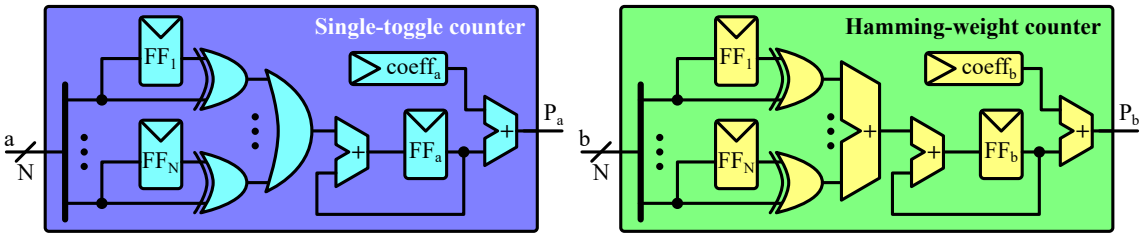


Figure 3.32: Architecture of single-toggle and Hamming-weight counters measuring the switching activity of a multi-bit signal.

The architecture of the toggling activity counters can be implemented in two different ways, depending on whether they measure single-toggle or Hamming-weight counts, as shown in Figure 3.32.

A single-toggle counter counts whether there is at least one change in the signal on the monitored physical wire. It takes the multi-bit input signal and combines a XOR-tree network with an OR gate to output 1 or 0 depending on whether at least one bit of the signal toggled, and the produced value is added to the cumulative switching activity.

Conversely, a Hamming-weight counter counts instead the number of bits in the signal that switched their values. It takes the multi-bit input signal and combines a XOR-tree network with an adder to output the number of switching bits, adding the latter value to the cumulative switching activity.

Concerning area overhead, the number of flip-flops for Hamming-weight counters increases with the size of the monitored signal and the number of clock cycles in the time window, while in the case of single-toggle counters it depends only on the time window, since for each clock cycle they perform at most a unitary increment. Hamming-weight counters also show a higher power overhead, due to their higher switching activity.

The run-time monitoring infrastructure instantiated by the automatic generation flow may contain both single-toggle and Hamming-weight counters. Regardless of the type of such counters, their switching activity values are multiplied by their respective model weights and aggregated to obtain the estimated power consumption at each time interval.

#### 3.2.2.5 Memory-Mapped Interface

An AXI memory-mapped register provides, at each time interval, the estimate of power consumption computed by the run-time power monitoring infrastructure. This register holds the estimated power consumption value provided by the run-time power monitoring infrastructure at each time interval. The estimate is periodically updated by the monitoring infrastructure, according to the selected time resolution. The register is read-only and can be accessed by the rest of the system through a read operation at the corresponding memory-mapped address.

#### 3.2.2.6 ISA Specialization

No ISA specialization is required, as the IP can be accessed via a standard AXI memory-mapped read operation.

#### 3.2.2.7 Evaluation Prototype

The effectiveness of the automatic instrumentation of a run-time power monitoring is evaluated, according to a variety of quality metrics, on a set of HLS-generated accelerators. The quality of the instantiated monitoring infrastructures is measured through their area and power overheads as well as the root-mean-square error (RMSE) of their estimations of run-time power consumption.

The target HLS-generated accelerators are implemented on an AMD Artix-7 100 FPGA that features 63400 lookup tables (LUTs), 126800 flip-flops (FFs), 240 digital signal processing (DSP) elements, and 135 36kb blocks of block RAM (BRAM).

The HLS-generated designs include accelerators for the AES, Blowfish, GSM, and MIPS applications of the CHStone [16] benchmark suite for C-based HLS. The AMD Vivado toolchain was employed for HLS, RTL synthesis and implementation, bitstream generation, FPGA programming, and simulation.

The experiments explored nine combinations of the four HLS-generated accelerators, producing a set of designs with a large variability in their use of the FPGA resources and in their maximum achievable operating frequency. The HLS-generated designs operate at clock frequencies ranging from 100 MHz to 150 MHz, consume between 0.18 W and 0.50 W, and occupy up to 84%, 57%, 69%, and 100% of the LUT, FF, BRAM, and DSP resources provided by the target FPGA chip.

The identification of the power model and the evaluation of power estimates obtained by the monitoring infrastructure are carried out by splitting the collected dataset of power traces and microachitectural statistics according to a 80:20 ratio and targeting a 10µs temporal resolution.

The first-order linear model takes as its inputs the switching activity of a selected subset of input and output signals of the modules in the design hierarchy and outputs the estimated power consumption. The hardware counters that monitor the selected signals are either single-toggle ones, counting any change in the target multi-bit signal, or Hamming-weight counters, that count the number of bits that toggled their values.

The experimental results show area and power overheads below 3% and an RMSE below 5%, on par with the run-time power monitoring solutions from the open literature [31].

### 3.2.3 Safety-Related Statistics Unit (SafeSU) – BSC

#### 3.2.3.1 IP Card

| | | |
|---|---|---|
| **Basic Info** | | |
| *IP name* | SafeSU | |
| *License* | Open-source (MIT License) | |
| *Repository* | https://github.com/bsc-loca/SafeSU/tree/9b7f520a258a97b462e6a64931147246c5e1743e | |
| **Architecture** | | |
| *Clock* | Number of clock domains | 1 |
| | Synchronous with system | Y |
| | Clock generated internally | N |
| *Ctrl Interface* | ISA extension? | N |
| | Memory mapped? | Y |
| | Protocol | APB 32b |
| | Address Map | Base (0x80100000) |
| *Initiator Interface* | Protocol | APB 32b |
| | Cached? | N |
| | IOMMU? | N |
| *Interrupts* | Interrupts | Y |
| **Microarchitecture** | | |
| *Parametrization* | Parametric no. units? | N |
| | Parameteric config? | Y |
| *Programmability* | Contains programmable cores? | N |
| | ISA | - |
| **Software** | | |
| *Compiler* | Requires specialized compiler? | N |
| | Compiler repository | - |
| *Hardware Abstraction Layer* | N/A | |
| *High-level API* | Is there a high-level API/SDK? | Y |
| | SDK repository | https://github.com/bsc-loca/SafeSU/tree/main/drivers |
| | Is there a domain-specific compiler? | N |
| **Integration** | | |
| *IP Distribution* | Manifest type (if any) | Y (Readme.md) |
| | Standalone simulation? | Y |
| | (if standalone sim) SW requirements? | QuestaSim, Vivado 2020.2, Verilator |
| | Integration documented / examples? | - |
| *Synthesis* | Is the IP synthesizable? | Y |
| | FPGA synthesis scripts/example available? | Y |
| | ASIC synthesis scripts/example available? | N |
| *Simulation* | Closed-source simulation? | Y (QuestaSim) |
| | Open-source simulation? | Y |
| *Evaluation* | PPA results available? | DOI 10.1109/ETS50041.2021.9465444 |

#### 3.2.3.2 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.

#### 3.2.3.3 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.

#### 3.2.3.4 Place in the System

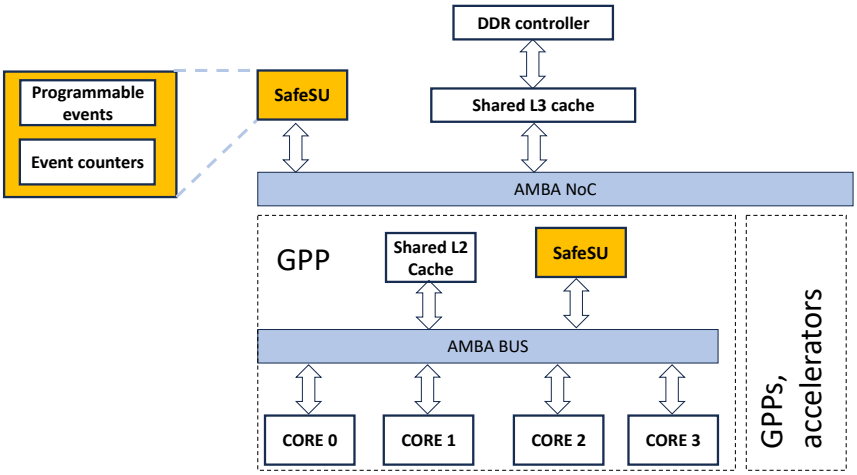


Figure 3.33: SafeSU system integration

The monitoring interface of the SafeSU depicted in Figure 3.33, 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.

#### 3.2.3.5 Architecture

The main components of the SafeSU are described next and can be seen in the block diagram of Figure 3.34:

- 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 a 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.

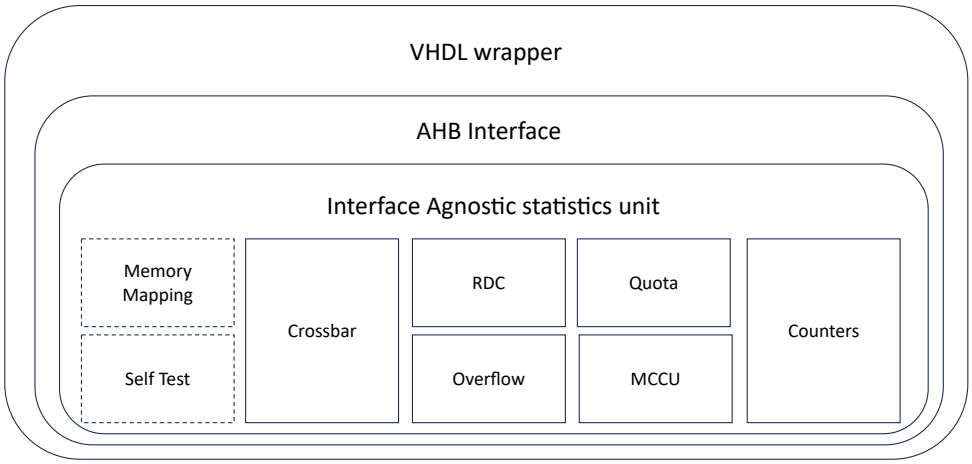


Figure 3.34: SafeSU architecture block diagram

#### 3.2.3.6 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

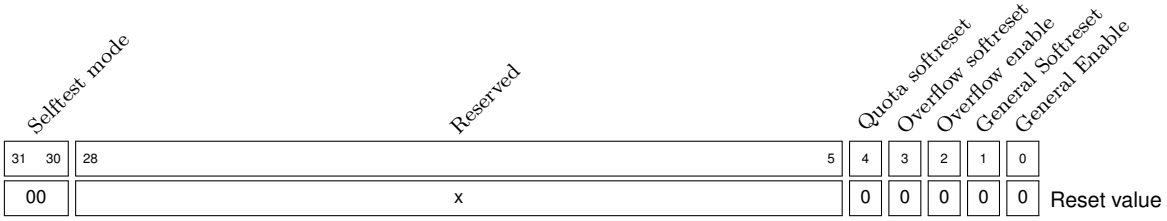


Figure 3.35: SafeSU - Base Configuration Register (0x000)

Reset and enable of overflow, quota, and regular counters' operations can be performed with the Base Configuration Register shown in Figure 3.2.3.6. 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 3.2.3.6:

- 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

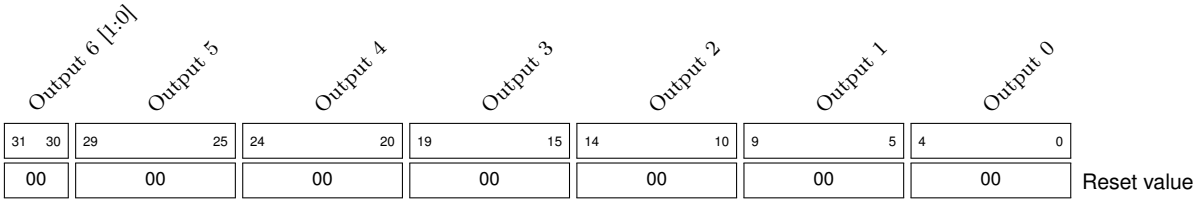


Figure 3.36: SafeSU - Crossbar Configuration Register 0 (0x0AC)

This feature allows routing any of the input signals of the SafeSU into any of the 24 counters of the SafeSU (see Table 3.14). Each one of the counters has a 5-bit configuration value. These values are stored in the registers shown in Figures 3.36, 3.37, 3.38 and 3.39. All the configuration values are consecutive. Thus, some values may have configuration bits in two consecutive

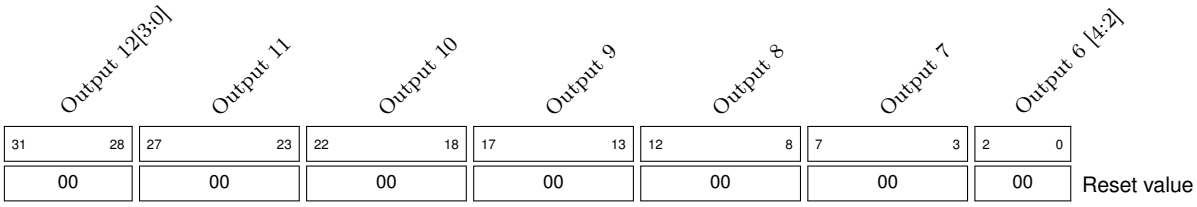


Figure 3.37: SafeSU - Crossbar Configuration Register 1 (0x0B0)

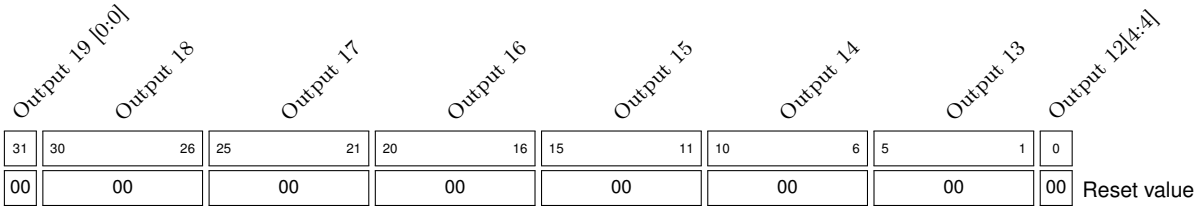


Figure 3.38: SafeSU - Crossbar Configuration Register 2 (0x0B4)

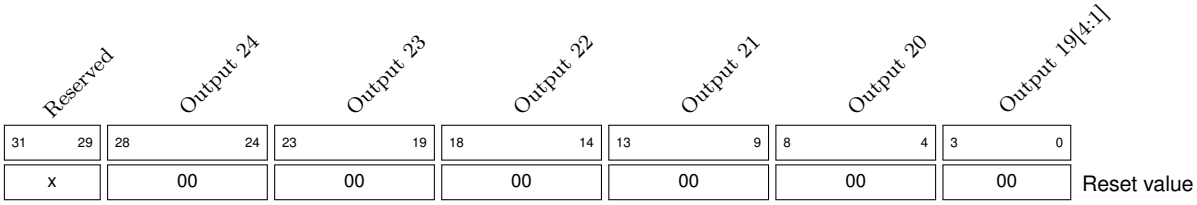


Figure 3.39: SafeSU - Crossbar Configuration Register 3 (0x0B8)

memory addresses. Examples of this are Output 6, 12, and 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, etc.

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 3.36 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.

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 3.40. If a counter with overflow detection active wraps over the maximum value, the corresponding bit of the Overflow Interrupt Vector register depicted in Figure 3.41 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.

| 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 3.14: SafeSU - Crossbar outputs and SafeSU capabilities

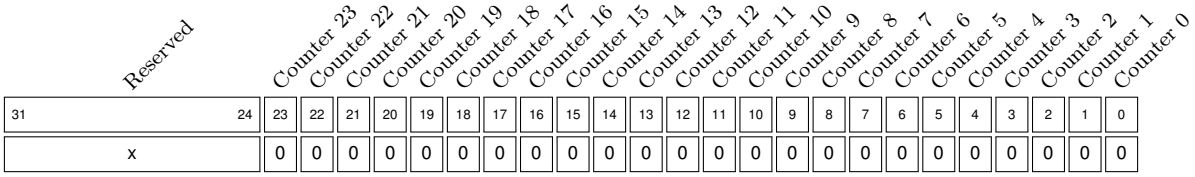


Figure 3.40: SafeSU - Overflow Interrupt Enable Mask (0x064)

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 3.42 shows the internal elements required to monitor the quota consumption of one core,

| Reserved | Counter 23 | Counter 22 | Counter 21 | Counter 20 | Counter 19 | Counter 18 | Counter 17 | Counter 16 | Counter 15 | Counter 14 | Counter 13 | Counter 12 | Counter 11 | Counter 10 | Counter 9 | Counter 8 | Counter 7 | Counter 6 | Counter 5 | Counter 4 | Counter 3 | Counter 2 | Counter 1 | Counter 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 31 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |

Reset value

Figure 3.41: SafeSU - Overflow Interrupt Vector (0x068)

given that there are four input events. When the events become active, they pass the value assigned in the weight register depicted in Figure 3.43 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.

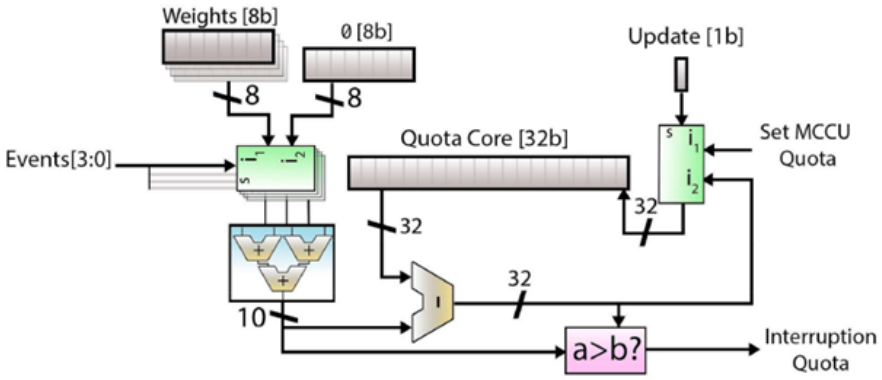


Figure 3.42: SafeSU - Block diagram of the MCCU mechanism for one core

| Input 3 | Input 2 | Input 1 | Input 0 |
|---|---|---|---|
| 31 24 | 23 16 | 15 8 | 7 0 |
| 00 | 00 | 00 | 00 |

Reset value

Figure 3.43: SafeSU - MCCU Event Weights Register 0 (shared with RDC; 0x098)

In the current release, the MCCU can be reset and activated with the respective fields of the MCCU Main Configuration register depicted in Figure 3.44. 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.

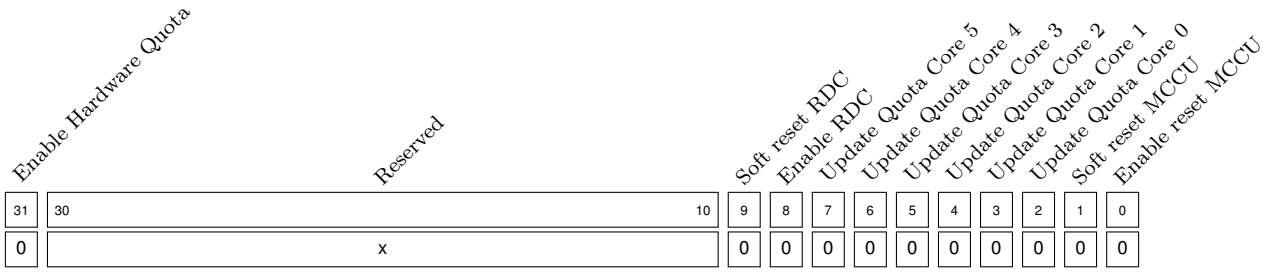


Figure 3.44: SafeSU - MCCU Main Configuration (0x074)

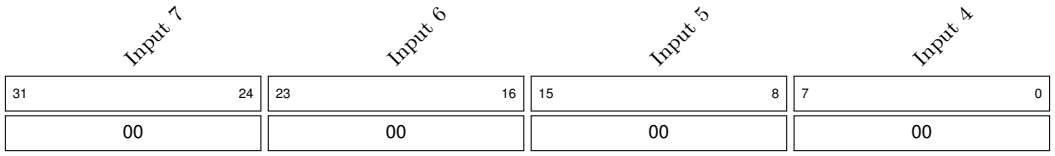


Figure 3.45: SafeSU - MCCU Event Weights Register 1 (shared with RDC; 0x09c)

More events can be provided in future releases. Table 3.14 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 3.43 and 3.45. 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 3.46 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.

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 3.48. 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 3.49. The RDC shares the main configuration register with the MCCU (Figure 3.44). 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 3.47), 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.

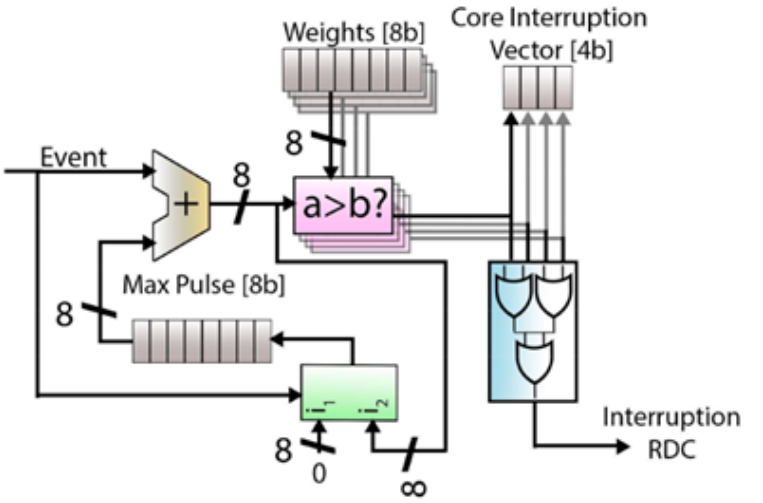


Figure 3.46: SafeSU - Block diagram of the RDC mechanism

| Reserved | | | Core 3 | Core 2 | Core 1 | Core 0 |
|---|---|---|---|---|---|---|
| 31 | | 4 | 3 | 2 | 1 | 0 |
| x | | | 00 | 00 | 00 | 00 |

Figure 3.47: SafeSU - RDC Interrupt Vector (0x0A0)

| Input 3 | | Input 2 | | Input 1 | | Input 0 | |
|---|---|---|---|---|---|---|---|
| 31 | 24 | 23 | 16 | 15 | 8 | 7 | 0 |
| 00 | | 00 | | 00 | | 00 | |

Figure 3.48: SafeSU - RDC Event Weights Registers 0 and 1 (shared with MCCU; 0x098, 0x09C)

| Input 7 | | Input 6 | | Input 5 | | Input 4 | |
|---|---|---|---|---|---|---|---|
| 31 | 24 | 23 | 16 | 15 | 8 | 7 | 0 |
| 00 | | 00 | | 00 | | 00 | |

Figure 3.49: 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 to specific physical addresses upon

integration into the platform. The hypervisor (XtratuM Next Generation, also known as XNG, 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. XNG guarantees this behavior, building on the MMU existing in the NOEL-V cores, which also realizes the RISC-V ISA hypervisor extension. Overall, the XNG 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.

In order to allow a partition to manage the SafeSU device from a high-level perspective, a driver at the hypervisor level has been developed and integrated by FENTISS within XNG, implementing a set of hypercalls. The new hypercalls defined to manage the SafeSU device are specified in Table 3.15, in which a description is included, and can be used by a partition with SafeSU access permissions.

For defining if a partition has access to the SafeSU device a new attribute in the XtratuM Configuration Files (XCF) has been added. In the case that "safesu" attribute is present in the XCF of a partition, that partition would be allowed to access SafeSU registers and the implemented hypercalls could be freely used. Otherwise, if the attribute is not set, if the partition tries to use any of the SafeSU hypercalls, an error code meaning invalid configuration will be returned.

Finally, all the interrupts which generate the SafeSU device need to be delegated to the partition in the XCF, so the partition can receive them. It is also important to note that the partition should allow the interrupts reception, unmask the desired interrupts and install the desired handlers to be triggered when an interrupt is received. These interrupts' management can be done through the use of other XNG's hypercalls.

#### 3.2.3.7 Evaluation Prototype

Next, we evaluated the SafeSU by integrating two instances in the SELENE SoC and implemented on a Xilinx FPGA operating at 100 MHz, as seen in Figure 3.33. One instance of the SafeSU, which we called SafeSU_AHB is connected to the AMBA AHB Bus and will calculate the contention occurring in this interface. Contention will occur when multiple cores are accessing the bus, as this interface only allows one manager operation at a time, and the other managers will be waiting to get granted access. At this level, the SafeSU_AHB will also be using the quota mechanism, MCCU, together with hardware interruptions when the user-defined quotas for each core are exceeded.

Evaluation of the integration will use a set of memory-stressing kernels on cores 2,3,4, while core 1 will be used as the critical application. Before starting execution on core 1, SafeSU configuration will be set with pre-defined quotas for cores 2,3,4 to ensure that when exceeded, they lose access to the bus. Execution times will be compared with the same execution but without quotas to observe that the SafeSU monitoring and quotas operation reduce the timing overhead of the critical application due to the contention suffered.

Similarly, the SafeSU connected to the AXI Bus, named SafeSU_AXI will monitor the contention happening at an outer level, in the AXI interface. Contention here will happen when cores accessing L2 create a miss and require access to main memory which is done through the AXI bus and

| Hypercall | Description |
|---|---|
| `retCode XSafeSuAssignEvent2Counter( counter, eventId)` | Assigns a specific event to a specific counter. |
| `retCode XSafeSuCountersEnable()` | Enables all SafeSU counters. |
| `retCode XSafeSuCountersDisable()` | Disables all SafeSU counters. |
| `retCode XSafeSuCountersReset()` | Resets all SafeSU counters. |
| `retCode XSafeSuGetCounterVal( counter, *value)` | Gets the current value of a specific counter. |
| `retCode XSafeSuRdcEnable()` | Enables the monitoring of the events' duration. |
| `retCode XSafeSuRdcDisable()` | Disables the monitoring of the events' duration. |
| `retCode XSafeSuRdcReset()` | Resets the status of the RDC. |
| `retCode XSafeSuRdcSetMaxDuration( counter, duration)` | Sets the maximum allowed duration for a given counter (compatible with RDC monitoring). |
| `retCode XSafeSuRdcReadWatermark( counter, *value)` | Reads the current maximum value for a given counter. |
| `retCode XSafeSuMccuEnable()` | Enables contention control. |
| `retCode XSafeSuMccuDisable()` | Disables contention control. |
| `retCode XSafeSuMccuReset()` | Resets the status of the MCCU. |
| `retCode XSafeSuMccuSetQuotaLimit( coreId, quota)` | Sets the maximum contention quota for a specific core. |
| `retCode XSafeSuMccuGetQuotaRemaining( coreId, *value)` | Gets the remaining contention quota for a specific core. |
| `retCode XSafeSuMccuSetEventWeights( counter, weight)` | Assigns weights to MCCU events for a given counter. |
| `retCode XSafeSuOverflowEnable()` | Enables counters overflow detection. |
| `retCode XSafeSuOverflowDisable()` | Disables counters overflow detection. |
| `retCode XSafeSuOverflowReset()` | Resets the status of the overflow submodule. |
| `retCode XSafeSuOverflowSetIrqMask( mask)` | Configures the overflow interrupt enable mask, enabling the overflow detection for the counters set in the mask. |
| `retCode XSafeSuOverflowGetIrqStatus( *mask)` | Gets the status of the active interrupts in the overflow submodule. |

Table 3.15: Hypercall table implemented within XNG for SafeSU device support

will *compete* with the accelerators integrated in the system.

The evaluation on this case will consist of only activating core 1, while the rest of the cores will be deactivated. Instead, the accelerator will be configured to access frequently the AXI bus. Then, the SafeSU_AXI will monitor the contention and, similarly to the other SafeSU use the quota mechanism to ensure that the contention created by the accelerator is limited.

## 3.2.4 Time Contract Monitoring Co-Processor (TCCP) – OFFIS

### 3.2.4.1 IP Card

| | | |
|---|---|---|
| **Basic Info** | | |
| *IP name* | Time Contract Monitoring Co-Processor (TCCP) | |
| *License* | Apache V2.0 | |
| *Repository* | not ready yet | |
| **Architecture** | | |
| *Clock* | Number of clock domains | 2 |
| | Synchronous with system | Y |
| | Clock generated internally | N |
| *Ctrl Interface* | ISA extension? | N |
| | Memory mapped? | Y |
| | Protocol | AXI |
| | Address Map | N.A. |
| *Initiator Interface* | Protocol | AXI |
| | Cached? | N |
| | IOMMU? | N |
| *Interrupts* | Interrupts | Y |
| **Microarchitecture** | | |
| *Parametrization* | Parametric no. cores? | N/A |
| | Parameteric config? | N/A |
| *Programmability* | Contains programmable cores? | Y |
| | ISA | RISC-V (cv64a6) |
| **Software** | | |
| *Compiler* | Requires specialized compiler? | Y |
| | Compiler repository | N/A (WP4) |
| *Hardware Abstraction Layer* | N/A | |
| *High-level API* | Is there a high-level API/SDK? | N |
| | SDK repository | - |
| | Is there a domain-specific compiler? | Y |
| **Integration** | | |
| *IP Distribution* | Manifest type (if any) | - |
| | Standalone simulation? | N |
| | (if standalone sim) SW requirements? | - |
| | Integration documented / examples? | - |
| *Synthesis* | Is the IP synthesizable? | Y |
| | FPGA synthesis example available? | N |
| | ASIC synthesis example available? | N |
| *Simulation* | Closed-source simulation? | N |
| | Open-source simulation? | Y (SystemC TCCP simulation) |
| *Evaluation* | PPA results available? | N |

### 3.2.4.2 General Information

As mentioned in Deliverable D3.1 (Section 3.3.4), this module serves as a modular and composable time contract monitoring co-processor. The monitoring approach builds upon previous work from the VE-VIDES project [10], a German-funded initiative, and extends earlier research [27]

conducted in EU-funded projects such as Productive4.0.

This co-processor is designed to support a formal Contract-Based Design (CBD) language [26]. It must handle fundamental timing properties, including aging, event occurrence, and reaction. Additionally, it should facilitate the monitoring of non-temporal properties, such as power consumption or ensuring that a specific controller maintains a parameter within a defined range over a given time frame. Examples include running closed-loop motor control in parallel with a computationally intensive task or verifying that a complex AI algorithm consistently completes its execution within a specified period.

Moreover, TCCP must be capable of monitoring multiple properties simultaneously. The functional and performance requirements of TCCP are detailed in Deliverable D1.2 (Section 3.3.2).

As shown in Figure 3.50, the contract-based runtime monitoring approach consists of three interacting components: At its core, the Time Contract Co-Processor (TCCP) connects to two other components, the TCCP-Compiler and the observer interfaces. The TCCP executes contract-based specifications in hardware, it observes various event sources via an observer interface. The observers are minimalistic adapters to source data, like a RISC-V trace port to observe computational progress or a memory content observer. The TCCP monitors events according to its programmable configuration derived from contract specifications. These specifications are processed by the TCCP compiler, which generates a configuration program for the TCCP. The TCCP compiler is build as part of WP4 in Task T4.3, while the target architecture is part of the automotive demonstrator in WP5, Task T5.2.

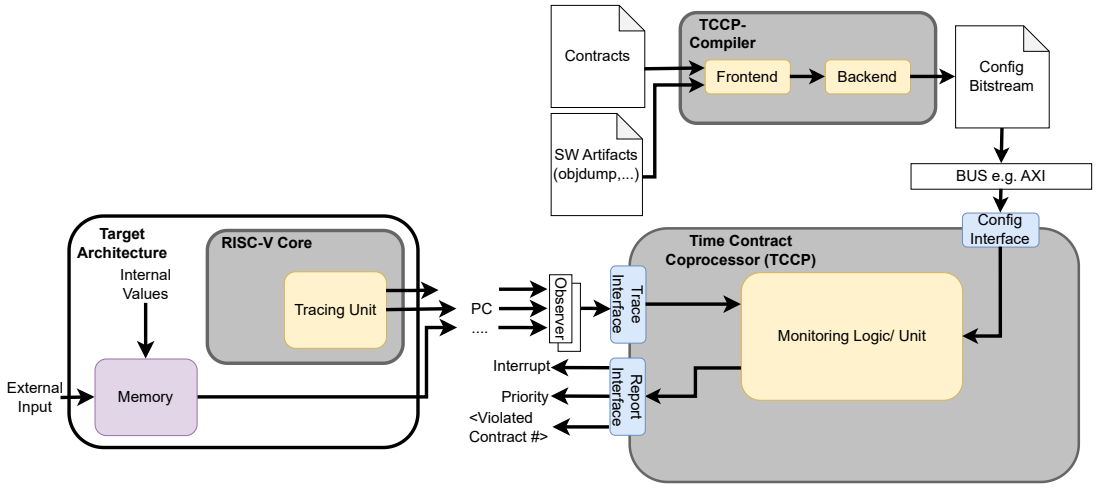


Figure 3.50: Time Contract Co-Processor (TCCP).

### 3.2.4.3 Purpose and Scope

The purpose of the TCCP is to advance the RISC-V ecosystem by developing high-performance architectures that incorporate the safety concept of contract-based monitoring. The scope in-

cludes multiple monitors for various properties, each with multiple interfaces, all operating simultaneously.

#### 3.2.4.4 Refined architecture description

The refined architecture of the Monitoring Unit (Figure 3.50), as depicted in Figure 3.51, illustrates the processing flow from the Trace Interface to the Report Interface, passing through the FIFO-, Contract Fetch-, and Contract Check modules. These modules may operate in different clock domains, indicated by color coding: light blue for the Monitor Clock domain and orange for the Observer Clock domain.

The Trace Interface collects multiple incoming streams from observers, each monitoring different properties at various points in the architecture. It transmits an event to the FIFO, consisting of an Observer ID, Location, Data Value, and timestamp. The Observer ID identifies the monitored property, the Location points to a program counter or other observed data, and the timestamp is generated using the TCCP clock or the target's clock. The FIFO buffers multiple incoming events from various traces. It has a fixed size, which can be adjusted, and reports any FIFO overruns. The Contract Fetch module attempts to match incoming trace events with one or more monitors. If a match is found, it enriches the event with monitor values and sends it to the Contract Check module. The Contract Check module cyclically checks each active monitor. Incoming events may initiate a new monitor, reset an existing one, or trigger a violation of one or more monitors. Any violations are then reported by the Report Interface.

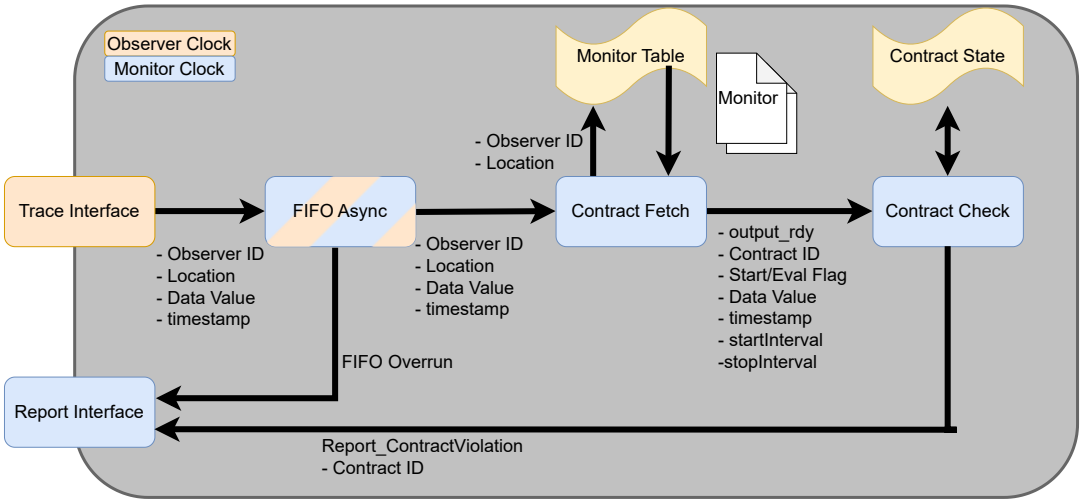


Figure 3.51: Refined Architecture of Time Contract Co-Processor (TCCP).

#### 3.2.4.5 Interfaces

As illustrated in Figure 3.50, the TCCP features three interfaces: a Configuration Interface (Config Interface), a Trace Interface and a Report Interface.

The Config Interface is used to transmit the monitoring information, derived from requirements and specifications, to the co-processor. This communication is planned to occur via an AXI Bus.

The Trace Interface connects the TCCP to the targeted modules through multiple Observers. Each Observer directly connects to gather information about a specific property, such as the current program counter or the power usage of a component.

The Report Interface collects information about violated contracts or monitors and provides details about the violated contract, including its possible priority. It can also optionally trigger an interrupt.

#### 3.2.4.6 Evaluation prototype

The evaluation concept is divided into three different stages and two different scenarios.

In the initial stage, SystemC is used to test the system using benchmarks derived from specific scenarios, simulating the system's behavior under various conditions to ensure it meets the required specifications and behaves as intended.

The second stage focuses on developing a custom testbench for the SystemVerilog simulation environment with Verilator. This testbench is designed to verify the functionality of the system within a minimal simulation environment.

The last stage involves testing two distinct scenarios to evaluate the system's capabilities in real-world applications. The first scenario tests the capabilities of the TCCP with the Trace Ingress Port Interface on a CVA6 architecture, developed by SYSGO and UNIBO. The second scenario tests the capabilities within the automotive demo use case of Task T5.2. Here, the TCCP is integrated into a NoelV architecture and coupled with a custom interface to an IR LED supervisor and with either a custom interface or AXI to the AI-ML Accelerator developed by FotoNation.

# 4 Conclusion

This deliverable provided refined architecture definitions and implementation details for the safety and security related hardware modules and extensions that are developed within WP3 (Accelerators and Extensions) of the ISOLDE project. The deliverable structure follows the WP3 tasks' structure as introduced in deliverable D3.1 that was presenting the early architecture of the WP3 hardware modules, pointing out technical progress since previous year deliverable.

Each specific safety / security hardware module described in this deliverable is presented together with:

- an **IP card** summarizing technical, licensing and availability of the hardware module.
- Both **general context information** as well as a presentation of the **purpose and scope** of the hardware extension.
- A **refined architecture** as well as both the software and the hardware interfaces of each module.
- Potential toolchain and ISA-specific **customizations**.
- The **evaluation prototype**, reporting the implementation development and early results prior to the integration infto the WP5 use-cases.

Based on this document, WP5 use-cases will select which safety / security related hardware extensions will be integrated, together with technologies from WP2 (*Foundation Cores*) and WP4 (*Software Tools*), paving the way to RISC-V-based safe & secure high-performance embedded computing systems.

With regards to WP3, this delivarble is an intermetiate report that will be finalized as part of Deliebrable D3.4 (*Final implementation for Safety and Security Module*) in month M33.

# 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 | Double Data Rate Synchronous Dynamic Random Access Memory |
| DES | Data Encryption Standard |
| DFT | Discrete Fourier Transform |
| DFU | Decoder Functional Units |

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

| | |
|---|---|
| MC | Memory Controller |
| MCCU | Maximum Contention Control Unit |
| MDPC | Moderate-Density Parity-Check |
| ML | Machine Learning |
| ML-DSA | Module-Lattice-based – Digital Signature Standard |
| ML-KEM | Module-Lattice-based – Key Encapsulation Mechanism |
| MMIO | Memory Mapped Input/Output |
| MMU | Memory Management Unit |
| MPSoC | Multiprocessor System on a Chip |
| MSW | Most Significant Word |
| NIST | National Institute of Standards and Technology |
| NoC | Network on Chip |
| NR | New Radio |
| NTT | Number Theoretic Transform |
| ONNX | Open Neural Network eXchange |
| OVI | Open Vector Interface |
| PC | Program Counter |
| PCA | Parallel Computing Accelerator |
| PE | Processing Engine |
| PMP | Physical Memory Protection |
| PMU | Performance Monitor Unit |
| POR | Power-On Reset |
| PPA | Power, Performance, and Area |
| PQC | Post-Quantum Cryptography |
| PQC-MA | Post-Quantum Crypto Accelerator |
| PRF | Polymorphic Register File |
| PRINCE | Low-latency block cipher |
| PRNG | Pseudorandom Number Generator |
| QC | Quasi-Cyclic |
| QUARMAv2 | Lightweight tweakable block cipher |
| RDC | Request Duration Counter |
| ReO | Rectangle Only |
| ReRo | Rectangle Row |
| ReTr | Rectangle Transposed |
| RF | Radio Frequency |
| RFOG | Register File Organization Table |
| RoCo | Row Column |
| ROM | Read-Only-Memory |
| ROP | Return Oriented Programming |
| RoT | Root-of-Trust |
| RSA | Rivest–Shamir–Adleman |
| RTL | Register Transfer Level |
| RTPM | Run-Time Power Monitoring instrumentation |
| RV32 | 32-bit RISC-V processor model |

| | |
|---|---|
| RVV | RISC-V Vector extension |
| S | Supervisor Mode |
| SafeSU | Safety-related Statistics Unit |
| SafeTI | Safety-related Traffic Injector |
| SCA | Shared Correlation Accelerator |
| SCH | SCHeduler |
| SCMI | System Control and Management Interface |
| SDK | Software Development Kit |
| SDRAM | Synchronous Dynamic Random Access Memory |
| SEC | SECured RISC-V processor with cryptographic accelerators |
| SHA | Secure Hash Algorithms |
| SIMD | Single Instruction Multiple Data |
| SLDU | SLiDe Unit |
| SLH-DSA | Stateless Hash-Based Digital Signature Standard |
| SM | Security Monitor |
| SNN | Spiking Neutral Networks |
| SoA | State of the Art |
| SoC | System on a Chip |
| SPI | Serial Peripheral Interface |
| SRAM | Static Random-Access Memory |
| SSCU | Safety and Security Control Unit |
| 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 |
| XCF | XtratuM Configuration Files |
| XIF | eXtension InterFace |
| XNG | XtratuM Next Generation |

# Bibliography

[1] arm. Whitepaper: Armv8.5-a memory tagging extension. online, 2024.

[2] Subhadeep Banik, Takanori Isobe, Fukang Liu, Kazuhiko Minematsu, and Kosei Sakamoto. Orthros: A low-latency PRF. Cryptology ePrint Archive, Paper 2021/390, 2021.

[3] Elaine Barker and John Kelsey. Recommendation for random number generation using deterministic random bit generators, June 2015.

[4] Christof Beierle, Jérémy Jean, Stefan Kölbl, Gregor Leander, Amir Moradi, Thomas Peyrin, Yu Sasaki, Pascal Sasdrich, and Siang Meng Sim. The SKINNY family of block ciphers and its low-latency variant MANTIS. Cryptology ePrint Archive, Paper 2016/660, 2016.

[5] Dušan Božilov, Maria Eichlseder, Miroslav Kneževic, Baptiste Lambin, Gregor Leander, Thorben Moos, Ventzislav Nikov, Shahram Rasoolzadeh, Yosuke Todo, and Friedrich Wiemer. PRINCEv2 - more security for (almost) no overhead. Cryptology ePrint Archive, Paper 2020/1269, 2020.

[6] Robert Buhren, Hans-Niklas Jacob, Thilo Krachenfels, and Jean-Pierre 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*, CCS '21, page 2875–2889, New York, NY, USA, 2021. Association for Computing Machinery.

[7] Stephen Cass. Top Programming Languages 2022. online, August 2022.

[8] Luca Cremona, William Fornaciari, and Davide Zoni. Automatic identification and hardware implementation of a resource-constrained power model for embedded systems. *Sustainable Computing: Informatics and Systems*, 29:100467, 2021.

[9] Morris Dworkin. Recommendation for block cipher modes of operation: Galois/counter mode (gcm) and gmac, November 2007.

[10] Elektronik- und Mikrosysteme VDI/VDE Innovation + Technik GmbH. VE-VIDES Project. https://www.elektronikforschung.de/projekte/ve-vides, 2024. Accessed: 2025-03-31.

[11] César Fuguet. HPDcache: Open-Source High-Performance L1 Data Cache for RISC-V Cores. In *Proceedings of the 20th ACM International Conference on Computing Frontiers*, pages 377–378, Bologna, Italy, 2023. ACM.

[12] Nicolas Gerlin, Endri Kaja, Fabian Vargas, Li Lu, Anselm Breitenreiter, Junchao Chen, Markus Ulbricht, Maribel Gomez, Ares Tahiraga, Sebastian Prebeck, Eyck Jentzsch, Miloš Krstić, and Wolfgang Ecker. Bits, flips and riscs. In *2023 26th International Symposium on Design and Diagnostics of Electronic Circuits and Systems (DDECS)*, pages 140–149, 2023.

[13] Sylvain Girbal, Jimmy Le Rhun, and Hadi Saoud. METrICS: a Measurement Environment for Multi-Core Time Critical Systems. In *Embedded Real Time Software and Systems*, ERTS '18, 2018.

[14] Sylvain Girbal, Jimmy Le Rhun, Daniel Gracia Pérez, and David Faura. Safety & Security monitoring convergence at the dawn of Open Hardware and Artificial Intelligence, June 2022.

[15] Google Project Zero. 0day "In the Wild" 2024. online, January 2025.

[16] Yuko Hara, Hiroyuki Tomiyama, Shinya Honda, Hiroaki Takada, and Katsuya Ishii. Chstone: A benchmark program suite for practical c-based high-level synthesis. In *2008 IEEE International Symposium on Circuits and Systems (ISCAS)*, pages 1192–1195, 2008.

[17] John L Hennessy and David A Patterson. Computer organization and design RISC-V edition: The hardware software interface, 2017.

[18] Endri Kaja, Nicolas Gerlin, Monideep Bora, Gabriel Rutsch, Keerthikumara Devarajegowda, Dominik Stoffel, Wolfgang Kunz, and Wolfgang Ecker. Fast and accurate model-driven fpga-based system-level fault emulation. In *2022 IFIP/IEEE 30th International Conference on Very Large Scale Integration (VLSI-SoC)*, pages 1–6, 2022.

[19] Yoongu Kim, Ross Daly, Jeremie Kim, Chris Fallin, Ji Hye Lee, Donghyuk Lee, Chris Wilkerson, Konrad Lai, and Onur Mutlu. Flipping bits in memory without accessing them: an experimental study of dram disturbance errors. *SIGARCH Comput. Archit. News*, 42(3):361–372, jun 2014.

[20] Hans Liljestrand, Thomas Nyman, Lachlan J Gunn, Jan-Erik Ekberg, and N Asokan. {PACStack}: an authenticated call stack. In *30th USENIX Security Symposium (USENIX Security 21)*, pages 357–374, 2021.

[21] Pascal Nasahl, Robert Schilling, Mario Werner, Jan Hoogerbrugge, Marcel Medwed, and Stefan 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*, pages 200–212, 2021.

[22] Shoei Nashimoto, Daisuke Suzuki, Rei Ueno, and Naofumi Homma. Bypassing isolated execution on risc-v using side-channel-assisted fault-injection and its countermeasure. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, 2022(1):28–68, Nov. 2021.

[23] RISC-V International. RISC-V Shadow Stacks and Landing Pads. online, July 2024.

[24] RISC-V Shadow-stack and Landing-pads Task Group. Shadow stack and landing pads (0036ff2). Technical report, RISC-V International, 2024. Commit: 0036ff20a7305f6705ad630c63a9aa58da7df285.

[25] Cyril Roscian, Alexandre Sarafianos, Jean-Max Dutertre, and Assia Tria. Fault model analysis of laser-induced faults in sram memory cells. In *2013 Workshop on Fault Diagnosis and Tolerance in Cryptography*, pages 89–98, 2013.

[26] Alberto Sangiovanni-Vincentelli, Werner Damm, and Roberto Passerone. Taming Dr. Frankenstein: Contract-Based Design for Cyber-Physical Systems. *European Journal of Control*, 18(3):217–238, 2012.

[27] Duc Do Tran, Kim Grüttner, Frank Oppenheimer, and Wolfgang 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)*, volume 1, pages 156–163, 2020.

[28] Andrew Waterman, Yunsup Lee, David A. Patterson, and Krste Asanovic. The RISC-V Instruction Set Manual, December 2019.

[29] F. Zaruba and L. Benini. The Cost of Application-Class Processing: Energy and Performance Analysis of a Linux-Ready 1.7-GHz 64-Bit RISC-V Core in 22-nm FDSOI Technology. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 27(11):2629–2640, Nov 2019.

[30] Davide Zoni, Luca Cremona, and William Fornaciari. Design of side-channel-resistant power monitors. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 41(5):1249–1263, 2022.

[31] Davide Zoni, Andrea Galimberti, and William Fornaciari. A survey on run-time power monitors at the edge. *ACM Comput. Surv.*, 55(14s), July 2023.