UNIVERSITY OF PIRAEUS - DEPARTMENT OF INFORMATICS

MSc «Cybersecurity and Data Science»

MSc Thesis

Thesis Title:	**Fault Injection and Lockstep Evaluation on Zynq Ultrascale+ MPSoC** Τεχνητή Εισαγωγή Σφαλμάτων και Αξιολόγηση Λειτουργίας του Zynq Ultrascale+ MPSoC
Student's name-surname:	**Nikolaos Alampasis**
Father's name:	**Athanasios**
Student's ID No:	**ΜΠΚΕΔ21001**
Supervisor:	**Michalis Psarakis, Professor**

2025-12

3-Member Examination Committee

Michalis Psarakis	**Panagiotis Kotzanikolaou**	**Dimitrios Agiakatsikas**
Professor	**Professor**	**Postdoc Researcher**

Acknowledgments

I would like to express my sincere gratitude to my professors, Mihalis Psarakis and Dimitrios Agiakatsikas, for their continuous guidance, encouragement, and trust throughout the course of my postgraduate studies. Their unwavering support and belief in my abilities, even during moments when I struggled to believe in myself, played a decisive role in the completion of this work. The presence of such academics is invaluable within the university environment. Their contribution to both research and teaching significantly enhances the quality, relevance, and advancement of higher education, and I am deeply thankful for the opportunity to have worked under their supervision.

I would also like to warmly thank my family, my mother and my brother, for standing by my side during every stage of my postgraduate journey. Their constant support, patience, and understanding provided me with stability and motivation, enabling me to remain focused and persistent during demanding periods of study and research.

In addition, I would like to acknowledge my close friends and colleagues, whose encouragement and companionship were a constant source of strength and reassurance. Their willingness to listen, discuss ideas, and offer support—both academic and personal—made the challenges of this process more manageable and meaningful.

Finally, I would like to extend my appreciation to the academic and professional environment that supported this work, including all those who contributed indirectly through discussions, feedback, shared knowledge, and access to resources. This thesis represents not only an individual effort, but also the result of collective support, collaboration, and inspiration received throughout my studies.

Summary

This dissertation presents an experimental investigation of fault behavior and fault-handling mechanisms in the AMD Zynq UltraScale+ MPSoC, using the Ultra96-V2 platform as a representative hardware target. The focus is placed on soft-error relevant events and the architectural features that support dependable operation in safety-oriented embedded systems, with particular emphasis on the dual-core Arm Cortex-R5 Real-Time Processing Unit (RPU) operating in lockstep mode, the Error Correcting Code (ECC) protection of on-chip memory (OCM), and the system-level coordination performed by the Platform Management Unit (PMU).

The work follows a hands-on methodology in which controlled fault conditions are introduced and their propagation is observed through firmware logs, runtime application behavior, and PMU-to-RPU signaling paths. A FreeRTOS-based RPU application was developed to provide deterministic workload execution and continuous liveness monitoring through a heartbeat task. In parallel, PMU firmware functionality was customized to detect memory-related ECC events and explicitly notify the RPU through Inter-Processor Interrupt (IPI) messages. This enabled a practical end-to-end validation of a fault notification and acknowledgement mechanism, where the PMU reports an event, the RPU handles the interrupt, and an acknowledgement is returned to the PMU to confirm correct reception.

Three experimental test cases are presented. First, a correctable ECC event is injected into OCM and shown to be handled transparently: the hardware ECC logic corrects the fault, the PMU reports the event and issues an IPI notification, and the FreeRTOS application continues normal execution without interruption. Second, an uncorrectable ECC event is injected and results in escalation behavior: the PMU captures additional error metadata, sends the event through IPI to the RPU, and the software confirms receipt while maintaining controlled execution, demonstrating observability and diagnostic reporting under higher-severity memory faults. Third, a lockstep mismatch is triggered through the RPU error-injection mechanism, demonstrating a fundamentally different class of fault: the lockstep comparator detects divergent execution between redundant cores and the PMU enforces a fail-safe recovery action by resetting the RPU and releasing it back to a known state. The observed behavior confirms that, unlike ECC-related memory faults, lockstep execution mismatches are treated as unrecoverable at the software level and require architectural reset semantics.

Overall, the dissertation contributes an experimentally grounded characterization of fault detection, reporting, and recovery paths on real MPSoC hardware. The results highlight the practical distinction between (i) memory-level faults that can be corrected or reported while execution continues and (ii) processor-level lockstep mismatches that demand immediate architectural recovery. Future work is identified in extending the methodology to additional memory regions (for example, TCM and DDR), enriching the software response beyond acknowledgement (for example, checkpointing or controlled degradation), and expanding the fault injection campaign to systematically evaluate coverage and recovery latency across operating conditions.

Περίληψη

Η παρούσα διατριβή παρουσιάζει μια πειραματική διερεύνηση της συμπεριφοράς σφαλμάτων και των μηχανισμών ανίχνευσης και χειρισμού τους στο AMD Zynq UltraScale+ MPSoC, με χρήση της πλατφόρμας Ultra96-V2 ως αντιπροσωπευτικού στόχου υλικού. Η μελέτη εστιάζει σε συμβάντα σχετιζόμενα με soft errors και στα αρχιτεκτονικά χαρακτηριστικά που υποστηρίζουν αξιόπιστη λειτουργία σε συστήματα ενσωματωμένης ασφάλειας, με ιδιαίτερη έμφαση στη μονάδα πραγματικού χρόνου Arm Cortex-R5 (RPU) σε λειτουργία lockstep, στην προστασία μνήμης On-Chip Memory (OCM) μέσω Error Correcting Code (ECC) και στον συντονισμό σε επίπεδο συστήματος από την Platform Management Unit (PMU).

Ακολουθείται πρακτική, πειραματική μεθοδολογία, κατά την οποία εισάγονται ελεγχόμενες συνθήκες σφάλματος και παρατηρείται η διάδοσή τους μέσω καταγραφών firmware, της συμπεριφοράς εφαρμογών κατά την εκτέλεση και της σηματοδότησης μεταξύ PMU και RPU. Αναπτύχθηκε εφαρμογή για την RPU βασισμένη σε FreeRTOS, παρέχοντας ντετερμινιστική εκτέλεση και συνεχή επιτήρηση λειτουργικότητας μέσω μηχανισμού heartbeat. Παράλληλα, τροποποιήθηκε το PMU firmware ώστε να ανιχνεύει συμβάντα ECC και να ειδοποιεί ρητά την RPU μέσω Inter-Processor Interrupts (IPIs), επιτρέποντας την άκρο-σε-άκρο επικύρωση ενός μηχανισμού ειδοποίησης και επιβεβαίωσης λήψης.

Παρουσιάζονται τρεις πειραματικές δοκιμές. Αρχικά, διορθώσιμο ECC σφάλμα στην OCM αντιμετωπίζεται διαφανώς, με τη συνέχιση της κανονικής εκτέλεσης. Στη συνέχεια, μη διορθώσιμο ECC σφάλμα οδηγεί σε κλιμακωμένη αναφορά και διαγνωστική παρατηρησιμότητα, διατηρώντας ελεγχόμενη λειτουργία. Τέλος, ασυμφωνία lockstep προκαλείται μέσω μηχανισμού εισαγωγής σφάλματος της RPU, οδηγώντας σε άμεση ανίχνευση από τον lockstep comparator και επιβολή ενέργειας ανάκτησης fail-safe μέσω επαναφοράς της RPU από την PMU.

Συνολικά, η διατριβή προσφέρει πειραματικά τεκμηριωμένη αποτύπωση των διαδρομών ανίχνευσης, αναφοράς και ανάκτησης σφαλμάτων σε πραγματικό υλικό MPSoC, αναδεικνύοντας τη διάκριση μεταξύ διορθώσιμων σφαλμάτων μνήμης και μη ανακτήσιμων σφαλμάτων εκτέλεσης σε lockstep. Ως μελλοντική εργασία προτείνεται η επέκταση της μεθοδολογίας σε επιπλέον περιοχές μνήμης και η ενίσχυση της απόκρισης του λογισμικού πέρα από την απλή επιβεβαίωση λήψης.

Contents

List of Figures

List of Tables

1. Introduction

1.1 Purpose of Work

This dissertation focuses on fault injection and fault behavior analysis in the Cortex-R5 RPU operating in lockstep mode. Lockstep execution provides hardware-level redundancy by duplicating processor execution and comparing results cycle by cycle, enabling rapid detection of faults affecting the processor core [52, 6]. This approach is commonly employed in safety-critical systems, where early fault detection is essential to prevent error propagation.

The Cortex-A53 APU is intentionally excluded from the primary scope of this work. While suitable for high-performance application processing, it does not provide an equivalent hardware lockstep mechanism, and fault detection at that level typically relies on software-based redundancy or operating-system-level error handling. Similarly, fault tolerance techniques implemented in the programmable logic, such as triple modular redundancy, are outside the scope of this study, as they require additional hardware design and introduce further architectural complexity.

By concentrating on the Cortex-R5 lockstep mechanism, this work evaluates the effectiveness of vendor-provided, in-silicon fault tolerance features without additional redundancy. The experiments conducted are exploratory and empirical in nature, aiming to observe how injected faults manifest at runtime, whether they are detected by lockstep comparison or ECC mechanisms, and how the system responds once faults are detected. The purpose is not to achieve formal certification, but to deepen understanding of fault behavior and error handling in a modern heterogeneous MPSoC through direct experimentation.

1.2 Motivation

Modern embedded systems are increasingly deployed in safety-critical domains such as automotive, aerospace, industrial automation, and robotics. In these applications, hardware or software faults may lead not only to service disruption but also to physical damage or risks to human safety. As a result, system dependability, encompassing reliability, availability, and fault tolerance, has become a primary design concern [15].

1.2.1 Dependability Challenges in Modern Embedded Systems

With continued technology scaling, semiconductor devices have become more susceptible to transient faults, commonly referred to as soft errors. These faults are often caused by radiation-induced phenomena, such as single-event upsets, which may corrupt memory cells or internal processor state without leaving permanent damage [15]. Although such events are rare, their impact can be severe in safety-critical systems if they remain undetected. Consequently, modern embedded platforms integrate hardware-based fault detection and mitigation mechanisms in order to reduce the probability of silent data corruption and uncontrolled system behavior.

1.2.2 Limitations of Specification-Level and Analytical Validation

While contemporary System-on-Chip platforms incorporate sophisticated fault tolerance mechanisms, their effectiveness is often validated primarily through analytical models, simulations, or compliance with design specifications. However, analytical reliability models inevitably rely on simplifying assumptions and may not capture complex interactions between hardware components, firmware, and application software [24]. In

addition, simulation-based approaches typically abstract away low-level timing effects and hardware-specific behavior that can influence fault manifestation and detection.

Radiation testing has been employed as an alternative method for evaluating fault resilience, but such experiments are costly, difficult to reproduce, and often require extreme conditions to provoke rare failure modes. Experimental studies on the Zynq UltraScale+ MPSoC under neutron irradiation, for example, have demonstrated very low observed failure rates due to the presence of ECC and other mitigation mechanisms [5]. While these results indicate strong baseline robustness, they also highlight the difficulty of exercising critical fault scenarios using natural fault sources alone.

As a consequence, relying exclusively on specification-level validation or accelerated testing may provide limited insight into how faults propagate through a real system and how fault-handling mechanisms behave under precisely controlled conditions. This gap motivates the use of deliberate fault injection as a complementary validation technique.

1.2.3 Motivation for Experimental Fault Injection on MPSoC Platforms

Fault injection is a well-established experimental methodology for evaluating the dependability of computing systems by intentionally introducing faults and observing the resulting behavior [24]. By injecting faults at specific locations and times, it becomes possible to assess fault detection coverage, response latency, and the effectiveness of recovery mechanisms in a repeatable manner. This approach is particularly valuable for complex MPSoC platforms, where multiple subsystems interact and fault effects may propagate across processor, memory, and management domains.

The AMD Xilinx Zynq UltraScale+ MPSoC integrates several hardware-level fault tolerance mechanisms, including lockstep execution in the Cortex-R5 Real-Time Processing Unit and ECC protection for on-chip and cache memories [52]. These mechanisms are designed to detect transient and permanent faults at runtime and to signal error conditions to higher-level system components. However, publicly available experimental studies that characterize the behavior of these mechanisms on real hardware remain limited, especially from an end-user perspective.

This lack of empirical evidence is particularly relevant for the Cortex-R5 lockstep mechanism. While vendor documentation describes its operation and intended fault coverage, the actual system-level response to injected faults—including how errors are propagated, signaled, and handled by firmware—cannot be fully understood without experimental investigation [6]. Similarly, although ECC mechanisms are widely assumed to provide reliable protection against memory faults, their behavior under controlled fault scenarios, such as uncorrectable multi-bit errors, must be validated in the context of the full system [52].

1.2.4 Motivation for Focusing on Cortex-R5 and System-Level Fault Handling

This dissertation focuses on the Cortex-R5 RPU and its associated fault-handling infrastructure, rather than on the application-class Cortex-A53 cores. The Cortex-R5 subsystem is specifically intended for deterministic and safety-oriented workloads and is therefore the natural candidate for hosting critical functionality in safety-related applications [19]. Lockstep execution provides hardware-level fault detection with minimal software overhead, making it a cornerstone of fault tolerance in such systems.

In addition to processor-level fault detection, the Zynq UltraScale+ MPSoC employs a centralized error management architecture involving the Platform Management Unit. Fault conditions detected within the processing system can be routed to the PMU, which executes dedicated firmware responsible for coordinating system-level responses, such as resets or controlled shutdowns [55]. Inter-processor interrupts are used as a signaling mechanism between the RPU and PMU, enabling timely notification and handling of critical fault events.

The motivation of this work is therefore to experimentally investigate how these mechanisms operate together under fault conditions. By combining controlled fault injection with observation of lockstep behavior, ECC responses, and PMU-mediated signaling, this dissertation aims to provide empirical insight

into the dependability of a modern safety-oriented MPSoC. This experimental perspective complements existing analytical and specification-based approaches and forms a necessary foundation for defining realistic research objectives and scope in the following section.

1.3 Research Objectives and Scope

This section defines the objectives and scope of the dissertation by mapping the experimental work performed on the Ultra96-V2 platform to specific research goals while explicitly stating the limits within which these goals are examined. In doing so, it attempts to provide the reader a clear understanding of the purpose and contributions of the work.

1.3.1 Research Objectives

The primary objective of this dissertation is to experimentally investigate the behavior of fault tolerance mechanisms integrated into a modern heterogeneous System-on-Chip platform, specifically the Zynq UltraScale+ MPSoC, under controlled fault injection scenarios. The emphasis is placed on mechanisms intended to support real-time and safety-oriented operation, rather than on general-purpose application processing.

More specifically, the objectives of this work are:

- To experimentally evaluate the behavior of the Arm Cortex-R5 Real-Time Processing Unit operating in lockstep mode when subjected to injected faults, and to observe how execution divergence is detected at the hardware level .
- To investigate the response of Error Correcting Code (ECC) mechanisms in on-chip memory (OCM) under controlled fault injection, with particular attention to the distinction between correctable and uncorrectable fault scenarios and their observable effects .
- To analyze how detected fault conditions propagate beyond the processor and memory subsystems, focusing on system-level fault signaling and handling mediated by the Platform Management Unit through inter-processor interrupts .
- To observe the interaction between hardware fault detection mechanisms and software execution contexts on the Cortex-R5, including bare-metal and FreeRTOS-based applications, in order to understand how fault events become visible at the software level.

These objectives are investigative rather than prescriptive. The intent is not to modify or redesign the fault tolerance mechanisms provided by the platform, but to characterize their observed behavior, limitations, and interactions through direct experimentation on real hardware.

1.3.2 Focus and Delimitation of the Study

This dissertation deliberately focuses on the Cortex-R5 processing subsystem of the Zynq UltraScale+ MPSoC and its associated fault tolerance features. The Cortex-R5 RPU is specifically designed for deterministic execution and high dependability, and it is therefore the natural candidate for hosting safety-critical functionality in MPSoC-based systems [19]. As such, the experimental investigation concentrates on the RPU operating in lockstep mode, the ECC-protected on-chip memory resources used by the processing system, and the system-level mechanisms responsible for fault signaling and management.

The study emphasizes fault injection at the processor and on-chip memory level, with the goal of observing how built-in detection mechanisms respond to injected faults and how fault information is propagated through the system. Particular attention is given to the role of the Platform Management Unit, which coordinates system-level responses to detected fault conditions, and to the use of inter-processor interrupts as a signaling mechanism between the RPU and PMU [55].

As a consequence of this focused investigation, several aspects of the Zynq UltraScale+ MPSoC are intentionally not addressed. Fault tolerance mechanisms associated with the Cortex-A53 Application Processing Unit are not examined, as the APU does not provide hardware lockstep execution and typically relies on software-based redundancy or operating-system-level fault handling. Similarly, fault mitigation techniques implemented in the programmable logic, such as triple modular redundancy or custom hardware protection schemes, are beyond the scope of this work, as they require additional hardware design and introduce architectural considerations distinct from the built-in processing system features.

Furthermore, this dissertation does not aim to perform formal functional safety certification or to demonstrate compliance with standards. Performance benchmarking and long-term reliability modeling are also outside the scope of the study. The focus remains on empirical observation and characterization of fault behavior under controlled experimental conditions, rather than on quantitative lifetime prediction or certification-oriented analysis.

2. Overview of Fault Tolerance in Embedded Systems

Each new technology generation brings challenges to increasing transistor density. As transistor sizes shrink, lithography techniques must advance to meet demands. Meanwhile, DRAM performance has lagged, creating a "memory wall" that led to faster memory and architectural solutions like prefetching. Recently, high power dissipation became critical, termed the "power wall," prompting manufacturers to innovate for efficiency. Now, transient faults from particles, known as the "soft error wall," are the next issue for chip reliability. Radiation-induced transient faults arise from energetic particles, such as alpha particles from packaging material and neutrons from the atmosphere, generating electron–hole pairs (directly or indirectly) as they pass through a semiconductor device. Transistor source and diffusion nodes can collect these charges. A sufficient amount of accumulated charge may invert the state of a logic device, such as a latch, static random access memory (SRAM) cell, or gate, thereby introducing a logical fault into the circuit's operation. Because this type of fault does not reflect a permanent malfunction of the device, it is termed *soft* or *transient*.[34]

In embedded systems, fault tolerance is crucial for maintaining reliable operation. Mostly in applications where system failure can have severe consequences, such as aerospace, automotive, and medical devices. One of the key challenges in fault tolerance is managing soft errors—transient errors caused by external factors like radiation, which can alter the state of a system temporarily without causing permanent damage. Unlike hard faults that require component replacement, soft errors typically arise from environmental influences such as cosmic rays or electromagnetic interference, and they can cause unpredictable bit-flips in memory or processors. Because of their transient nature, soft errors pose unique challenges for embedded systems, demanding robust fault-tolerant mechanisms to ensure systems continue functioning correctly and safely.

2.1 Reliability in Safety-Critical Systems

The nature of embedded systems in safety-critical applications demands high reliability, where even a minor fault can compromise functionality and safety. As it is vigorously asserted in literature, systems such as aircraft controllers or nuclear plant systems must prioritize fault tolerance due to the life-threatening risks associated with failure. Soft errors, specifically those induced by radiation, represent a major threat to these systems. Radiation exposure, even in low doses, can cause single-event upsets (SEUs)—a type of soft error that flips a bit in memory or logic circuits, potentially altering the system's behavior without warning. Consequently, fault-tolerant systems in such environments must employ techniques like error detection and correction (EDC) and redundancy to identify and address these transient errors before they escalate into critical faults.

Soft errors are transient faults that pose a particular challenge in embedded systems due to their random and temporary nature. To mitigate these, redundancy-based techniques, such as duplication of control units and power supplies, are commonly employed, as seen in duplicated Central Processing Unit(CPU) and Power Supply Unit(PSU) architectures (Figures 2.1 and 2.2). These architectures ensure that if a fault affects one unit, another can seamlessly take over, maintaining system reliability. Triple Modular Redundancy (TMR) is another method that protects against soft errors by using three parallel processors, comparing their outputs, and relying on majority voting to identify errors. This approach effectively masks soft errors by isolating and correcting transient faults, allowing the system to continue functioning as intended.

Effective fault tolerance relies not only on detecting soft errors but also on quickly recovering from them. Watchdog timers and checkpointing are essential tools in fault recovery, allowing systems to revert to known good states if a fault is detected. For instance, a watchdog timer can monitor the system for irregular behavior and trigger a reset or recovery process if an error is detected. Checkpointing allows the system to save states periodically, so in the event of a soft error, it can rollback to the last correct state. This combination of detection and recovery ensures that even if soft errors momentarily disrupt the system, the impact remains

minimal, and full functionality can be quickly restored.

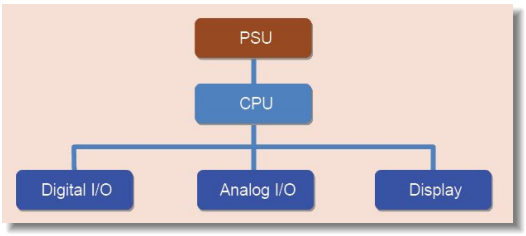

Figure 2.1: Single-PSU embedded system architecture.

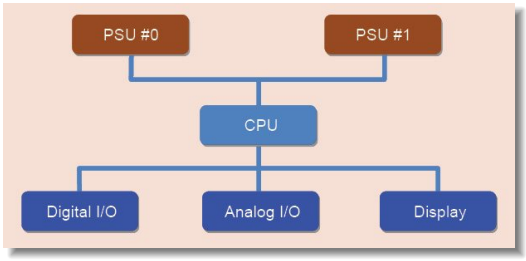

Figure 2.2: Redundant-PSU embedded system architecture

Safety-critical embedded systems are commonly developed in accordance with international functional safety and assurance standards, such as IEC 61508, ISO 26262, and the aerospace standards DO-254 and DO-178C [25, 26, 21, 20]. These standards establish structured approaches for identifying hazards, classifying safety integrity levels, and defining requirements for fault detection, error containment, and safe system behavior throughout the system lifecycle.

IEC 61508 provides a generic framework for functional safety applicable across multiple industrial domains, emphasizing systematic fault management, quantitative safety targets, and architectural measures for fault tolerance [25]. ISO 26262 adapts these principles to automotive electronic systems by introducing Automotive Safety Integrity Levels (ASILs), which guide the selection of architectural mitigation mechanisms and fault handling strategies in complex electronic control systems [26].

In the aerospace domain, DO-254 and DO-178C define design assurance objectives for airborne electronic hardware and software, respectively. These standards place strong emphasis on deterministic behavior, rigorous verification, and controlled system response to faults, particularly in environments where fault consequences may be catastrophic [21, 20].

While this dissertation does not aim to demonstrate compliance with any specific certification standard, the fault tolerance mechanisms examined are directly motivated by the principles established within these frameworks. In particular, the emphasis on early fault detection, controlled recovery, and prevention of hazardous behavior aligns with the objectives of safety-critical system design as defined by these standards.

2.2 Errors in Safety Critical Systems

Errors in computing systems arise from various factors, which include manufacturing defects, material degradation, and external influences. Manufacturing imperfections, such as contaminants or physical defects in silicon, can lead to early failures or malfunctions, often seen as ”infant mortality” in device reliability studies[32]. As components are used over time, they also experience material wearout, such as electromigration and oxide breakdown, which degrade performance and lead to permanent faults. Additionally, ongoing thermal and mechanical stresses increase these wear effects, especially as device sizes shrink and densities increase, which can accelerate the progression of faults. At a higher level, software bugs and design flaws in logic contribute to errors, as these logical faults introduce unpredictable behaviors in system performance. However, among all sources, radiation-induced errors, primarily from alpha particles and cosmic rays, pose a unique challenge. These high-energy particles interact with semiconductor materials, causing random bit flips and transient faults that can compromise both memory and logic circuits. Radiation-induced errors, also known as soft errors, have become increasingly significant as devices continue to miniaturize, making systems more susceptible to these external, unpredictable influences[33]

2.2.1 Taxonomy of Faults, Errors, and Silent Data Corruptions

Errors and faults can be categorized based on their source, nature, and manifestation in electronic components. This section defines the types of errors—single-event upsets (SEUs), single-event transients (SETs), and single-event latch-ups (SELs)—and explains the environmental and material factors that cause them.

i. Faults: Origins and Types

Faults are the foundational disruptions in microelectronic systems, originating from physical interactions or material impurities that cause deviations in expected behavior. These faults can be classified as follows:

- **Radiation-Induced Faults:** High-energy particles, such as cosmic rays, alpha particles from packaging materials, and atmospheric neutrons, can strike semiconductor materials, creating ionized particles that disrupt circuit behavior. This disruption is especially problematic in high-altitude or space environments, where reduced atmospheric shielding allows more radiation to reach electronic components [28],[23],[27] .

- **Material and Packaging-Induced Faults:** Impurities in packaging materials, including uranium and lead isotopes, emit alpha particles that can trigger faults when they strike sensitive circuit elements. For instance, the lead isotopes in solder used in electronic packaging emit alpha particles during decay, which can cause faults if these particles impact memory cells or critical logic components [46] .

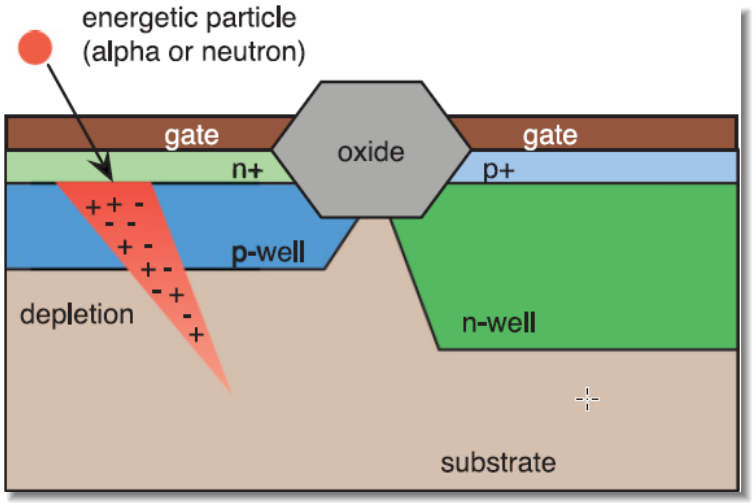

Figure 2.3: Alpha-particle emission from packaging materials impacting sensitive circuit regions, illustrating a packaging-induced radiation fault [28].

ii. Errors: Manifestations of Faults

When faults impact data storage or processing within the system, they manifest as errors, which can be transient or persistent. Errors are categorized by their behavior and impact: » Explain critical charge more

1. **Single Event Effects (SEEs):** SEEs are errors caused by particle strikes that induce temporary or lasting changes in digital elements. The primary types of SEEs include:

- **Single-Event Upset (SEU):** Are bit flips in memory cells or registers caused by a particle strike exceeding the critical charge *(Q_{crit})* required to change the cell state. SEUs result in data corruption without physically damaging the hardware. They are a significant source of silent data corruption (SDC) in systems where detection mechanisms are not equipped to catch transient bit flips [28],[16] .
- **Single-Event Transient (SET):** Are a temporary voltage or current pulses generated by a particle strike. If this pulse is wide enough, it may propagate through logic gates and potentially affect downstream flip-flops, resulting in transient logic errors. SETs become increasingly critical in scaled-down devices, where the reduced Q_{crit} and shorter propagation times make SETs more likely to influence system functionality [36],[23],[27] .
- **Single-Event Latch-up (SEL):** Are persistent errors triggered by particle strikes that activate parasitic structures in the silicon substrate, causing a high-current state. This condition can lead to thermal runaway and permanent damage if not addressed immediately. SELs are particularly hazardous in high-power applications [27],[46] .
- **Single-event function interrupts (SEFIs):** are disruptions to normal device operation (beyond a simple corruption of user data). These types of effects alter the functionality of the circuit and typically require reconfiguration/reset orpower cycling for recovery [46] .

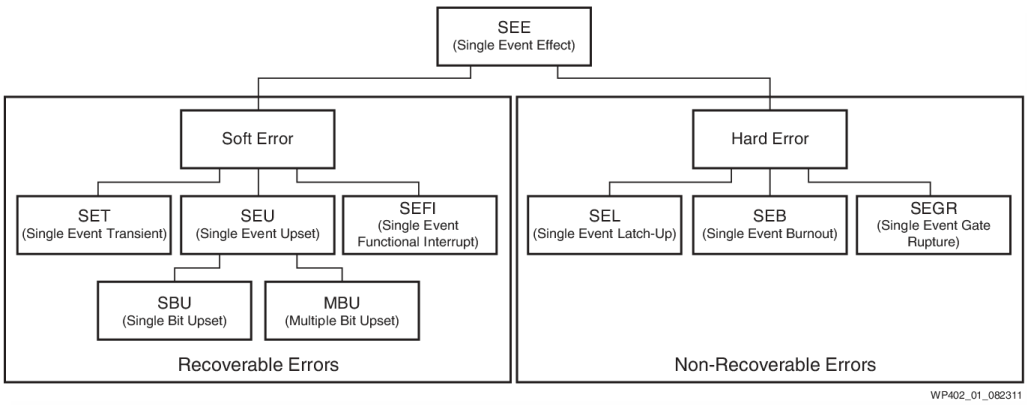

Figure 2.4: Types of Single Event Effects [46].

2. **Soft Errors:** These are transient, non-destructive errors caused by environmental factors, such as radiation. Soft errors affect data without damaging the physical circuitry, meaning the erroneous data can often be corrected by simply rewriting it. However, smaller technology nodes increase the likelihood of soft errors, as the critical charge Q_{crit} needed for an upset decreases with scaling, making devices more vulnerable to these errors [28],[23] .

iii. Silent Data Corruptions (SDCs): Undetected Errors

Silent Data Corruptions (SDCs) represent a specific class of errors where faults go undetected, allowing erroneous data to propagate unnoticed within the system. SDCs are particularly critical in systems where undetected faults can compromise functionality or safety.

- **Connection between SEEs, SEUs, and SDCs:** SEUs are a primary contributor to SDCs in microelectronic systems. When SEUs alter the state of memory cells or registers without detection, they silently propagate through the system, leading to incorrect final outcomes. For example, a single-bit SEU in a control register might silently alter the computational flow of a safety-critical process, resulting in SDCs that are only observable in the incorrect output.
- **Impact of SETs on SDCs:** SETs, though transient, can also cause SDCs when they alter critical logic states undetected. This is particularly relevant in high-speed circuits and scaled-down de-

vices where brief transient faults may propagate before stabilization or detection occurs, causing subtle but impactful disruptions.

2.3 Foundations of Dependability

The original definition of dependability is the ability to deliver service that can justifiably be trusted. This definition stresses the need for justification of trust [11]. In computing, the prevalence of errors and faults—ranging from temporary disturbances like soft errors to more enduring hardware failures—highlights dependability as essential to system design. It ensures availability, reliability, safety, and security especially as systems are increasingly used in critical sectors where minor errors can have serious consequences.

Dependability covers a range of impairments, including hazards, defects, faults, errors, malfunctions, and failures. While these terms overlap, each represents distinct impacts on system performance. Despite challenges in consistent terminology, significant progress has been made in standardizing dependability across industries, with organizations like the IEC, ISO, and ECSS establishing unified frameworks and best practices. There are five principal dimensions to dependability, as shown in the figure below:

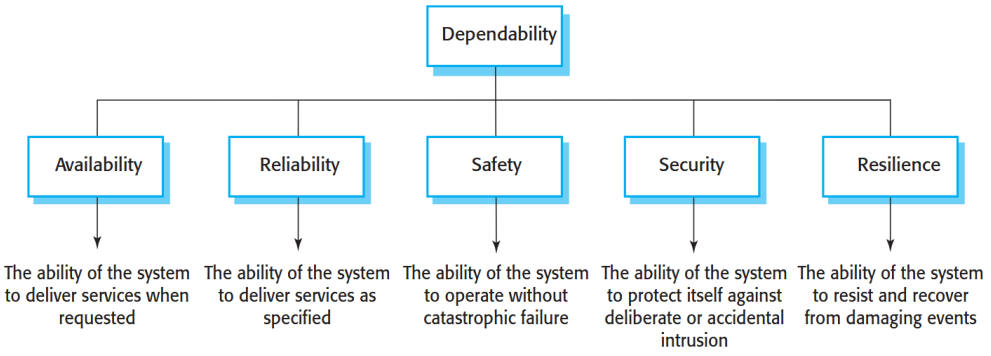

Figure 2.5: Principal Dependability Properties [40].

2.3.1 Dependability Attributes

The attributes of dependability define the essential properties a system should possess. Avizienis *et al.*(2004) outlines five key dependability attributes: (1) *availability* —the readiness for correct service, (2) *reliability* —the system's capacity to deliver correct service continuously, (3) *safety* —the absence of catastrophic impacts on users and the external environment, (4) *integrity* —the prevention of improper system alterations and (5) *maintainability* —the system's ability to undergo repairs and modifications. While all the above attributes collectively contribute to system dependability, more recent approached ahve enriched the term of depentability. The table below expands on these attributes and includes additional aspects, providing both qualitative descriptions and quantitative measures for each attribute:

2.3.2 Reliability

Reliability is defined as the probability that a system performs its intended function correctly for a specified period of time under stated operating conditions. In safety-critical and embedded systems, reliability is commonly associated with uninterrupted correct execution during a mission interval.

Term	Qualitative Usage(s)	Quantitative Measure(s)
Availability	Highly available High-availability Continuously available	(Pointwise) Availability Interval availability MTBF, MTTR
Integrity	High-integrity Tamper-proof Fool-proof	
Maintainability	Easily maintainable Maintenance-free Self-repairing	
Performability		Performability MCBF
Reliability	Reliable Highly reliable High-reliability Ultrareliable	Reliability MTTF or MTFF
Resilience	Resilient	
Robustness	Robust	Impairment tolerance count
Safety	High-safety Fail-safe	Risk
Security	Highly secure High-security Fail-secure	
Serviceability	Easily serviceable	
Testability	Easily testable Self-testing Self-checking	Controllability Observability

Table 2.1: Principal Dependability Properties [37].

Reliability is typically expressed as a function of time:

$$R(t) = P(T > t) \tag{2.1}$$

where T denotes the time to failure. Under the common assumption of a constant failure rate λ, reliability can be modeled as:

$$R(t) = e^{-\lambda t} \tag{2.2}$$

Although this exponential model is widely used due to its analytical simplicity, it relies on assumptions that may not hold for complex MPSoC platforms. Heterogeneous architectures, layered fault mitigation mechanisms, and interactions between hardware and software can result in non-uniform fault behavior over time. Consequently, analytical reliability models may fail to capture the full range of failure manifestations observed in practice, motivating empirical evaluation through experimental fault injection.

2.3.3 Availability

Availability represents the probability that a system is operational and able to deliver its intended service at a given point in time. Unlike reliability, which focuses on failure-free operation over a continuous interval, availability explicitly incorporates both failure occurrence and recovery.

Availability is commonly quantified as:

$$A = \frac{\text{MTTF}}{\text{MTTF} + \text{MTTR}} \tag{2.3}$$

where MTTF is the Mean Time To Failure and MTTR is the Mean Time To Repair or recovery. This formulation highlights that availability depends not only on how frequently faults occur, but also on how effectively the system detects, isolates, and recovers from them.

In embedded systems, recovery does not necessarily imply full system restart. Instead, availability may be preserved through localized recovery mechanisms such as interrupt-driven fault handling, task restart, or memory correction. As a result, MTTR becomes strongly dependent on architectural design and fault management strategies.

For MPSoC platforms, availability is therefore closely tied to hardware-assisted error detection and supervisory mechanisms that enable rapid fault signaling and controlled recovery. Evaluating availability in practice requires observing service interruption and restoration behavior under fault conditions, which cannot be fully derived from analytical models alone.

2.3.4 Performability

Performability extends traditional dependability attributes by jointly considering system performance and fault tolerance. Rather than treating system operation as binary (operational or failed), performability captures the impact of faults on performance degradation and degraded modes of operation.

A commonly used formulation expresses performability as an expected reward:

$$E[P] = \sum_i p_i \cdot r_i \tag{2.4}$$

where p_i represents the probability of the system being in state i, and r_i denotes the performance level associated with that state. States may include normal operation, degraded operation, recovery phases, or fault-handling modes.

In real-time and embedded systems, performance degradation may manifest as increased execution latency, reduced throughput, or altered scheduling behavior following fault detection and recovery actions. Consequently, performability is influenced not only by fault rates, but also by how recovery mechanisms interact with runtime execution.

In MPSoC-based systems executing real-time software, fault handling may temporarily suspend tasks, trigger service interruption, or introduce scheduling delays. These effects directly impact performance even when functional correctness is preserved, making experimental evaluation essential for capturing realistic performance–dependability trade-offs.

2.3.5 Integrity and Safety

Integrity and safety address the correctness and acceptability of system behavior in the presence of faults. Integrity refers to the system's ability to prevent unintended modification or corruption of data and state, ensuring that the information remains accurate and consistent throughout operation. Safety concerns the avoidance of hazardous behavior of the system that may harm humans, equipment, or the environment.

Integrity can be expressed probabilistically as the likelihood that an internal error does not propagate

into an externally observable incorrect state:

$$P_{\text{integrity}} = 1 - P(\text{error propagation}) \tag{2.5}$$

Safety is often characterized by the probability that a failure leads to a hazardous condition:

$$P_{\text{unsafe}} = P(\text{failure} \rightarrow \text{hazard}) \tag{2.6}$$

Safety-oriented systems are typically designed to favor fail-safe or fail-silent behavior over continued operation with incorrect results. Architectural mitigation mechanisms such as redundancy, error detection, and containment aim to reduce the likelihood that faults compromise system integrity or lead to unsafe behavior.

In practice, evaluating integrity and safety requires observing whether faults are successfully detected and contained before they affect system outputs or control flow. Experimental fault injection provides insight into the effectiveness of architectural mitigation mechanisms under realistic fault conditions.

2.4 Summary of Findings and Gaps

This chapter has presented the foundational concepts of fault tolerance and dependability in embedded systems, including the classification of faults, errors, and failures, as well as the principal dependability attributes of reliability, availability, performability, integrity, and safety. These concepts define how systems are expected to behave under fault conditions and how mitigation mechanisms aim to preserve correct and acceptable service.

Despite the maturity of dependability theory, much of the existing analysis relies on abstract fault models and simplifying assumptions that do not fully capture the behavior of modern MPSoC platforms. In practice, real fault manifestations may differ significantly from modeled assumptions due to architectural complexity, heterogeneous processing elements, and tightly coupled hardware–software interactions. As a result, architectural fault mitigation mechanisms may exhibit unexpected behavior when exercised under realistic fault conditions.

Furthermore, dependability attributes ultimately manifest through observable system-level phenomena, such as interrupts, error status indications, recovery delays, or performance degradation. Understanding how faults propagate to these observable signals is essential for evaluating fault tolerance mechanisms in practice, yet such behavior cannot be fully inferred from specification-level or analytical analysis alone.

These limitations motivate the use of controlled experimental fault injection as a means of studying real system behavior. By deliberately introducing faults and observing their effects on execution, recovery, and performance, it becomes possible to assess how architectural mechanisms operate in practice. The following chapters build upon this foundation by examining the architecture of the target MPSoC platform, defining an experimental fault injection methodology, and analyzing observed system behavior under injected fault conditions.

3. Architecture and Technical Specifications

This chapter presents the architectural and technical foundations of the system under study, serving as the bridge between the dependability theory introduced in Chapter 2 and the experimental methodology discussed in Chapter 4. The chapter presents an overview of the Ultra96-V2 platform and the Zynq UltraScale+ MPSoC presenting their cababilities and aims to detail the architectural mechanisms that enable fault detection and error signaling, and observability.

A top-down description of the Ultra96-V2 development board is presented, highlighting the Processing System (PS), the integrated ZU3EG device, memory components (including OCM, TCM, and LPDDR4), and external interfacing features. Section 3.2 zooms in on the Zynq UltraScale+ MPSoC architecture, detailing its processing clusters, interconnects, memory controllers, and safety features. Section 3.3 then focuses specifically on the Cortex-R5 subsystem and its lockstep execution mode, discussing how redundant execution supports error detection. In Section 3.4, we examine ECC-based memory protection across various memory tiers and how it interacts with lockstep mechanisms. Finally, Section 3.5 presents system-level error handling through the PMU, interrupt architecture, and error propagation infrastructure.

Each section is designed to establish the architectural contract of the platform and its inherent fault response features, which will be later exercised and evaluated through controlled fault injection experiments.

[51]

3.1 Overview of AMD Zynq UltraScale+ MPSoC

The Ultra96-V2 is a compact development board based on the Xilinx Zynq UltraScale+ MPSoC ZU3EG, integrating a heterogeneous multiprocessor system-on-chip into a small-form-factor 96Boards-compliant platform. Designed by Avnet, the board is intended for edge-computing, machine learning, and embedded control applications. Its architecture, interfaces, and observability features also make it a strong candidate for experimental fault injection studies in a laboratory context [13].

3.1.1 Board-Level Architecture

At the core of the Ultra96-V2 is the Zynq UltraScale+ ZU3EG device in the SBVA484 package. This MPSoC combines a quad-core ARM Cortex-A53 APU, dual-core Cortex-R5 RPU (real-time processing), Mali-400 GPU, and Programmable Logic (PL). The board supports 2 GB of LPDDR4 memory connected via a 32-bit wide interface to the DDR controller.

All critical components are interconnected through a high-bandwidth AXI interconnect structure provided by the Zynq platform itself. The external components supporting the SoC include:

- **Memory:** 2 GB Micron LPDDR4 (32-bit), 16 GB Industrial MLC microSD
- **Connectivity:** USB 3.0 upstream, USB 3.0/2.0 downstream, Mini DisplayPort, Wi-Fi/Bluetooth module
- **I/O Expansion:** 40-pin low-speed and 60-pin high-speed 96Boards headers
- **Power:** Supplied via 12 V barrel jack; integrated power management from Infineon/ON Semiconductor
- **Debug:** JTAG/UART pod interface through Micro-USB, compatible with Xilinx tools

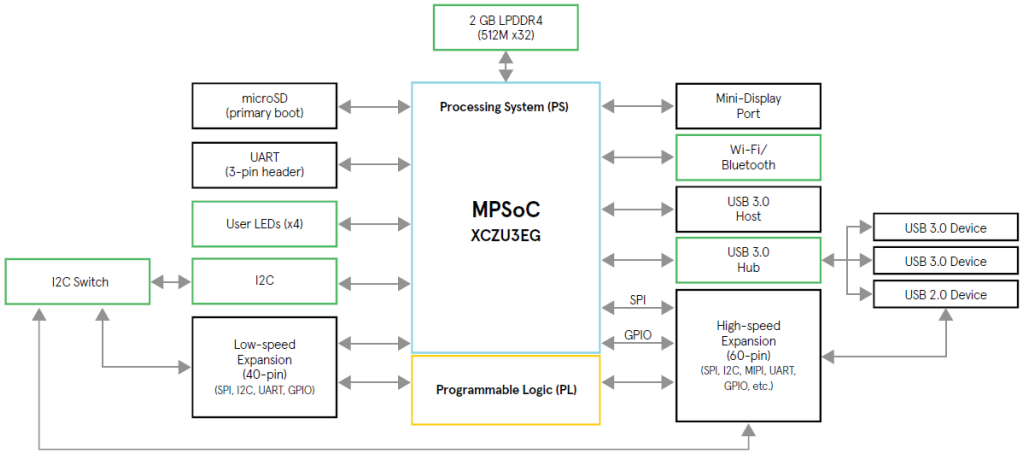

Figure 3.1: Ultra96-V2 Board Block Diagram showing integration of the ZU3EG MPSoC and memory components [13].

3.1.2 System Boot and Configuration

The Ultra96-V2 is typically booted from microSD via the onboard SDIO interface. SW3 provides a boot mode selector that allows choosing between SD, QSPI, or JTAG modes. A dedicated USB-to-JTAG/UART pod enables early-stage software observability. The board's boot flow leverages the platform management features of the ZU3EG's PMU and ROM-based BootROM [13].

3.1.3 PS Connectivity and IO

The Processing System (PS) provides 78 multiplexed I/O lines (MIO), which are grouped into three banks (Banks 500–502). These I/Os are fixed at 1.8 V and support multiple peripheral standards including UART, SPI, I2C, and GPIO. The USB/UART pod connects to PS MIO0/1 and enables runtime communication with the processor. Additional I/O is exposed via EMIO to the PL and further routed through the expansion headers.

3.1.4 Experimental Relevance

The Ultra96-V2 provides an excellent foundation for experimental fault injection thanks to its support for ECC-protected memory, Cortex-R5 lockstep execution, and multiple observability points across the system. These features enable the detection and tracking of faults as they propagate through the hardware. Although the board has some constraints—such as the lack of native Ethernet connectivity and a relatively narrow DRAM interface—its structured processor-memory architecture and accessible debugging interfaces allow for precise control and repeatable fault behavior analysis. These characteristics make it particularly suitable for the kind of low-level experimentation and fault response observation explored in this work.

3.2 Zynq UltraScale+ MPSoC

The Zynq UltraScale+ MPSoC represents a heterogeneous multiprocessor architecture that integrates a powerful processing system (PS) with programmable logic (PL) on a single die. The architecture is designed to provide flexibility, performance scalability, and advanced capabilities for safety- and security-critical applications.

3.2.1 High-Level Architecture

The Processing System (PS) of the Zynq UltraScale+ includes multiple functional units: a quad-core ARM Cortex-A53 Application Processing Unit (APU), a dual-core ARM Cortex-R5 Real-Time Processing Unit (RPU), a Mali-400MP2 Graphics Processing Unit (GPU), and a Platform Management Unit (PMU). These cores are supported by an extensive interconnect, a system memory architecture, and dedicated accelerators.

Figure 3.3 illustrates the high-level layout of the Zynq UltraScale+ MPSoC processing system.

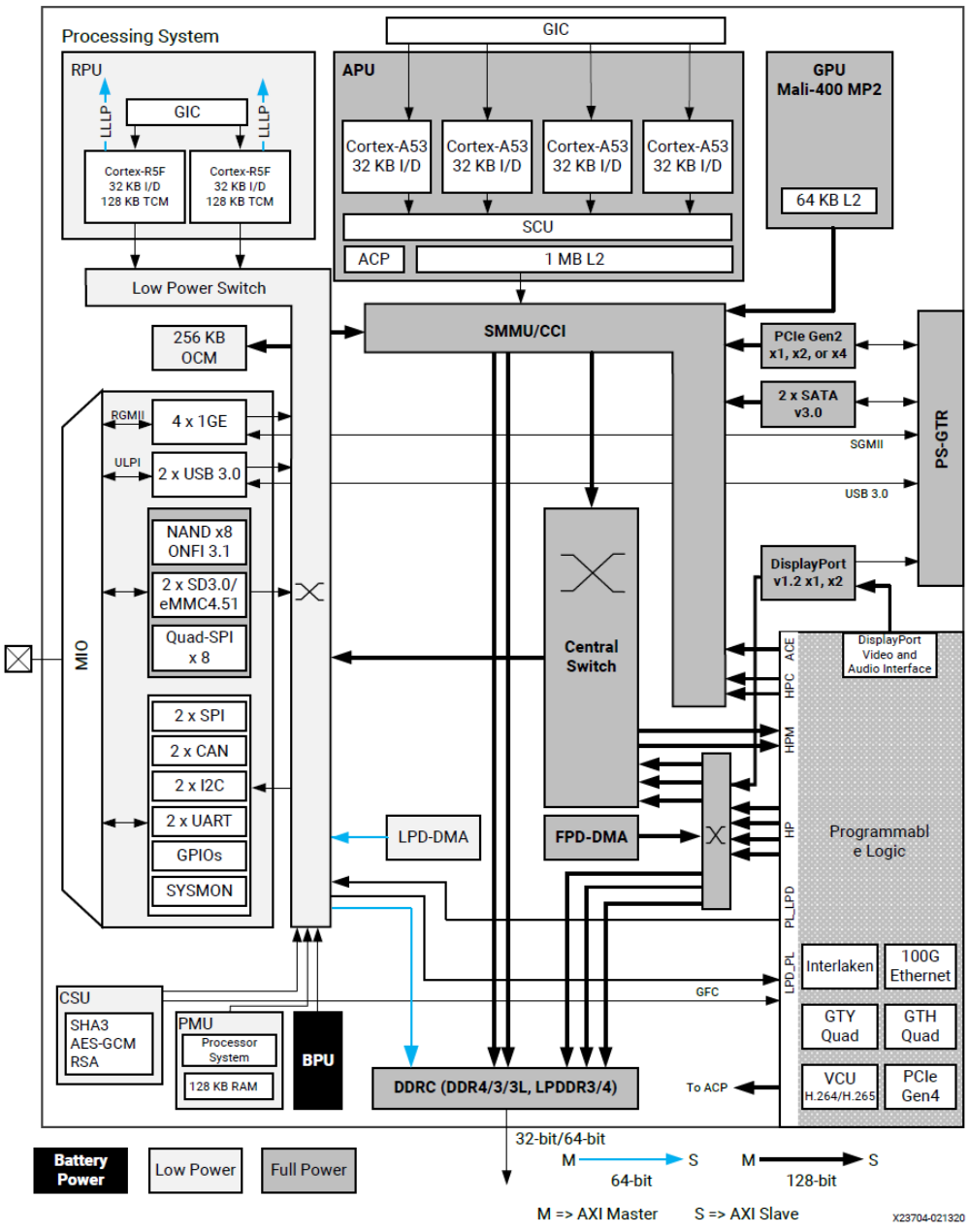

Figure 3.2: Zynq UltraScale+ MPSoC Processing System architecture [4].

3.2.2 Application Processing Unit (APU)

The APU comprises four 64-bit ARM Cortex-A53 cores that support symmetric multiprocessing (SMP). These cores operate in a coherent cluster using an ARM v8-A architecture and are ideal for general-purpose operating systems, such as Linux. The APU can operate independently from the RPU and includes its own interrupt controller (GIC-500) and L2 cache controller [4].

3.2.3 Real-Time Processing Unit (RPU)

The RPU consists of two ARM Cortex-R5F cores that can be configured either in split mode or lockstep mode for increased reliability. It operates using ARM v7-R architecture and includes tightly-coupled memories (TCM) and ECC-enabled caches. The RPU is intended for deterministic real-time applications and is frequently used in safety-critical designs [4].

3.2.4 Platform Management Unit (PMU)

The PMU is responsible for critical functions such as system monitoring, power management, safety supervision, and firmware-driven reset handling. It plays a central role in system-level fault management, working closely with other PS blocks to coordinate safe state transitions, clock gating, and voltage domain control. The PMU includes its own MicroBlaze processor and RAM for executing platform firmware [4].

3.2.5 Memory Architecture

The PS is equipped with a rich memory subsystem:

- **DDR Memory Controller:** Supports DDR4/LPDDR4/DDR3/DDR3L memories via a high-bandwidth AXI interface.
- **On-Chip Memory (OCM):** A 256 KB memory region accessible by all processor units, with support for ECC-based protection.
- **Tightly-Coupled Memory (TCM):** Associated with the RPU cores, though discussed in detail in Section 3.3.
- **Cache Hierarchy:** APU includes private L1 and shared L2 caches; RPU and GPU have their own memory hierarchies.

All address spaces are mapped through a unified 40-bit virtual addressing scheme, and the interconnect supports quality-of-service (QoS) enforcement for memory transactions.

3.2.6 Interconnect and Peripherals

The PS integrates a high-performance AXI interconnect fabric that enables communication among the cores and between the PS and PL. Key interconnect features include:

- Fully coherent interfaces between APU cores and L2 cache
- AXI High-Performance Master and Slave ports (HPM/HP) bridging PS and PL
- Low-Power Domain (LPD) and Full-Power Domain (FPD) separation

Peripheral support includes multiple USB 3.0, SD, CAN, I2C, SPI, UART, and Gigabit Ethernet interfaces [4]. These peripherals are routed through the Multiplexed I/O (MIO) and Extended MIO (EMIO) paths.

3.2.7 Security and Isolation Features

The architecture includes hardware-level security features such as the Configuration Security Unit (CSU), cryptographic engines (AES, RSA, SHA-3), TrustZone for partitioned execution, and System Memory Management Units (SMMUs) for virtualization and memory protection.

3.2.8 Relevance to Fault Injection Studies

For fault tolerance research, Zynq UltraScale+ MPSoC provides a rich substrate of observability and control features. These include:

- Dedicated error signaling to the PMU
- Programmable reset and power control granularity
- Integration of ECC-enabled memories
- Fine-grained interrupt routing and status registers

These architectural elements allow targeted fault injection, system-wide monitoring, and deterministic isolation of fault propagation effects.

3.3 The Cortex-R5

The Real-Time Processing Unit (RPU) in the Zynq UltraScale+ MPSoC is a dual-core subsystem comprising two ARM Cortex-R5F processors designed for deterministic, low-latency tasks. Unlike the APU, the RPU operates within the Low-Power Domain (LPD), allowing power-efficient and reliable execution of safety-critical workloads.The RPU in the Zynq UltraScale+ MPSoC includes two ARM Cortex-R5F cores implementing the Armv7-R architecture, optimized for safety-critical and low-latency applications. These cores can operate independently in split mode or together in lockstep mode for fault-tolerant operation.

Each Cortex-R5F core includes the following key architectural features [3, Chapter 4, p. 68]:

- **Instruction and Data Caches:** 32 KB instruction cache and 32 KB data cache per core, both with ECC protection to enhance reliability. These caches accelerate access to external memory while maintaining data integrity.
- **Tightly Coupled Memories (TCMs):** Two TCM interfaces per core: ATCM for instructions and BTCM for data. Each TCM has 64 KB capacity and 64-bit data width. BTCM is divided into two 32 KB banks (BTCM0 and BTCM1), allowing interleaved accesses. In lockstep mode, TCMs from both cores are aggregated to provide 128 KB ATCM and 128 KB BTCM to the active core.
- **Floating-Point Unit (FPU):** Each processor includes a single and double-precision FPU compliant with the ARM VFPv3 standard, supporting hardware-accelerated math operations.
- **AXI Interfaces:** Each core connects to system memory and peripherals via a 64-bit AXI3 master interface. Additionally, a 32-bit peripheral AXI interface provides direct access to low-latency devices like the Generic Interrupt Controller (GIC).
- **Interrupt Handling:** Fast interrupt response is enabled by a dedicated peripheral port to the GIC and support for non-maskable fast interrupts. Each core also features an interrupt injection mechanism for internal testing, capable of injecting to 160 SPIs via APB.
- **Memory Protection Unit (MPU):** A region-based MPU is integrated into each core, supporting up to 16 configurable regions. It enforces access control, caching behavior, and privilege levels for memory regions.

- **Built-in Self-Test (BIST):** Each core supports BIST for diagnosing random hardware faults in logic and memory.
- **ECC Mechanisms:** ECC is applied across the L1 caches and TCMs using a single-error correction, double-error detection (SEC-DED) scheme. This strengthens the system's resilience against soft errors.
- **Performance Monitoring:** Event counters and a performance monitoring unit enable tracking of processor-level events like cache misses, useful for profiling real-time performance.
- **Power Management:** The cores support multiple power states—Run, Standby, and Shutdown—allowing dynamic trade-offs between performance and energy consumption. Each mode offers a balance of clock gating, logic retention, and TCM memory retention.

All these architectural features are designed to support fast, predictable execution with hardware mechanisms for safety, making the RPU well-suited for real-time and mission-critical applications in the context of fault injection and mitigation.

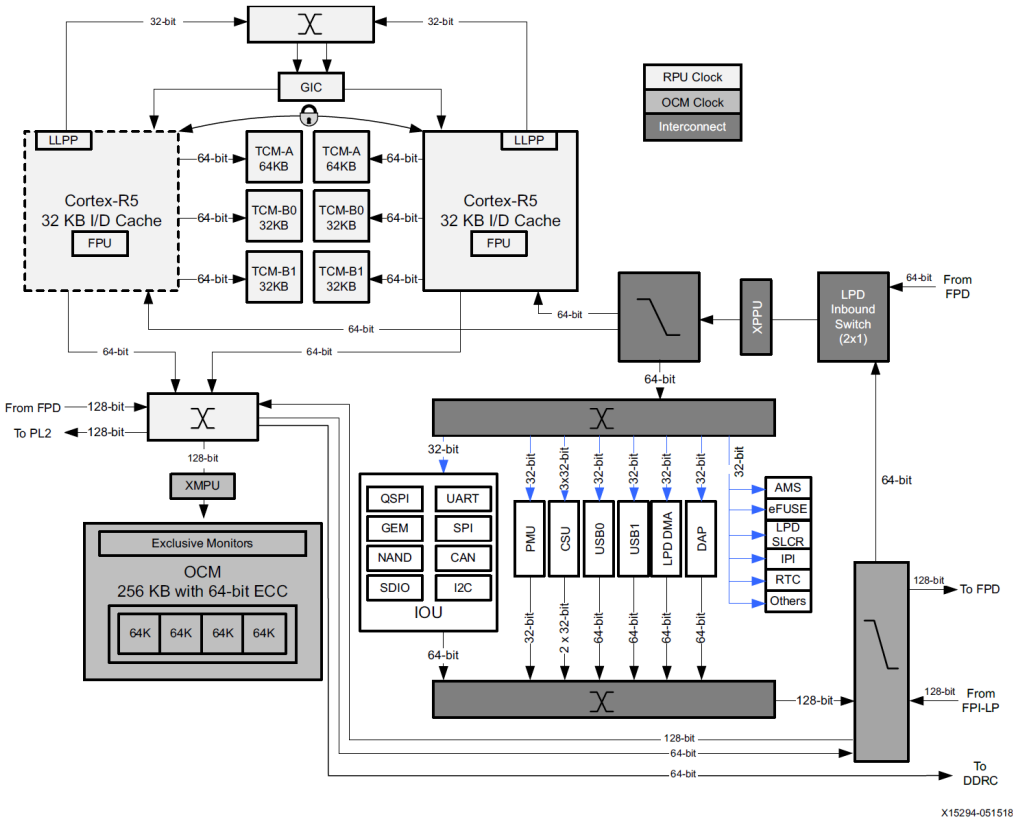

Figure 3.3: Zynq UltraScale+ MPSoC Processing System architecture [3, Chapter 4, p. 70].

3.3.1 Defining Real-Time Processing

Each Cortex-R5F processor in the RPU uses separate paths for instructions and data. This means that fetching instructions and reading or writing data can happen at the same time, avoiding bottlenecks and improving consistency. To support fast and predictable execution, each core also includes 32 KB of instruction cache and 32 KB of data cache, along with two local memory blocks called TCMs (Tightly Coupled Memories).

TCM-A is a 64 KB instruction memory, and TCM-B is divided into two 32 KB data banks (B0 and B1) [3, Chapter 4].

What makes real-time systems different from regular computers is that they're not only expected to produce the right result they also have to produce it on time. Timing is just as important as correctness. In many critical applications, a late response can be just as bad as a wrong one. For example, an anti-lock braking system in a car must react within milliseconds to prevent a wheel from locking. If it's even slightly late, the safety mechanism fails. That's what we call a "hard" real-time system. Other types, like video streaming or air conditioning control, are more 'forgiving', meaning that they might tolerate occasional delays, though too many can still hurt performance or comfort.

Because of this focus on timing, real-time processors like the RPU are built differently from general-purpose processors. General CPUs are designed for high overall throughput and often use large caches and out-of-order execution to go faster on average. But this comes at the cost of unpredictability, which isn't acceptable in systems that need guaranteed response times. The RPU, on the other hand, uses simpler and more controlled memory structures, like TCMs, to make sure that access times are always predictable — which is exactly what real-time systems need.

Real-time systems are typically classified into three categories based on their timing constraints [18]:

- **Hard Real-Time:** Missing a deadline constitutes total failure. Applications like pacemakers and aircraft flight control systems demand strict timing guarantees.
- **Firm Real-Time:** Occasional missed deadlines are tolerable, but delayed results are discarded. An example is a digital video decoder.
- **Soft Real-Time:** Timeliness affects quality of service but not correctness. For instance, air-conditioning feedback systems benefit from timely updates but remain functional despite delays.

3.3.2 Memory Architecture and Determinism

The deterministic execution of the RPU is strongly supported by its multi-tiered memory architecture. This includes Level 1 memory consisting of Tightly Coupled Memories (TCMs) and caches, and Level 2 memory comprising On-Chip Memory (OCM) and external DDR memory. The layout of these components is shown in Figure 3.4.

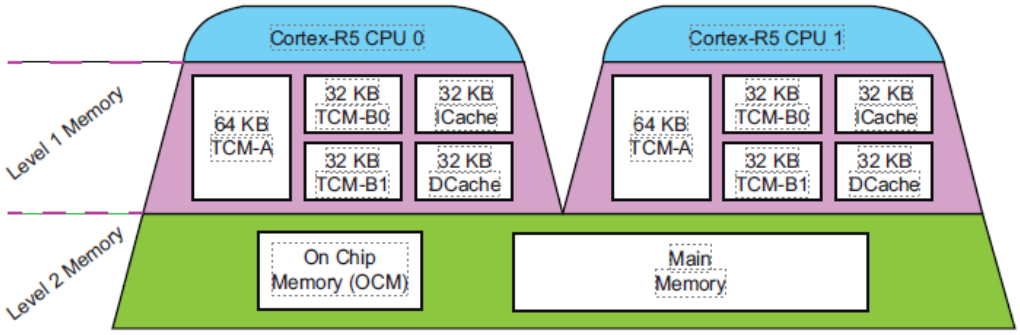

Figure 3.4: Memory hierarchy of the RPU subsystem showing per-core TCMs, caches, and access paths to shared OCM and main DDR memory [18, Chapter 7, Figure 7.2].

3.3.3 Tightly Coupled Memories (TCMs)

The Tightly Coupled Memories (TCMs) are a key part of what gives the Cortex-R5 cores their predictable and low-latency behavior. These are small blocks of memory directly attached to each processor core. Each

Cortex-R5 has two banks: TCM-A and TCM-B, both 64 KB wide. TCM-B is further split into two 32 KB interleaved segments (B0 and B1), which improves access efficiency when the processor is reading or writing data in rapid bursts[3, 18].

Unlike normal memory or caches, TCMs offer consistent, fixed access time. This makes them perfect for storing real time and critical data, code instructions like interrupt handlers or high priority control loops. TCMs eliminate the uncertainty associated with cache misses by offering predictable, low-latency memory access, which is crucial for meeting strict real-time deadlines. For instance, TCM-A is often used to hold short, high-priority code such as exception routines, while TCM-B is ideal for time sensitive data blocks, such as streaming input from sensors or intermediate processing results.

The behavior of TCMs also changes depending on the operating mode of the RPU. In **split mode**, each Cortex-R5 core operates independently with its own private set of 128 KB (64 KB ATCM + 64 KB BTCM). In **lockstep mode**, the two cores execute the same code in parallel, and their TCMs are combined. This means that the single ”active” core (typically R5_0) gets access to the full 256 KB of TCM (128 KB ATCM + 128 KB BTCM). This configuration provides a wider and more capable memory interface, improving system throughput and fault resilience which translates to higher reliability.

	R5_0 View (Start Address)	R5_1 View (Start Address)	Global Address View (Start Address)
Split mode			
R5_0 ATCM (64 KB)	0x0000_0000	N/A	0xFFE0_0000
R5_0 BTCM (64 KB)	0x0002_0000	N/A	0xFFE2_0000
R5_0 instruction cache (32 KB)	I-Cache	N/A	0xFFE4_0000
R5_0 data cache (32 KB)	D-Cache	N/A	0xFFE5_0000
R5_1 ATCM (64KB)	N/A	0x0000_0000	0xFFE9_0000
R5_1 BTCM (64KB)	N/A	0x0002_0000	0xFFEB_0000
R5_1 instruction cache (32 KB)	N/A	I-Cache	0xFFEC_0000
R5_1 data cache (32 KB)	N/A	D-Cache	0xFFED_0000
Lock-step mode			
R5_0 ATCM (128KB)	0x0000_0000	N/A	0xFFE0_0000
R5_0 BTCM (128KB)	0x0002_0000	N/A	0xFFE2_0000
R5_0 instruction cache (32 KB)	I-Cache	N/A	0xFFE4_0000
R5_0 data cache (32 KB)	D-Cache	N/A	0xFFE5_0000
R5_1 slave port is not accessible in lock-step mode.			

Figure 3.5: TCM Address Map [3].

This difference in configuration and memory mapping is illustrated in Figure 3.5, which shows how TCMs are exposed in the system’s memory space under split and lockstep modes. Regardless of the mode, TCMs remain tightly integrated into the pipeline, supporting deterministic and parallel access through dedicated ports.

Access to TCMs is tightly managed inside the processor. When multiple internal or external parts in or out of the core such as the load/store unit (LSU), instruction prefetch unit (PFU) or AXI slave interfaces , simultaneously request access to the TCMs, the Cortex-R5F processor handles these requests using a fixed, hardware-defined priority. Load/store operations are always served first, followed by instruction fetches and finally any external access through the AXI interface is served. This predictable access order ensures that time-critical code and data are delivered without delay, which is essential for maintaining the timing required in real-time and safety-critical applications [18].

From a software perspective, TCMs are exposed at fixed memory regions. Application code and data can be statically placed into those address spaces using linker scripts during compilation. Firmware routines typically ensure that the memory controller and address decode logic route access correctly, depending on

whether the system is in split or lockstep mode.

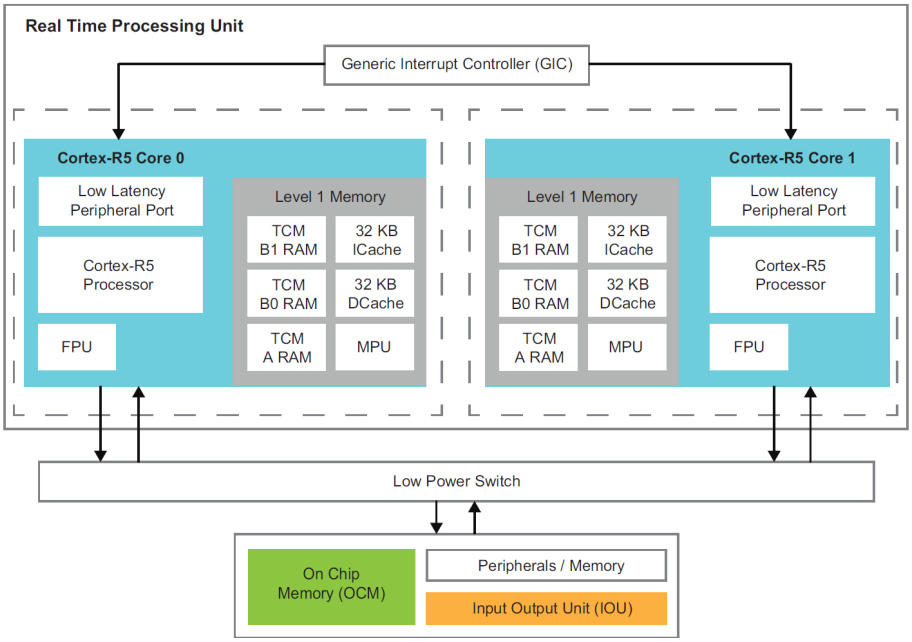

Figure 3.6: RPU Memory Topology [18, Chapter 7, p. 164]

The overall memory topology, including how TCMs relate to shared memory like OCM and external DDR, is shown in Figure 3.6. This view helps clarify how the RPU accesses local and global memory resources under different configurations.

In addition to supporting predictable execution, TCMs play an important role in how faults appear and propagate through the system. Because they are directly connected to the processor and operate with well defined timing and memory layouts, they provide clear points of failure. They are also points where faults can be observed, injected, or controlled. Their predictable behavior under both split and lockstep modes allows fault injection tests to isolate timing effects, assess fault coverage, and correlate system responses with specific memory transactions.

3.3.4 On-Chip Memory (OCM)

The On-Chip Memory (OCM) in the Zynq UltraScale+ MPSoC provides a centralized, shared SRAM resource totaling 256 KB, physically partitioned into four 64 KB memory banks. It is accessed through a 128-bit AXI slave interface and operates at up to 600 MHz, offering low-latency performance for cores such as the Cortex-R5. Compared to external DDR memory, OCM is notably faster and more predictable, making it a critical shared memory region for early stage boot processes, intercore communication, and system fault handling.

Each of the four banks in the OCM is protected by Error Correction Code (ECC), capable of detecting multi-bit errors and correcting single-bit faults. This makes it the preferred region for fault tolerant communication buffers and shared status storage, especially in systems where high reliability is required. The OCM's error detection capabilities complement the lockstep mechanism of the RPU and provide additional observability into soft-error effects that may appear during runtime.

All access to OCM is routed through the OCM switch and passes through a Xilinx Memory Protection Unit (XMPU).The XMPU enforces access control by dividing the OCM into 64 separate 4 KB regions. Each region can be configured to allow or block access based on the requesting processor or subsystem.

This enables isolation between components like the RPU, APU, and PMU, which is highly important in systems running tasks with different safety or security requirements [18, 3, Chapter 11.5.2].

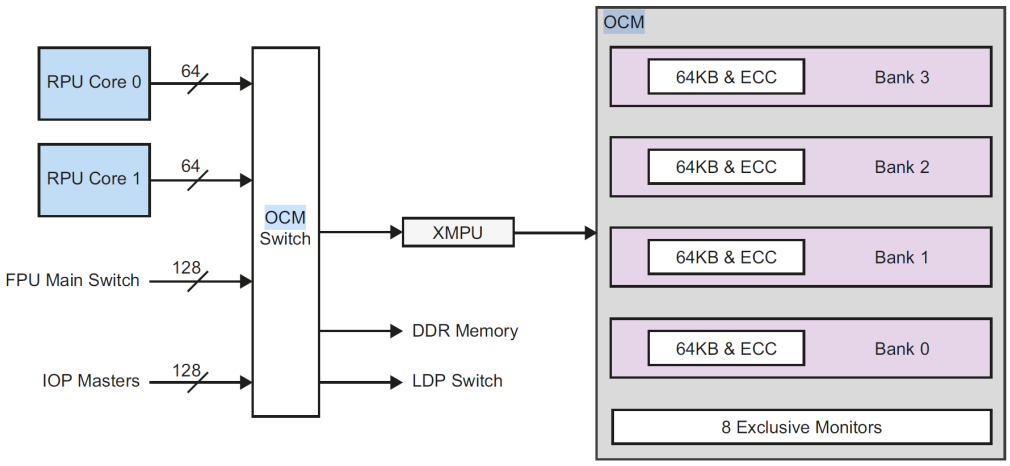

Figure 3.7: OCM Architecture: Four 64 KB ECC-protected memory banks connected via an OCM switch to multiple system domains. [18, Chapter 11.5.2].

Since performance depends on how data is read from and written to OCM, the memory is organized into 256 bit wide physical lines. This means that operations that are aligned to 256 bit boundaries and use full 256 bit transfers are faster. If transfers were smaller or misaligned, the memory would have to do extra work like reading, modifying or writing back data and that would slow things down.

In early system operation, the OCM plays a key role as the execution region for the First Stage Boot Loader (FSBL). This stage is responsible for initializing the processor subsystems, clocks, peripherals, and loading higher stage software such as baremetal applications or operating systems. Its usage during boot reinforces the memory's importance in deterministic startup sequence and secure system configuration.

The OCM is also used as a shared buffer for communication between processors and management units. For example, IPI messages often include status flags or data structures placed in OCM. The PMU relies on this region to exchange error information or runtime updates with other cores, utilizing the fast access and central location within the system. [18, 3, Chapter 11.5.2]

Memory Bank	Size	Address Range
0	64 KB	FFFC_0000 — FFFC_FFFF
1	64 KB	FFFD_0000 — FFFD_FFFF
2	64 KB	FFFE_0000 — FFFE_FFFF
3	64 KB	FFFF_0000 — FFFF_FFFF

Figure 3.8: OCM Memory Map: The address ranges of the four OCM banks, each mapped to a 64,KB region. [18, Chapter 11.5.2].

In summary, the OCM acts as a high speed shared memory with strong reliability and isolation features. Its use during early boot, fault messaging, and real-time coordination across processors makes it a central component in both performance and dependability considerations.I ECC capabilities ensure single bit correction and multi bit error detection across system transactions. Because of these combined architectural

features, the OCM has a foundational role not only in system startup and runtime communication, but also in enabling fault detection and mitigation workflows involving the PMU, IPI subsystem, and secure firmware.

Main DDR Memory

The RPU can also access external DDR memory through the PS-side AXI interconnect. While DDR offers a large address space, its latency is non-deterministic due to the use of complex memory controllers, variable burst timing, and contention from other system components. For this reason, DDR is typically avoided for time-critical routines but may be used for non-real-time tasks or large data buffers [3, Chapter 4].

This memory hierarchy is designed to balance speed, predictability, and storage size. Time-critical code is usually placed in TCMs because they give consistent, fast access. Less critical shared can be stored in OCM, while large or background tasks can use the main DDR memory. In real-time systems, the most important factor is not just how fast memory access is, but how predictable that access time remains. This is what determinism is in that context. TCMs always respond in the same amount of time, no matter what the system is doing. In contrast, caches and DDR memory can be unpredictable due to delays from other traffic or cache misses. For real-time applications, that kind of variability is a risk [18, Chapter 7].

3.3.5 Generic Interrupt Controller (GIC)

The interrupt system for the Real-Time Processing Unit (RPU) in the Zynq UltraScale+ MPSoC is built around the ARM PrimeCell PL390 Generic Interrupt Controller (GIC), which is compliant with the GICv1 architecture. This controller is responsible for managing both internal and external interrupts targeting the Cortex-R5 cores. It is located within the Low Power Domain (LPD) and serves as the central hub for interrupt routing, prioritization, and delivery.

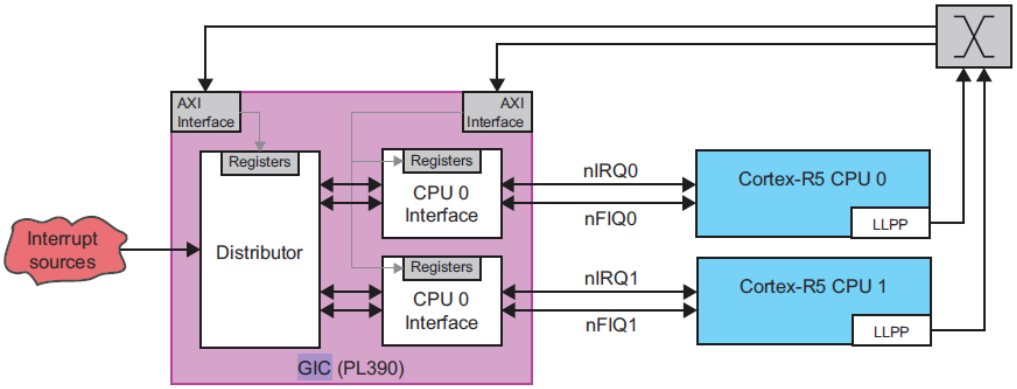

Figure 3.9: Block diagram of the RPU interrupt system, showing the PL390 GIC distributor and its interfaces to the Cortex-R5 cores [18, Chapter 7, p. 169].

Interrupt Delivery Architecture

The PL390 GIC is divided into a *distributor* and per-core *CPU interfaces*. The distributor receives all incoming interrupts, including shared peripheral interrupts (SPIs), software generated interrupts (SGIs) and private peripheral interrupts (PPIs). It determines their priority and target cores. It also forwards active, high-priority interrupts to the CPU interfaces, which then raise interrupt signals (nIRQ or nFIQ) to the corresponding Cortex-R5 processors.

A key architectural feature is the dual interrupt lines per core: **IRQ** (standard) and **FIQ** (fast). FIQs

support lower latency through higher priority, direct vectoring, and dedicated register banking. This makes them ideal for critical tasks that require guaranteed rapid responses, such as hard real-time feedback loops[3].

Determinism and Responsiveness

Interrupt latency is a critical factor in meeting real-time requirements, and it depends on both processor design and software behavior. The Cortex-R5 includes a feature called restartable instructions, which helps reduce worst-case interrupt delays. Normally, a long instruction such as a multi register load must finish before an interrupt can be handled. With restartable instructions, the processor can pause certain operations, handle the interrupt, and then restart the original instruction, improving responsiveness in time-critical paths.

In addition, the ARMv7-R architecture includes dedicated instructions to speed up interrupt handling. The `SRS` (Store Return State) instruction quickly saves the processor's current context to the stack. The `RFE` (Return From Exception) restores the saved context and resumes execution after the interrupt. The `CPS` (Change Processor State) is used to enable or disable interrupts by changing the processor mode. Together, these instructions reduce the overhead of entering and exiting interrupt service routines, helping the system respond more consistently and with lower jitter..

In the GIC, fast response is supported by its direct connection to the Cortex-R5 processors through the Low-Latency Peripheral Port (LLPP). This dedicated interface avoids delays that could occur if the GIC had to share a general-purpose AXI bus with other system components. As a result, interrupt signals and configuration data can reach the processor more quickly, which helps reduce overall interrupt latency. [18]

Integration with Lockstep Operation

In lockstep mode, the GIC plays an important role in maintaining instruction and data coherence between R5_0 and R5_1.This means that both Cortex-R5 cores must receive exactly the same inputs to stay synchronized. To achieve this, the GIC sends interrupts only to the main core (R5_0), while the second core (R5_1) passively follows the same execution path. This strategy ensures that both cores remain in identical execution states therefore preserving the fault detection properties of lockstep execution [18].

Fault Signaling and Error Reporting

The GIC is also responsible for routing interrupt signals related to fault conditions—such as ECC errors in TCMs or the OCM—to the appropriate processing unit. Interrupt lines can be mapped to the RPU or APU depending on system requirements, and are categorized in the global interrupt map of the device. These signals are critical for enabling immediate exception handling or system recovery routines in response to soft errors[3].

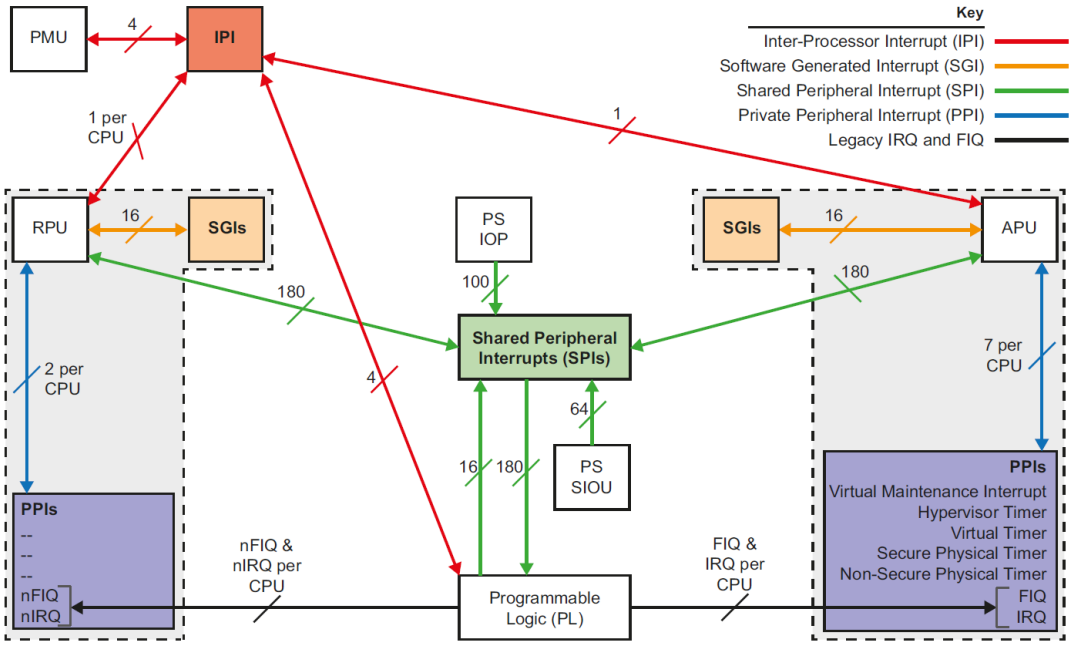

Figure 3.10: Zynq UltraScale+ MPSoC interrupt topology. Shows the routing of IPI, SGI, SPI, and PPI across system components [18, Chapter 11, p. 280].

3.4 Fault Detection and Correction Mechanisms

Fault resilience is a core requirement in systems targeting safety critical domains such as automotive electronics, industrial automation, and avionics. In such environments, silent data corruption or missed deadlines can result in dangerous outcomes. The Zynq UltraScale+ MPSoC addresses this by integrating multiple fault detection and mitigation mechanisms within its Real-Time Processing Unit (RPU).

A central feature of this fault-tolerant design is the use of lockstep mode in the Cortex-R5 cores, a technique that compares the outputs of two identical processor cores running the same instructions in parallel. Lockstep is widely used in systems that require high fault coverage, such as automotive controllers certified under ISO 26262 and safety systems in rail networks. Alongside lockstep, the Zynq UltraScale+ MPSoC integrates additional safety mechanisms including Error Correcting Codes (ECC), parity protection, and Inter-Processor Interrupt (IPI) signaling. These features help to detect and isolate faults in both logic and memory subsystems, working in coordination with the Platform Management Unit (PMU), which oversees fault handling and orchestrates system-level responses.

3.4.1 Lockstep Fault Detection in the RPU

Lockstep execution is a fault detection technique that operates by running the same instructions on two processor cores in parallel and continuously comparing their outputs. In the Zynq UltraScale+ MPSoC, the Cortex-R5 cores within the Real-Time Processing Unit (RPU) can be configured to operate in lockstep mode, where one core (typically CPU 1) mirrors the execution of the other (CPU 0). If the outputs differ at any point, a fault is assumed to have occurred, and an error signal is generated for further handling by the system.

Redundancy and Fault Coverage

Redundancy is a fundamental principle in the design of fault-tolerant systems, representing the intentional duplication or multiplication of critical system components, information, or operations beyond the minimum required for basic functionality [48]. The primary objective of redundancy is to ensure that the system can continue to operate correctly even when one or more of its components fail. In safety-critical applications—such as aerospace, automotive, medical devices, and industrial control systems—redundancy serves as the cornerstone of achieving high reliability and availability.

Types of Redundancy

Redundancy techniques can be classified into four fundamental categories, each addressing different aspects of fault tolerance :

- **Hardware Redundancy** involves the physical replication of critical hardware components. This is the most direct form of redundancy and includes techniques such as:
  - *Dual Modular Redundancy (DMR)*: Two identical components operate in parallel, with comparison logic detecting discrepancies between their outputs.
  - *Triple Modular Redundancy (TMR)*: Three identical modules execute the same operations, with a majority voter determining the correct output, providing fault masking capability .
  - *N-Modular Redundancy (NMR)*: A generalization where N identical modules operate concurrently, with voting logic that can tolerate up to $\lfloor (N-1)/2 \rfloor$ faulty modules [14].

  Hardware redundancy provides immediate fault detection (in DMR) or fault masking (in TMR and higher-order NMR), but comes at the cost of increased area, power consumption, and system complexity. For instance, TMR implementations typically incur approximately 200% area overhead and 214% energy overhead compared to a simplex (non-redundant) system [38].
- **Software Redundancy** employs multiple versions of software, typically developed by independent teams using diverse design approaches, to mitigate systematic faults that might be present in a single implementation. This technique, known as N-version programming or design diversity, is particularly effective against software design errors but requires significant development effort and careful management of common-mode failures [12].
- **Information Redundancy** adds extra bits to data to enable error detection and correction without duplicating the entire hardware. Key techniques include:
  - *Parity Checking*: Adds a single bit to detect odd numbers of bit errors.
  - *Error Correcting Codes (ECC)*: Typically uses Hamming codes or Reed-Solomon codes to provide Single-Error Correction and Double-Error Detection (SECDED) capability [47].
  - *Cyclic Redundancy Checks (CRC)*: Used primarily for detecting errors in data transmission and storage.

  ECC is widely deployed in memory subsystems and data buses to protect against soft errors caused by radiation-induced single-event upsets (SEUs). A typical SECDED implementation for a 64-bit data word requires 8 additional check bits, representing only 12.5% overhead compared to 200% for full hardware duplication [42].
- **Time Redundancy** performs the same operation multiple times on the same hardware, comparing results to detect transient faults. While this approach adds minimal hardware overhead, it increases execution time and is effective only for transient faults, not permanent hardware failures. Variants include:

- *Recomputation with shifted operands*: Helps detect faults that might produce identical errors on repeated executions.
- *Alternating logic*: Uses complemented logic in successive executions to detect stuck-at faults.

International functional safety standards mandate specific levels of redundancy based on the criticality of the application. ISO 26262, the automotive functional safety standard, defines four Automotive Safety Integrity Levels (ASIL) from A (lowest) to D (highest) [26]. For ASIL D applications the most stringent safety level required for systems whose failure could result in severe injury or death—hardware redundancy such as dual-core lockstep operation is strongly recommended or required [22]

The standard specifies target metrics for hardware architectural metrics including:

- *Single Point Fault Metric (SPFM)*: The fraction of single-point faults covered by safety mechanisms.
- *Latent Fault Metric (LFM)*: The fraction of latent faults (undetected faults that could lead to safety violations when combined with subsequent faults) that are covered.

For ASIL D compliance, typical requirements include SPFM $> 99\%$ and LFM $> 90\%$, which generally necessitate hardware redundancy combined with diagnostic coverage [41].

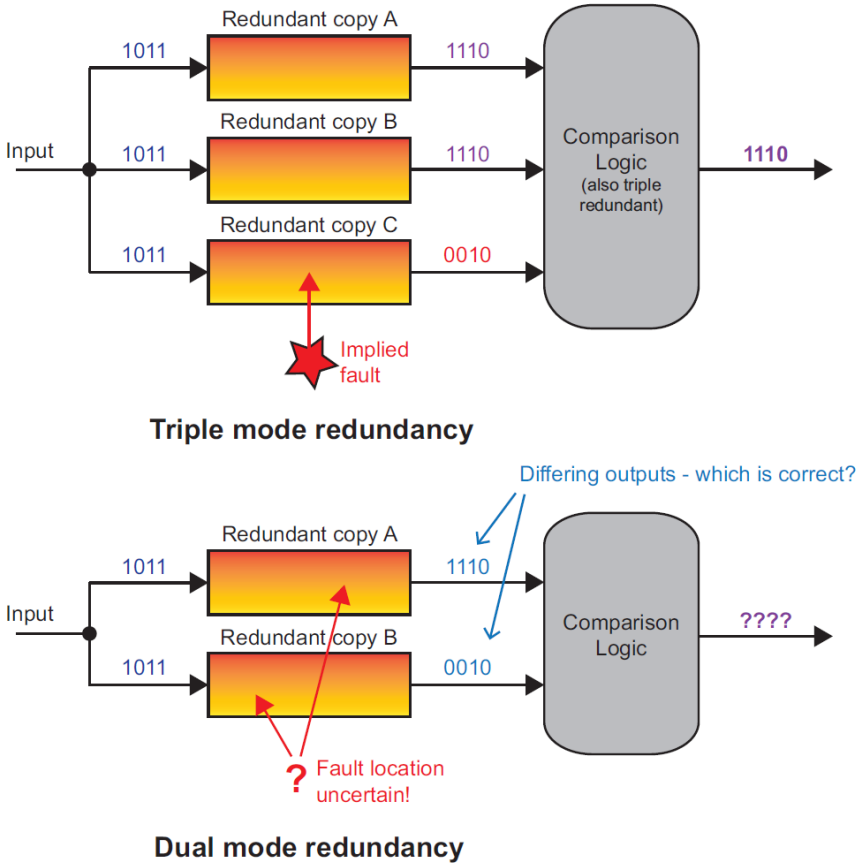

Figure 3.11: Comparison of triple and dual modular redundancy for fault detection. Adapted from [18, Chapter 9, p. 228].

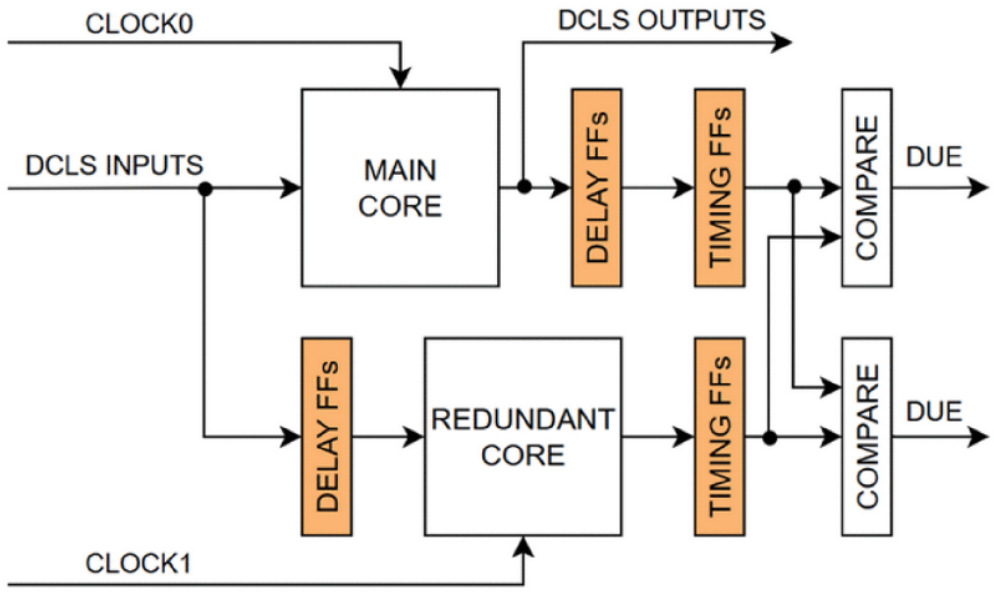

Figure 3.12: Dual-Core lockstep system example [30].

Dual Modular Redundancy and Lockstep Architecture

Dual Modular Redundancy (DMR) represents a specific form of hardware redundancy where two identical functional units execute the same operations in parallel, with their outputs compared to detect discrepancies . Lockstep operation is the most common implementation of DMR in processor systems, where two CPU cores execute the same instruction stream synchronously, maintaining identical architectural state throughout execution. It offers significantly lower hardware overhead compared to TMR, making it suitable for embedded safety systems with tight resource budgets. It is commonly applied in domains where high availability is required, but full fault masking is not necessary.

In a lockstep configuration, the system comprises two identical processor cores—typically designated as the master (or primary) core and the shadow (or redundant) core. Both cores receive identical inputs, including instruction fetches, data reads, and interrupt signals. They execute instructions in perfect synchronization, advancing through the same pipeline stages simultaneously. At each clock cycle, comparison logic monitors critical output signals from both cores [7].

The key principle underlying lockstep operation is *determinism*: given identical inputs and identical initial states, two identical processors executing the same instruction sequence must produce identical outputs at every cycle.This architecture provides *fault detection*: when the two cores produce different outputs, the system immediately knows a fault has occurred. However, Dual Modular Redundancy (DMR) does not provide *fault correction*. When a mismatch is detected, the system cannot determine which of the two cores produced the incorrect result. Both cores are functionally identical, so a mismatch proves that at least one has failed, but comparison alone is not able to identify which one has faulted. [30].

Lockstep in Cortex-R5 Processor

The Cortex-R5 can be configured to operate in Dual-Core Lock-Step (DCLS) mode, where two identical processor cores—designated master and redundant (or shadow) execute the same instruction sequence in perfect synchronization. This configuration provides cycle-by-cycle fault detection through continuous comparison of core outputs.

In DCLS configuration, both cores receive identical inputs including instruction fetches, data reads, and interrupt signals. The master core drives the processor output pins and controls shared cache memories, while the redundant core operates in parallel without directly affecting system outputs. This asymmetric arrangement ensures that only verified results—those agreed upon by both cores—propagate to the memory system and external interfaces.

To minimize area overhead while maintaining fault coverage, the Cortex-R5 DCLS implementation employs selective resource sharing:

Shared Resources: Instruction and data cache memory arrays are shared between cores, as these memories are independently protected by SECDED ECC mechanisms. Similarly, TCM interfaces are shared, with ECC protection available on TCM data. This sharing reduces the area penalty from potentially 300% (full duplication) to approximately 185-200% relative to a single-core implementation [31].

Duplicated Logic: Critical processor components are fully replicated in each core, including instruction fetch and decode logic, execution pipelines (ALU, multiply-accumulate units, load-store units), register files (general-purpose and system registers), Memory Protection Unit, and pipeline control sequencing .

Comparison Logic: Dedicated comparison circuitry continuously monitors outputs from both cores at every clock cycle. Compared signals include memory transaction addresses and data, control signals for cache operations, external bus transaction requests, and interrupt acknowledgments. The comparison logic generates error indication signals when mismatches are detected, specifically through the DCCMOUT[7:0] output signals that can be monitored by external system logic, and DCCMINP[7:0] plus DCCMINP2[7:0] input signals that allow external comparison logic to communicate with the processor [7].

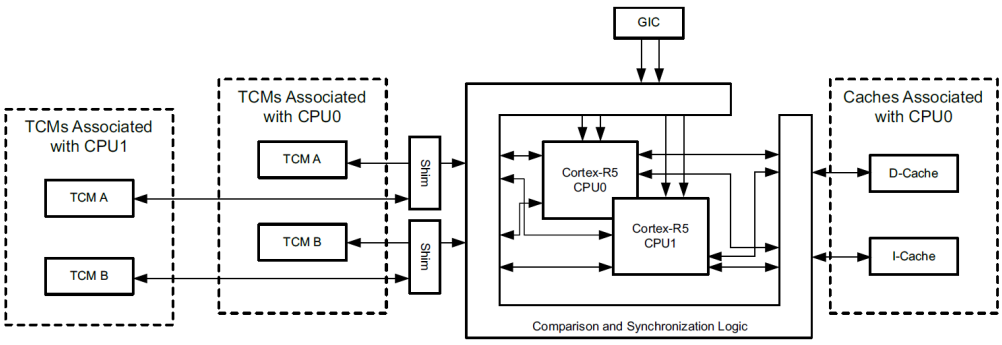

Figure 3.13: RPU Cortex-R5 Processor Lock-step Mode [3].

Power consumption in DCLS mode approximately doubles relative to single-core operation: dynamic power consumption doubles due to switching activity in both cores, clock tree power increases by 80-100%, and static leakage power nearly doubles due to transistor duplication. However, throughput performance remains identical to single-core operation since both cores execute the same instruction stream, contrasting favorably with time-redundancy approaches that double execution time. [7]

Zynq Ultrascale Plus RPU Configuration for Lockstep Mode

The operating mode of the RPU is controlled by two primary configuration signals that must be set during power-on reset and cannot be changed dynamically during runtime. SLSPLIT (Split/Lock Control) determines the fundamental operating mode: when set LOW (0), the RPU configures in lockstep (safety) mode where both R5_0 and R5_1 execute identically; when set HIGH (1), the RPU operates in split (performance) mode where each core executes independently. SLCLAMP (Output Clamp Control) manages the multiplexing and clamping logic specific to lockstep configuration; when LOW in lockstep mode, the shadow core (R5_1) outputs are clamped to prevent interference with the master core (R5_0) outputs, ensuring only the master core drives system outputs.

These configuration signals are set through System-Level Control Registers (SLCR) before releasing the RPU from reset. Once the RPU exits reset in a particular mode, the mode selection is static and fixed for the entire system runtime. Changing modes requires a complete power-on reset sequence, making it impossible to dynamically switch between split and lockstep operation during application execution. This static configuration approach simplifies the hardware design and eliminates race conditions that might arise from mode transitions during execution [3, 4].

Memory Organization in Lockstep

Memory organization differs significantly between split and lockstep modes, affecting the total available memory for applications. In split mode, each core has its own dedicated Tightly-Coupled Memory (TCM): ATCM (Address TCM) of 64 KB per core, and BTCM (Block TCM) of 64 KB per core. In lockstep mode, these memories are combined into unified address spaces that both cores can access together: ATCM becomes a single 128 KB block, and BTCM becomes a single 128 KB block.

This memory unification provides a practical benefit: the lockstep processor has twice as much fast, low-latency memory compared to a single core. This helps compensate for the disadvantage of lockstep mode, where both cores execute the same program rather than running two different applications in parallel.

Cache memories work differently: each core keeps its own 32 KB instruction cache and 32 KB data cache. However, in lockstep mode, both caches must always hold identical data. This is maintained automatically by hardware: when one core reads or writes to cache, the hardware ensures the other core's cache is updated identically. Both caches are also protected by ECC (Error Correcting Code) mechanisms, which detect and correct single-bit errors in the cached data, independent of the lockstep fault detection [3].

Error Detection and Response

The comparison logic continuously monitors both R5_0 and R5_1 every single clock cycle throughout program execution. It checks whether the outputs from both cores match, examining memory addresses and data when the cores fetch instructions or read and write data, control signals for cache operations, signals on the AXI bus that connects the processor to the rest of the system, transactions with peripheral devices, and debug interface activity. The comparison happens after both cores have completely finished their pipeline operations for that clock cycle, ensuring all processing is complete before outputs are compared.

Another important feature of the Lockstep mode is temporal diversity: the two cores operate slightly offset in time by 2 clock cycles as shown in Figure 3.14. This time offset provides protection against radiation events that might otherwise affect both cores simultaneously. If a cosmic ray or other radiation strike causes a transient fault, the 2-cycle delay means the cores are processing different instructions at the exact moment of the event, reducing the likelihood that both cores produce the same incorrect result. This temporal separation helps defend against common-mode failures caused by external interference.

When a mismatch is detected—meaning the two cores produced different outputs in the same clock cycle—the comparison logic immediately asserts error signals. The system logs detailed error information in dedicated status registers, recording the memory address and type of operation (read, write, instruction fetch) where the mismatch occurred. The Platform Management Unit (PMU), which oversees system-wide error handling and power management, is notified of the lockstep error either through an interrupt signal or by monitoring dedicated status registers. The PMU then reads the error information and determines the appropriate response based on policies that were configured when the system was initialized.

The system response to detected lockstep errors is configurable and depends on the application's safety requirements. For safety-critical applications requiring ISO 26262 ASIL D certification (the highest automotive safety level), the typical response is immediate: the system transitions to a safe state or performs a controlled shutdown, preventing any potentially incorrect computations from affecting system operation. For less critical applications, software-based error handlers may attempt recovery procedures. These handlers can log the error for later analysis, perform diagnostic tests to determine the nature of the fault, attempt

to restore correct state by rolling back to a previously saved checkpoint, or transition the system to a degraded mode where it continues operating with reduced functionality. Some systems implement threshold-based responses: the first detected error triggers logging and a recovery attempt, but if multiple errors occur within a short time window, the system concludes a serious fault exists and transitions to shutdown. Critical systems prioritize safety by shutting down immediately, while systems that must maintain availability can attempt recovery before resorting to shutdown. [3], [4], [18].

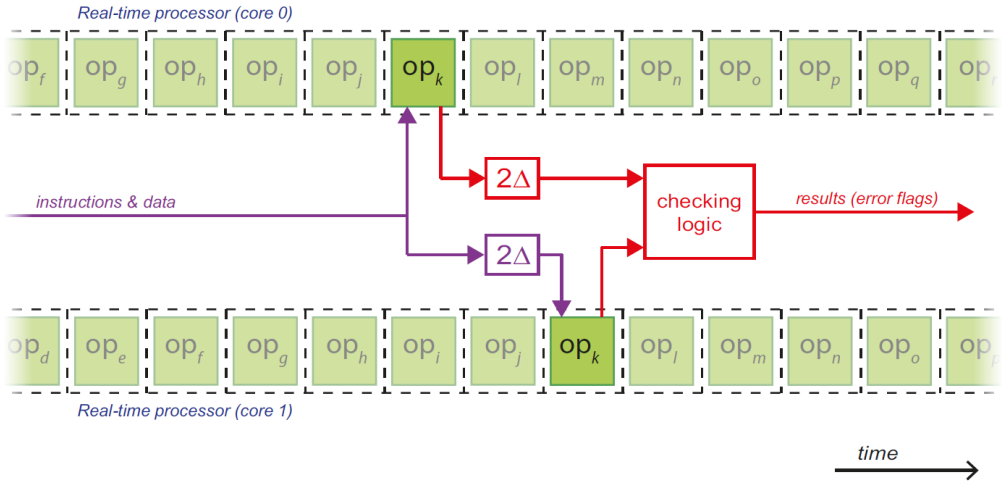

Figure 3.14: Intuitive diagram of lockstep operation (operating both Arm Cortex-R5 cores with a time offset and checking mechanism) [18, Chapter 9 p. 234].

Trade-offs and Limitations

Operating the RPU in lockstep mode requires accepting several significant trade-offs. The silicon area overhead is approximately 185-200% relative to a single-core system. This means that the lockstep implementation uses almost twice as much chip area as a single core: about 180-190% for the duplicated processor logic (two complete pipelines, register files, and control units), plus 2-5% for the comparison circuitry that monitors both cores, plus 1-3% for error detection and reporting. While this is substantially less than Triple Modular Redundancy (which would require 300% additional area), it still represents a significant cost in chip size and manufacturing expense [31].

Power consumption approximately doubles in lockstep mode compared to single-core operation . Both cores are actively executing, so the electrical power needed doubles. Additionally, the clock distribution network must supply twice as many circuits, increasing power consumption by another 80-100%. The static power (power consumed even when the processor is idle) nearly doubles because of the doubled transistor count. This doubling of power consumption has practical consequences: systems require larger power supplies, more effective cooling, and suffer reduced battery life in portable applications [6].

However, lockstep operation does not harm execution speed. Since both cores execute the same instructions in parallel, the overall execution time remains the same as a single-core system. This is a significant advantage compared to time based redundancy (executing the same code twice sequentially on a single core) which doubles the execution time. Software-based checking techniques also add overhead to program execution time, but lockstep avoids this penalty completely. [6].

Another fundamental limitation is that lockstep detects faults but does not automatically correct them . When the comparison logic detects a mismatch between the two cores, the system knows a fault has occurred, but it cannot determine which core is producing incorrect results. Both cores are identical, so a mismatch proves at least one has failed, but the system cannot automatically decide which output is correct. The system must rely on external mechanisms (software error handlers or external monitoring) to decide what action to

take: attempt recovery, transition to a safe state, or shut down. In contrast, Triple Modular Redundancy with three cores would allow automatic correction—the system could simply choose the output that two of three cores agree on—but at the cost of tripling area and power.

Common-mode failures cannot be detected by lockstep. These are faults that affect both cores identically, causing both to produce the same (incorrect) result. Examples include a faulty clock signal supplied to both cores, a power supply voltage drop affecting both cores, failures in shared hardware like the TCM memory interface or a software bug that causes both cores to compute the same wrong answer. Since both cores produce identical outputs, the comparison logic sees no mismatch and does not detect the failure.

Protecting against common-mode failures requires additional mechanisms beyond lockstep: independent monitoring of clock and power supply signals to detect distribution faults, software checks that verify computation results are plausible, watchdog timers that detect if the processor hangs, and careful analysis of potential systematic design faults. These complementary protections must be designed and verified during the system safety certification process [4].

3.4.2 Error Correction Code (ECC) in Memory

Most memories in the Zynq UltraScale+ MPSoC Real-time Processing Unit (RPU) employ Error Correction Code (ECC) as a critical safety feature to detect and correct errors caused by radiation, electromagnetic interference, or other transient faults. The OCM, TCM, L1 caches, and DDR memory all use ECC protection, but with different configurations tailored to their access patterns and performance requirements.

ECC Protection Across RPU Memories

The Cortex-R5 processor cluster uses SEC-DED ECC protection for multiple memory subsystems, each with configurations optimized for their specific usage patterns:

- **Tightly-Coupled Memories (TCM)**: Each Cortex-R5 core has dedicated tightly-coupled memories comprising 128 KB of ATCM (instruction-focused TCM) and 128 KB of BTCM (data-focused TCM) when operating in lock-step mode. Both TCM banks are protected with 32-bit ECC. The TCM is accessed directly by the processor with deterministic, low-latency timing, making it ideal for storing interrupt handlers and time-critical data. The 32-bit ECC protection was selected because TCM operations frequently involve byte and halfword accesses, where 32-bit ECC minimizes the performance penalty of read-modify-write operations.
- **L1 Instruction and Data Caches**: Each Cortex-R5 core contains 32 KB of instruction cache and 32 KB of data cache. These caches can be configured with either 32-bit or 64-bit ECC protection, depending on the expected access patterns. Instruction caches typically use 64-bit ECC because instruction fetches are predominantly 64-bit aligned and doubleword-sized, making the higher memory overhead of 64-bit protection acceptable. Data caches often use 32-bit ECC to minimize the performance impact of read-modify-write operations on byte and halfword accesses.
- **On-Chip Memory (OCM)**: The 256 KB OCM is shared across the system and accessible by the RPU, APU, and other bus masters. The OCM uses 32-bit ECC protection, balancing error detection capability with performance for mixed access patterns. The OCM frequently experiences byte, halfword, and word accesses from various masters, making 32-bit ECC the optimal choice.
- **DDR Memory Controller**: The external DDR memory controller supports ECC protection for the off-chip DRAM. Unlike the on-chip memories, the DDR controller uses a 64-bit ECC configuration with 8 syndrome bits, optimized for the naturally aligned, burst-oriented access patterns typical of DRAM transactions.

The remainder of this subsection focuses specifically on the ECC implementation in the OCM, with

detailed explanations of the Hamming code encoding, syndrome calculation, and error correction mechanisms [6], [52], [8].

ECC Overiview

Error Correction Code (ECC) is a method of detecting and correcting bit errors in memory by storing additional redundant information alongside the actual data. In silicon devices, stray radiation (such as cosmic rays and alpha particles from package materials), electromagnetic interference, and other environmental effects can cause a bit stored in memory to spontaneously flip from its correct value—changing a 0 to a 1 or vice versa. As mentioned in Chapter 2 this is called a soft error because the memory cell itself is not permanently damaged. The error is temporary and specific to the affected bit.

A 32-bit Single Error Correction - Double Error Detection (SEC-DED) approach provides the following capabilities:

- For each 32-bit word of data, seven additional ECC protection bits are computed and stored alongside the data
- The ECC can **correct** any single bit error that occurs within the 32-bit data word or its associated protection bits
- The ECC can **detect** (but not correct) any double bit error that occurs within the 32-bit data word or its associated protection bits
- If three or more bit errors occur simultaneously in a single 32-bit word, the ECC may fail to detect them, or worse, may misinterpret them as a single-bit error and attempt to correct the wrong bit

The choice of 32-bit ECC (rather than 64-bit ECC) represents a practical trade-off between error detection capability and the amount of extra memory needed to store the protection bits. This trade-off is better revealed considering the memory overhead for protecting 64 bits of data:

- **32-bit ECC approach**: To protect 64 bits of data, the 32-bit ECC protection must be applied twice—once for each 32-bit half. This requires 7 protection bits for the first 32-bit word plus 7 protection bits for the second 32-bit word, totaling 14 extra bits to protect 64 bits of data. The overhead is therefore 14/64 = 21.9%
- **64-bit ECC approach**: A single 64-bit ECC protection applies to the entire 64-bit word at once, requiring only 8 protection bits total. The overhead is therefore 8/64 = 12.5%

While 64-bit ECC has lower memory overhead, it is less suited to the a memory's access pattern. For example the Cortex-R5 processor frequently performs byte and halfword writes to the OCM. With 64-bit ECC, every sub-word write triggers an expensive read-modify-write operation that must recalculate all 8 protection bits. With 32-bit ECC, the overhead is still present but involves recalculating only 7 protection bits per 32-bit word, and many writes fit cleanly into a single 32-bit boundary. Therefore, for the OCM where write performance matters, the 32-bit approach provides better real-world performance despite higher memory overhead [7].

Hamming Code Structure and Syndrome Calculation

The 32-bit SECDED protection used in the OCM is based on an extended Hamming code. The Hamming code itself provides single-error correction, and an additional overall parity bit extends it to also detect double-bit errors. Figure 3.15 shows the organization of a Hamming code for 26 data bits with 5 parity bits. OCM uses a similar structure scaled to 32 data bits with 6 Hamming parity bits plus 1 overall parity bit (total 7 protection bits).

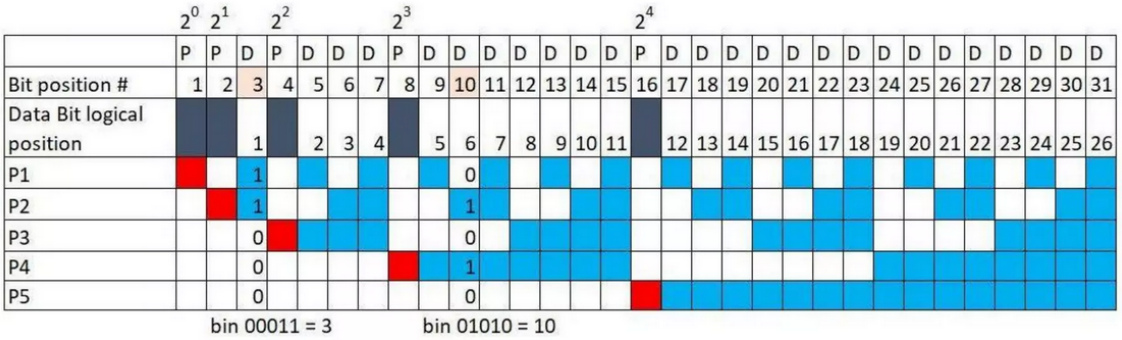

Figure 3.15: Hamming code organization showing parity bits (P) at power-of-2 positions and their coverage of data bits (D) [39].

The key principle of Hamming codes is that parity bits are placed at positions that are powers of 2 (positions 1, 2, 4, 8, 16, etc.), and each parity bit covers all bit positions that have the corresponding bit set in their binary representation. As shown in the figure:

- **P1** (at bit position 1 = 2^0) covers all positions with bit 0 set in their binary representation: positions 1, 3, 5, 7, 9, 11, ... (highlighted in red in the figure)
- **P2** (at bit position 2 = 2^1) covers all positions with bit 1 set: positions 2, 3, 6, 7, 10, 11, 14, 15, ... (highlighted in red)
- **P3** (at bit position 4 = 2^2) covers all positions with bit 2 set: positions 4, 5, 6, 7, 12, 13, 14, 15, ... (highlighted in red)
- **P4** (at bit position 8 = 2^3) covers all positions with bit 3 set: positions 8, 9, 10, 11, 12, 13, 14, 15, 24, 25, ... (highlighted in red)
- **P5** (at bit position 16 = 2^4) covers all positions with bit 4 set: positions 16 and beyond with bit 4 set (highlighted in red)

The data bits occupy all positions that are not powers of 2. The figure shows data bits labeled with their logical position (1 through 26 in this example), distinct from their physical bit position in the encoded word.When a 32-bit word is ***written*** to the OCM, the ECC encoder calculates the seven syndrome bits as follows:

1. **Hamming Parity Bits (6 bits)**: Each of the six Hamming parity bits is computed as the XOR (exclusive OR) of all data and parity bits at positions covered by that parity bit. For example, P1 is the XOR of all bits at odd positions, P2 is the XOR of positions where bit 1 is set, and so on.
2. **Overall Parity Bit (1 bit)**: The seventh protection bit is an overall parity bit computed as the XOR of all 32 data bits and all 6 Hamming parity bits. This additional bit enables the distinction between single-bit and double-bit errors.
3. **Storage**: The 32-bit data word and 7-bit syndrome (6 Hamming + 1 overall parity) are stored together in the OCM RAM, requiring a total of 39 bits of physical storage per 32-bit logical word.

When data is ***read*** from the OCM, the ECC decoder performs the following operations to detect and potentially correct errors:

1. **Retrieve**: Read both the 32-bit data word and its 7-bit syndrome (6 Hamming parity + 1 overall parity) from the OCM.

2. **Recalculate**: Compute what the 6 Hamming parity bits *should* be based on the retrieved data bits.
3. **XOR Comparison**: Compute the syndrome value by XOR-ing each stored Hamming parity bit with its corresponding recalculated value. This produces a 6-bit syndrome value.
4. **Check Overall Parity**: Compute the overall parity of all retrieved bits (32 data + 6 Hamming parity) and compare it to the stored overall parity bit.
5. **Error Analysis**: Based on the syndrome value and overall parity check:
    - **Syndrome = 000000 and parity matches**: No error detected , the data is valid
    - **Syndrome $\neq$ 000000 and parity does not match**: Single-bit error detected. The non-zero syndrome value directly encodes the bit position of the error. For example, if the syndrome is 001011 (binary for decimal 11), the error is at bit position 11. The error can be corrected by flipping that bit.
    - **Syndrome $\neq$ 000000 and parity matches**: Double-bit error detected. The syndrome is invalid for pinpointing an error location, and correction is not possible. The OCM controller will signal an uncorrectable error.
    - **Syndrome = 000000 and parity does not match**: An error occurred in the overall parity bit itself (single-bit error in protection bits). This can be detected but does not affect data integrity.

In Hamming codes the syndrome value directly identifies the bit position of a single-bit error. Because each parity bit covers positions with a specific bit pattern, the XOR of the parity checks produces the binary encoding of the error location. This enables rapid error correction without exhaustive searching.

Error Correction Mechanisms

When a single-bit error is detected in the OCM, the system must correct it. The Cortex-R5 processor uses two different correction strategies depending on who is reading the data [7]:

- **Correct-Inline**: For certain types of accesses, particularly AXI slave interface reads from theOCM (for example, when the APU or DMA controller accesses the OCM), the ECC logic uses correct-inline correction:
    1. The syndrome value identifies which bit is corrupted
    2. That bit is flipped to produce the correct data
    3. The corrected data is sent immediately to the requester
    4. The error remains in the OCM (not fixed in memory)

    This method is fast because no write-back to memory is needed, but the corrupted data stays in the OCM. If that memory location experiences another radiation hit before being accessed again, a double-bit error could result.
- **Correct-and-Retry**: For instruction fetches and data accesses initiated by the Cortex-R5 cores themselves, the ECC logic uses correct-and-retry correction:
    1. The syndrome value identifies which bit is corrupted
    2. That bit is flipped to produce the correct data
    3. The corrected data is written back to the OCM, replacing the corrupted value
    4. The processor re-executes the instruction to fetch the corrected data

This method takes longer (at least 9 clock cycles for error recovery), but it repairs the error in memory. This approach takes more clock cycles—at least nine cycles in the event of a correctable error—but has the critical advantage of fixing the error in memory. This prevents single-bit errors from accumulating over time. If the same memory location experiences another radiation-induced bit flip before the first error is corrected, the result would be a double-bit error that cannot be corrected. By immediately repairing single-bit errors, the correct-and-retry mechanism maintains the full protection capability of the SEC-DED code [7].

Read-Modify-Write Operations

A significant consideration for ECC-protected memory is handling writes that are smaller than the ECC data chunk size. The OCM uses 32-bit ECC, but the Cortex-R5 processor can write data as small as a single byte.

When a byte, halfword, or unaligned word write occurs, the ECC logic cannot simply compute protection bits for the new data alone. The protection bits must be calculated for the entire 32-bit word. Therefore, the ECC logic must perform a *read-modify-write* sequence:

1. **Read**: Fetch the current 32-bit word and its ECC protection bits from the OCM
2. **Check and Correct**: Verify the syndrome and correct any single-bit error in the retrieved word
3. **Modify**: Merge the new bytes being written with the unchanged bytes from the existing 32-bit word
4. **Compute ECC**: Calculate new syndrome bits (6 Hamming parity + 1 overall parity) for the complete merged 32-bit word
5. **Write**: Store the updated 32-bit word and new 7-bit syndrome back to the OCM

This sequence introduces additional latency for sub-word writes approximately 3-4 clock cycles but is necessary to maintain ECC protection integrity. The 32-bit ECC approach was chosen for the OCM because it provides better performance for the typical access patterns (many byte, halfword, and word accesses from diverse bus masters) compared to a 64-bit ECC approach, which would require read-modify-write for nearly all writes [8].

Sub-Word Write Handling

The OCM ECC protects 32-bit words, but the Cortex-R5 processor often writes data smaller than 32 bits. It often writes a single byte, a 16-bit halfword, or an unaligned word. In these cases the ECC protection bits cannot be calculated for only the new bytes. They must cover the entire 32-bit word. Therefore, when for example a sub-word write occurs, the OCM must temporarily handle the operation in five steps:

1. **Read the current word**: Fetch the existing 32-bit value and its ECC protection bits
2. **Check for errors**: Verify the ECC and correct any single-bit error
3. **Merge the data**: Replace only the bytes being written, keeping the rest unchanged
4. **Compute new protection**: Calculate new ECC bits for the complete merged 32-bit word
5. **Write back**: Store the updated word and new ECC bits to the OCM

The drawback of this process is that it adds 3 to 4 clock cycles of overhead for each sub-word write. However, the 32-bit ECC configuration was chosen for the OCM specifically because it provides better overall performance than 64-bit ECC would. With 64-bit ECC, nearly every write (even aligned word writes) would require this read-modify-write sequence [8].

Error Detection and System Response

The OCM hardware continuously monitors data as it is read and written. When an error is detected, the system captures diagnostic information and notifies the Platform Management Unit (PMU) through an interrupt:

- **Correctable Error Detected**: A single-bit error is found. The OCM records the memory address, the corrupted data, and the syndrome value. Software (the PMU firmware) can examine this information to track error patterns over time. The system continues operating because the error is automatically corrected.
- **Uncorrectable Error Detected**: A double-bit error is found. The OCM records the same diagnostic information, but correction is not possible. The system response depends on the application's safety requirements. For safety-critical applications (ISO 26262 ASIL-D) though the system typically performs an immediate controlled shutdown to prevent incorrect behavior [49].

Software can enable or disable ECC checking entirely, and can configure the ECC to operate in detection-only mode (detect errors but do not attempt correction). However, in normal operation for safety-critical systems, both detection and correction are enabled [50, 4].

Performance Impact

The ECC protection has minimal performance impact during error-free operation for aligned 32-bit accesses. However, certain access patterns can introduce additional latency:

- **Sub-word writes** (byte, halfword): Require read-modify-write, adding approximately 3-4 cycles
- **Correctable error with correct-and-retry**: Adds minimum 9 cycles to re-execute the instruction
- **Uncorrectable error**: Generates an exception, suspending normal execution

The correct-and-retry mechanism prioritizes data integrity over absolute performance. Since memory reads that encounter errors represent a small fraction of total memory traffic, this trade-off provides the best practical balance between reliability and speed [7].

3.4.3 PMU Error Detection and IPI-Based Notification

When a single-bit or double-bit error occurs in the OCM due to bit flips from radiation or other transient faults occurring in the memory, the Zynq UltraScale+ MPSoC employs a distributed detection and notification architecture. The Platform Management Unit (PMU) continuously monitors error signals from critical memory resources throughout the device. Upon detecting an OCM ECC error, the PMU captures the error information and notifies the appropriate processor using the Inter-Processor Interrupt (IPI) mechanism—a message-based communication infrastructure designed for low-latency coordination between heterogeneous processing elements. This error management flow ensures that safety-critical applications running on the Cortex-R5 can respond to memory faults with minimal delay, enabling corrective actions such as error logging, memory scrubbing, or controlled system reconfiguration.

Platform Management Unit Overview

The Platform Management Unit is the central coordination hub for power, error, and reset management in the Zynq UltraScale+ MPSoC. Located within the Low-Power Domain (LPD), the PMU operates continuously , even when other processing subsystems are powered down or in sleep states.As illustrated in Figure 3.16, the PMU comprises the following key components:

- **Triple-Redundant MicroBlaze Processor:** The PMU employs three hardened MicroBlaze processors operating in lock-step with majority voting logic. This fault-tolerant architecture ensures reliable error handling even in the presence of Single Event Upsets (SEUs) or transient faults within the PMU itself.
- **Memory Resources:** The PMU includes a 32 KB ROM containing boot code, interrupt vectors, and default service routines, plus a 128 KB RAM with Error Correction Code (ECC) protection for user-defined firmware and runtime data structures.
- **Internal Interrupt Controller:** Manages 23 interrupt sources, including four dedicated IPI channels, system error signals from PS blocks (including OCM), peripheral interrupts, and wake events from GPIO, RTC, and USB interfaces.
- **Register Blocks:** The PMU maintains two register sets: (1) *PMU Global Registers*, accessible by all bus masters in the system and containing power control, isolation, reset request, and error capture registers; and (2) *PMU Local Registers*, accessible only by the PMU processor itself, protecting safety-critical configuration and state information.
- **General Purpose I/O (GPI/GPO):** Four GPI banks and four GPO banks provide flexible signaling between the PMU and other PS/PL components. GPI1 monitors wake-up requests (including from the APU GIC and RPU GIC), while GPI2 monitors power control and reset requests.

IPI Communication Infrastructure

The inter-processor interrupt (IPI) subsystem is implemented as a dedicated hardware block in the low-power domain rather than inside the PMU itself, but the PMU is one of the eight IPI agents and owns four private IPI channels that are hard-wired to its local interrupt controller. The IPI block combines two complementary mechanisms: per-agent interrupt registers that fan out a logical interrupt *channel* from each sender to all potential receivers, and a shared pool of 32-byte message buffers that carry request and response payloads between agents in a point-to-point fashion. Together these mechanisms allow the heterogeneous processors (APU, RPU cores, PMU, and PL endpoints) to exchange short, low-latency messages without polling or shared-memory arbitration.

The heterogeneous system connects eight *IPI agents*: the APU MPCore, RPU0, RPU1, the PMU, and four PL endpoints (PL0–PL3). Each agent participates in two orthogonal structures. First, there are eleven IPI interrupt channels, which define how an interrupt raised by one agent is presented to all others. Second, there are eight sets of message buffers, which provide private 32-byte request and 32-byte response slots for every ordered pair of requester and responder agents. The channel-level view of these interrupts is illustrated in Figure 3.17, while the message-passing mesh among agents is shown in Figure 3.18.

The IPI hardware defines eleven interrupt channels in total: seven normal channels and four channels that are hard-wired exclusively to the PMU. The seven normal channels are, by default, owned by the APU (channel 0), RPU0 (channel 1), RPU1 (channel 2), and PL0–PL3 (channels 7–10), but these assignments can be changed in software because the corresponding IRQ lines are distributed to the APU GIC, the RPU GIC, the GIC proxy, and PL outputs. The four PMU channels (PMU_0–PMU_3, numbered 3–6) are fixed and their IRQ signals are only routed to the PMU interrupt controller.PMU_0 is reserved to transition the PMU into sleep mode.

In addition to the interrupt channels, the IPI hardware also provides eight *message buffer sets*. Seven of these sets can be assigned to any agent, and one set is reserved for the PMU. Each set contains 8 *Request* buffers and 8 *Response* buffers. There are 16 buffers per set and 128 buffers in total. Every buffer can hold one fixed 32-byte message. In practice, software assigns these buffers so that each pair of agents has its own request and response locations, allowing them to exchange messages without interfering with other agents, as sketched in Figure 3.18. All four PMU IPI interrupts use the same PMU-dedicated buffer set, so the PMU firmware must share that set between its different message types when sending notifications or callbacks [3].

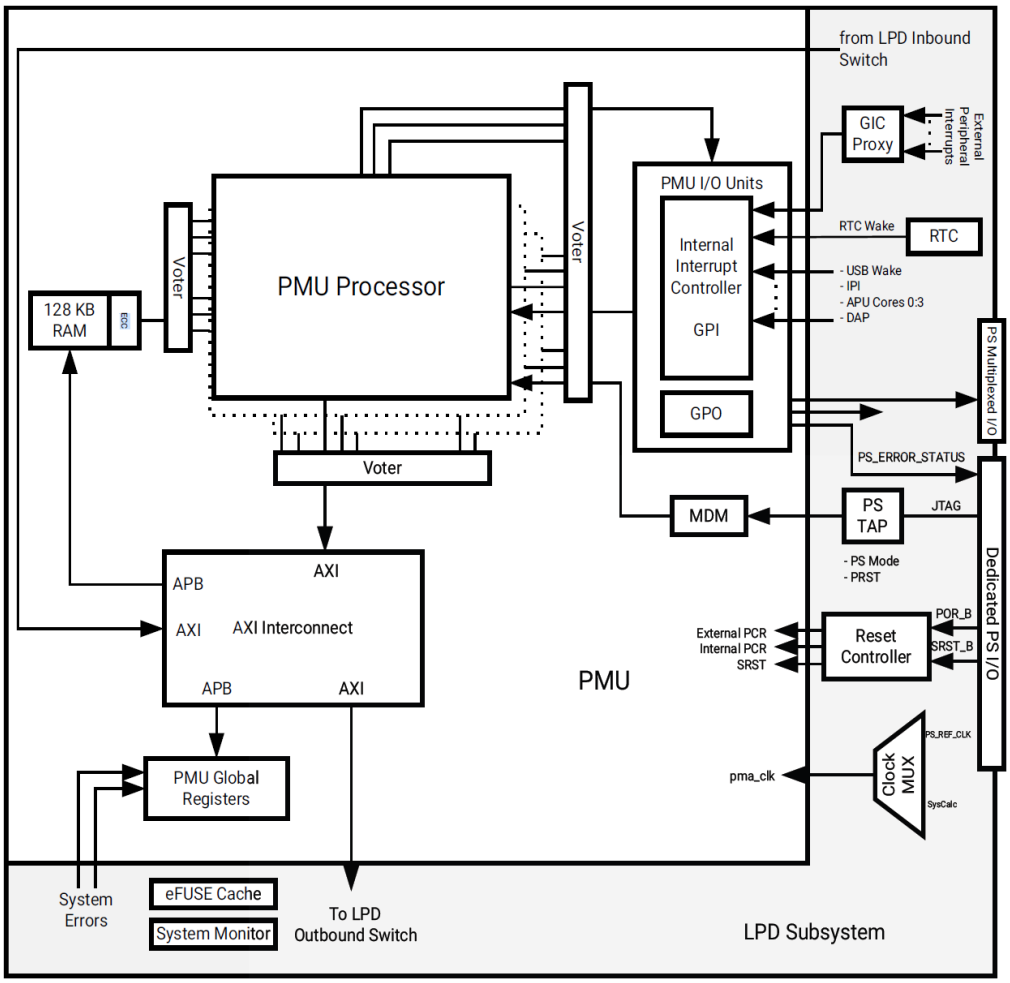

Figure 3.16: Platform Management Unit Block Diagram with triple-redundant MicroBlaze processor [3].

From the perspective of a given agent, an IPI channel is the interrupt line that connects this agent as a sender to all possible receivers. Each sending agent (APU, RPU0, RPU1, PMU_0–PMU_3, PL0–PL3) sees the same small group of six registers for each of the eleven channels [3]. Two registers represent and implement the send side:

- **TRIG (Trigger):** Write-only; writing a 1 to bit n asserts an interrupt request to receiver n on that channel.
- **OBS (Observation):** Read-only; mirrors the target agents' ISR bits so the sender can observe whether its request is still pending.

and the remaining four registers represent the receive side for that agent:

- **ISR (Interrupt Status/Clear):** Read shows which senders have raised IPIs to this agent on that channel; writing back the read value clears those bits.
- **IMR (Interrupt Mask):** Read-only; a 1 in bit n masks the corresponding ISR bit so it does not generate an IRQ, while a 0 allows it to propagate.

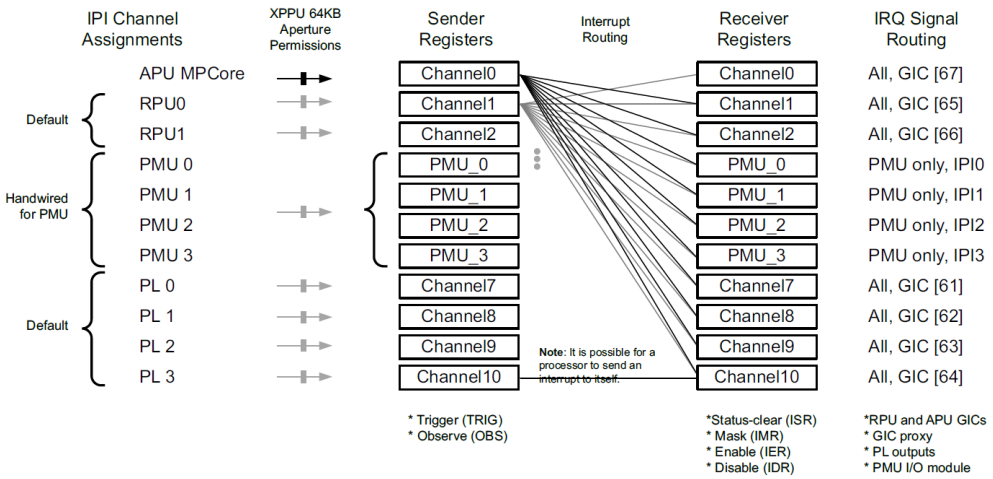

Figure 3.17: IPI Interrupt Channel Architecture. [3, Chapter 13, p. 316].

- **IER (Interrupt Enable):** Write-only; writing a 1 in bit n sets the corresponding IMR bit to 1, masking that interrupt.
- **IDR (Interrupt Disable):** Write-only; writing a 1 in bit n clears the corresponding IMR bit to 0, unmasking that interrupt [3].

Figure 3.19 illustrates this sender receiver interaction: the sender sets the appropriate bit in its TRIG register, the receiver observes the request in its ISR register, the IMR/IER/IDR registers determine whether that bit contributes to the channel IRQ. The sender can monitor progress via OBS. In a single 32-bit channel register, different bit positions correspond to different agents; for example, bit 0 encodes the APU, bits 8–9 correspond to RPU0 and RPU1, bits 16–19 to PMU_0–PMU_3, and bits 24–27 to PL0–PL3.

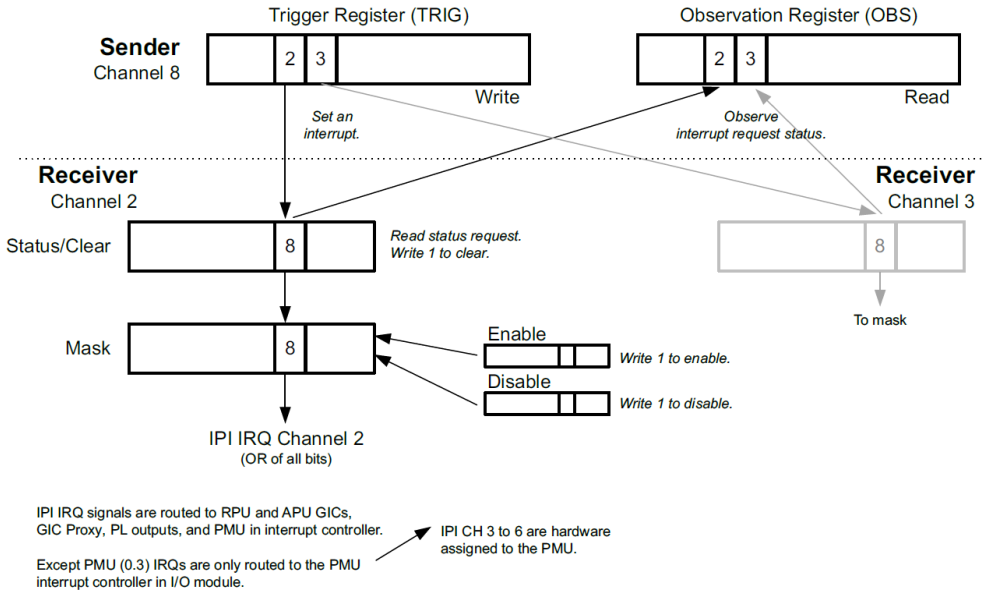

Figure 3.19: Sender–receiver IPI interrupt functions, showing TRIG/OBS on the sender side and ISR/IMR/IER/IDR on the receiver side [3, Chapter 13, p. 314].

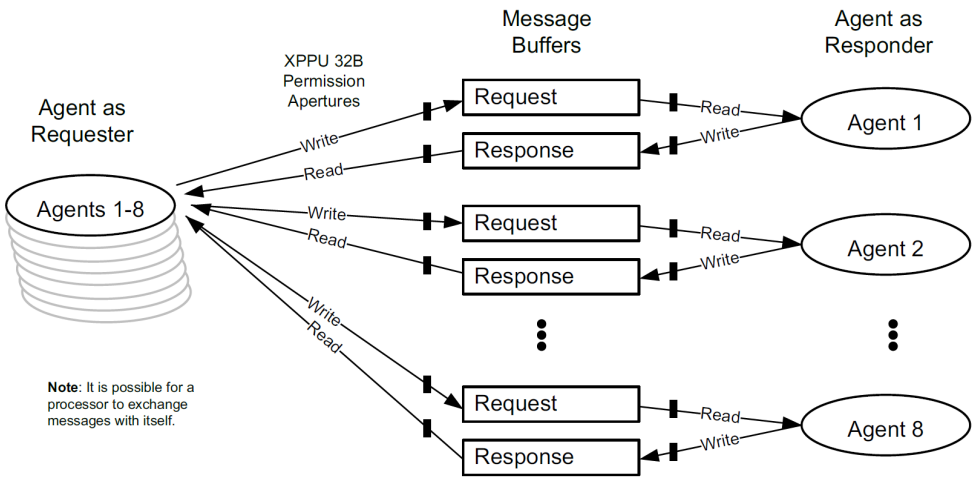

Figure 3.18: IPI Message passing Architecture [3, Chapter 13, p. 317].

All non-PMU IPI channel IRQ signals (channels 0–2 and 7–10) are broadcast to four destinations: the RPU GIC, the APU GIC, the GIC proxy feeding the PMU, and four PL IRQ outputs. Each processor is responsible for masking unwanted IPIs in its own GIC, so an agent can ignore channels or senders that are irrelevant to its software. In contrast, the four PMU IPI IRQs (driven by PMU_0–PMU_3) are only connected to the PMU interrupt controller and never reach the APU or RPU GICs.

Access to both the IPI interrupt registers and the message buffers is controlled by the XPPU protection unit. Each channel's six registers occupy a 64 KB aperture; the XPPU grants read/write permission for that aperture to exactly one master, ensuring that each channel register set has a unique owner. Likewise, the 128 message buffers are divided into 32-byte apertures, and XPPU permissions ensure that only the intended agent can access each buffer, preventing accidental or malicious cross-talk between subsystems [3].

Message Exchange Sequence

A complete IPI transaction between two agents uses both the message buffers and the interrupt registers. A typical send sequence is as follows:

1. The initiating agent writes a 32-byte request structure into the request buffer assigned to the target agent in the appropriate message-buffer set.
2. It writes a 1 into the bit corresponding to the target in its TRIG register on the selected channel, asserting the IPI.
3. Optionally, it polls its OBS register until the bit is seen as set, confirming that the target's ISR has latched the request.
4. It then either waits for a response IPI from the target or polls OBS until the bit clears, according to the agreed software protocol between sender and receiver.

On the receive side, the target agent prepares to handle IPIs by enabling the relevant sender bits in its IER register and enabling the corresponding channel IRQ in its local interrupt controller (GIC or PMU interrupt controller). When an IPI arrives:

1. The target's IRQ handler reads ISR and IMR to determine which sender and channel caused the interrupt.

2. It reads the 32-byte request from the appropriate message buffer, processes the request, and optionally writes a 32-byte response into the paired response buffer.

3. It signals completion either by clearing the ISR bit (writing back the value that was read) or by issuing a return IPI to the original sender, depending on the pre-defined protocol.

For the PMU specifically, all four IPI interrupts share a single PMU-dedicated buffer set, so the PMU firmware typically encodes both a target module identifier and an API identifier into the first word of each message and uses an IPI manager layer to demultiplex incoming IPIs to the appropriate module handler. This design allows the PMU to act as a central coordination point, while the IPI hardware provides a uniform, low-latency mechanism for all agents to generate, observe, and clear interrupt-driven messages across the device [3].

PMU Error Handling and R5 Notification Flow

The IPI infrastructure described above provides the transport mechanism for messages between processors. On top of this, the PMU firmware implements a structured error-handling flow that turns raw hardware error signals into meaningful notifications for the Cortex-R5. In the case of an error happening in the OCM , meaning once an OCM ECC event has raised an interrupt into the PMU and the relevant status has been latched, the error management logic decides what to do with that information. And when required it forwards it to the R5 over an IPI.

The PMU continuously monitors a wide set of system error inputs routed from various PS blocks, including the OCM controller's ECC error outputs. Error status is captured in the `PMU_GLOBAL.ERROR_STATUS_1` and `ERROR_STATUS_2` registers, which act as central latches for error events. Each error source can be individually enabled or masked using the `ERROR_EN_1/2` and `ERROR_INT_MASK_1/2` registers.

When an enabled, unmasked error occurs, the PMU's internal interrupt controller asserts an interrupt to the triple-redundant MicroBlaze processor, which in turn executes the firmware defined handlers that are placed in the Error Management (EM) module. For OCM ECC errors, the handler reads the OCM-specific status registers to obtain the fault address, error type (correctable versus uncorrectable) and syndrome or error code and then decides whether the event should be escalated to the software running on the Cortex-R5 via the IPI mechanism. This setup lets the PMU collect all error events in one place and forward only the important ones to other processors, while the IPI mechanism provides a fast way to deliver those notifications.

At the hardware level, the OCM ECC block asserts a dedicated system interrupt whenever it detects a correctable or uncorrectable error. This interrupt is one of the shared peripheral interrupts (IRQ 42) and is routed, via the GIC proxy into the PMU interrupt controller. This is how OCM errors can be handled even when the application processors are powered down. When this interrupt fires:

1. The corresponding bit in the PMU error status registers is set, indicating that an OCM error has occurred.

2. The PMU firmware reads the OCM controller's own status registers to capture detailed information such as error address and ECC syndrome.

3. The error is classified by severity and mapped into either `ERROR_STATUS_1` or `ERROR_STATUS_2`, depending on whether it is considered lower or higher priority. Error status 1 captures lower priority errors such as software-level errors, timeout errors while Error status 2 captures higher priority errors such as PLL lock failures, watchdog events and uncorrectable memory errors [3, 50].

This capture step guarantees that every OCM ECC event is recorded in a well defined set of PMU registers, so the firmware can always read back the same information for logging, reporting, or recovery.

Once the PMU detects an OCM error via interrupt, the PMU firmware's Error Management (EM) module is invoked. The EM module is responsible for deciding what action should be taken in response to each

hardware error source, including OCM ECC events. For every error type, the firmware configuration selects one or more actions that determine whether the PMU will:

- Generate a system reset (SRST)
- Generate a power-on reset (POR)
- Assert the PS_ERROR_OUT signal to external systems
- Trigger an interrupt to the PMU processor itself, invoking a custom error handler
- Send a notification to another processor (for example, the Cortex-R5) using an IPI message

When the configuration for the OCM ECC error includes the last option, the custom PMU error handler builds a 32-byte error report and sends it to the R5 using the PMU's outbound IPI channel and message buffer, following the steps described below:

1. **Word 0 (Header):** Encodes the target module ID on the R5 side and an API or event ID, allowing fast demultiplexing in the R5 IPI handler.
2. **Word 1 (Primary payload):** Contains the error type and high-level status flags.
3. **Words 2–5 (Extended payload):** Carry the error address, ECC syndrome or error code, and optional diagnostic fields if needed.
4. **Word 6 (Reserved):** Reserved for future expansion.
5. **Word 7 (Optional checksum):** Can be used as a checksum in safety-critical configurations [4].

After the PMU writes this message and triggers the IPI:

1. The R5's GIC records the IPI interrupt and vectors execution to the R5's IPI ISR.
2. The ISR reads the IPI status (ISR) to confirm that the PMU is the sender and then reads the 32-byte message from the associated message buffer.
3. Using the header (module ID and API ID), the ISR dispatches the event to the correct R5-side handler.
4. The handler interprets the error payload, logs the event, and executes the appropriate recovery action (for example, memory scrubbing, isolating a faulty region, or escalating to a higher-level fault manager).
5. Optionally, the R5 writes an acknowledgement or follow-up information into its own request buffer and triggers a return IPI back to the PMU.

This bidirectional, step-by-step flow completes the chain from hardware ECC detection, through centralized PMU error management, to application-level handling on the R5, all built on the IPI communication infrastructure described in the previous subsection [3, 4].

4. Experimental Setup and Methodology

The effectiveness of fault tolerance mechanisms integrated into modern heterogeneous System-on-Chip platforms cannot be fully understood through specification level analysis or theoretical modeling alone. While vendor documentation describes the intended operation of features such as lockstep execution and Error Correcting Code (ECC) memory protection, their actual behavior under controlled fault conditions requires direct experimental validation on real hardware. Fault injection provides a means to deliberately introduce faults at precise locations and times, enabling systematic observation of detection mechanisms, error propagation paths, and system-level responses.

This chapter describes the hardware platform and software development environment used throughout the experimental work. The choice of development board, toolchain configuration, and runtime environment directly influences the scope and repeatability of fault injection experiments. Understanding these foundational elements is essential for interpreting the results presented in later chapters and for assessing the generalizability of findings to other UltraScale+ MPSoC-based systems.

4.1 Development Tools

The effectiveness of fault tolerance mechanisms integrated into modern heterogeneous System-on-Chip platforms cannot be fully understood through specification-level analysis or theoretical modeling alone. While vendor documentation describes the intended operation of features such as lockstep execution and Error Correcting Code (ECC) memory protection, their actual behavior under controlled fault conditions requires direct experimental validation on real hardware. Fault injection provides a means to deliberately introduce faults at precise locations and times, enabling systematic observation of detection mechanisms, error propagation paths, and system-level responses.

To understand how well the fault tolerance mechanisms in modern System-on-Chip devices actually perform in practice, we must test them under real conditions. While manufacturer documentation describes what these features are supposed to do, direct experimentation on actual hardware is the only way to verify their effectiveness. This thesis presents a controlled experimental investigation of the fault tolerance capabilities integrated into the Zynq UltraScale+ MPSoC found on the Ultra96-V2 board. By deliberately introducing faults at specific locations and measuring how the system detects and responds to them, we can gain insights into both the strengths and limitations of vendor-provided protection mechanisms. The faults we inject are carefully chosen to be representative of real hardware failures, and the experiments are designed to be repeatable so that results can be verified and compared across multiple test runs.

4.1.1 Ultra96-V2 Development Board

The experimental work presented in this thesis is conducted on the Avnet Ultra96-V2 development board, which hosts the AMD Xilinx Zynq UltraScale+ MPSoC device. The Ultra96-V2 board has been made available by the Hardware Lab of the University of Piraeus Research Center for the purposes of this research. This board provides access to the fault tolerance mechanisms under investigation, including the dual-core Cortex-R5 Real-Time Processing Unit with lockstep capability, ECC-protected On-Chip Memory and a Platform Management Unit.

Figure 4.1 shows the physical Ultra96-V2 board, highlighting the key components relevant to this study. The board conforms to the 96Boards Consumer Edition specification, providing standardized expansion headers and a compact 85 mm $\times$ 54 mm form factor suitable for both prototyping and embedded deployment [29, 43].

The Zynq UltraScale+ MPSoC integrates a heterogeneous multiprocessor system comprising a quad-

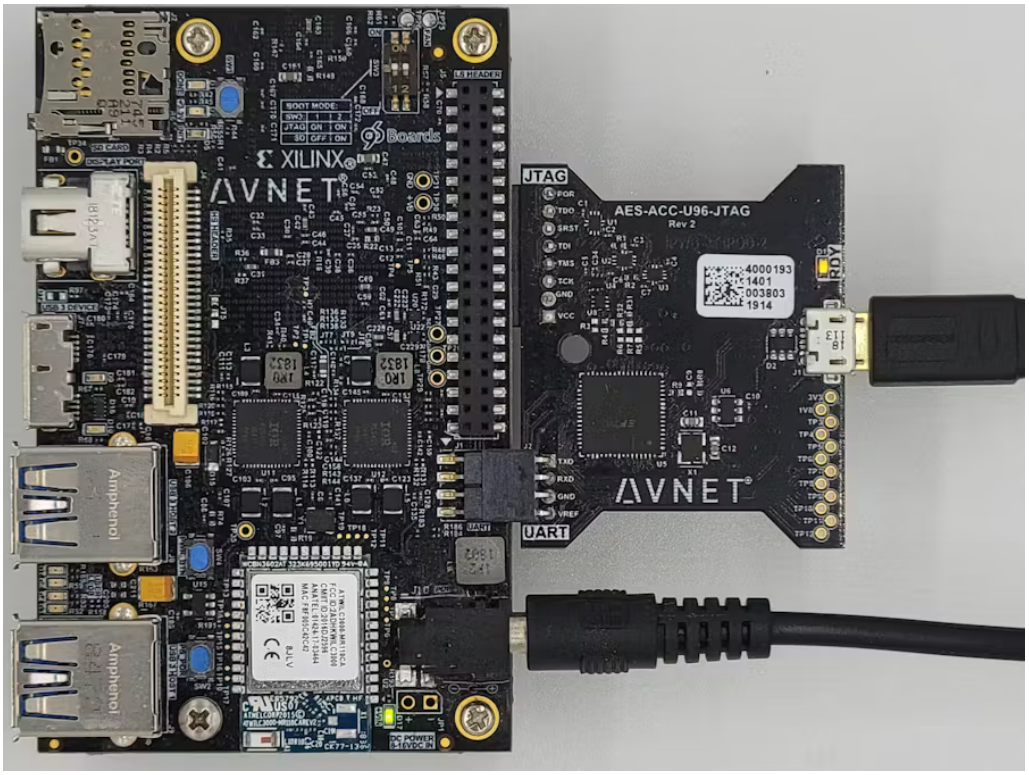

Figure 4.1: Avnet Ultra96-V2 development board based on Zynq UltraScale+ MPSoC ZU3EG device. The board integrates 2 GB LPDDR4 memory, microSD boot storage, USB 3.0 connectivity, Mini DisplayPort output, Wi-Fi/Bluetooth module, and 96Boards-compliant expansion headers. JTAG and UART debug interfaces enable real-time system monitoring during fault injection experiments [43].

core Arm Cortex-A53 Application Processing Unit operating at up to 1.5 GHz, a dual-core Arm Cortex-R5F Real-Time Processing Unit operating at up to 600 MHz in lockstep mode, and 154,000 logic cells of programmable logic fabric [53, 43]. The system memory consists of 2 GB of Micron LPDDR4 SDRAM in a 512M $\times$ 32 configuration, complemented by 256 KB of ECC-protected on-chip memory with both correctable and uncorrectable error detection capabilities [3, 43].

Table 4.1 summarizes the key specifications of the Ultra96-V2 platform that are directly relevant to the fault injection experiments conducted in this study.

The board provides two boot modes controlled by a DIP switch (Boot Mode selector) located on the Ultra96-V2 board. The first mode boots from the microSD card, which stores the First Stage Boot Loader (FSBL), PMU firmware, and application binaries. This is the primary boot method used during normal development, enabling straightforward firmware deployment through simple file replacement on the microSD card, followed by a power cycle [4, 43]. The second mode boots via JTAG, which allows direct loading of firmware from a host workstation through the JTAG interface without requiring the microSD card. The JTAG boot mode is useful for rapid development cycles and debugging scenarios where quick firmware updates are needed [43].

Debug and system monitoring access is provided through interfaces connected to a host workstation. The UART header connects via a USB-to-serial adapter, enabling real-time console logging during fault injection experiments at 115200 baud rate. The JTAG header provides low-level hardware debugging capabilities for register inspection, memory access, and direct firmware loading. These interfaces work together to provide comprehensive observability into system behavior during experiments.

Table 4.1: Ultra96-V2 key specifications for fault injection experiments [43, 53].

Component	Specification
MPSoC Device	Xilinx Zynq UltraScale+
APU	Quad-core Arm Cortex-A53 @ 1.5 GHz
RPU	Dual-core Arm Cortex-R5F @ 600 MHz (lockstep mode)
GPU	Arm Mali-400 MP2
Programmable Logic	154K logic cells, 141K flip-flops, 70K LUTs
System Memory	2 GB Micron LPDDR4 (512M × 32) @ 533 MHz
On-Chip Memory	256 KB OCM with ECC protection
Boot Storage	MicroSD slot (32 GB)
Debug Interfaces	UART console (115200 baud), JTAG header
Expansion	40-pin low-speed header, 60-pin high-speed header
Power Supply	12 V @ 2 A (barrel jack)
Operating Temperature	Commercial (0–70°C) and Industrial (-40–85°C)
Form Factor	85 mm × 54 mm (96Boards CE compliant)

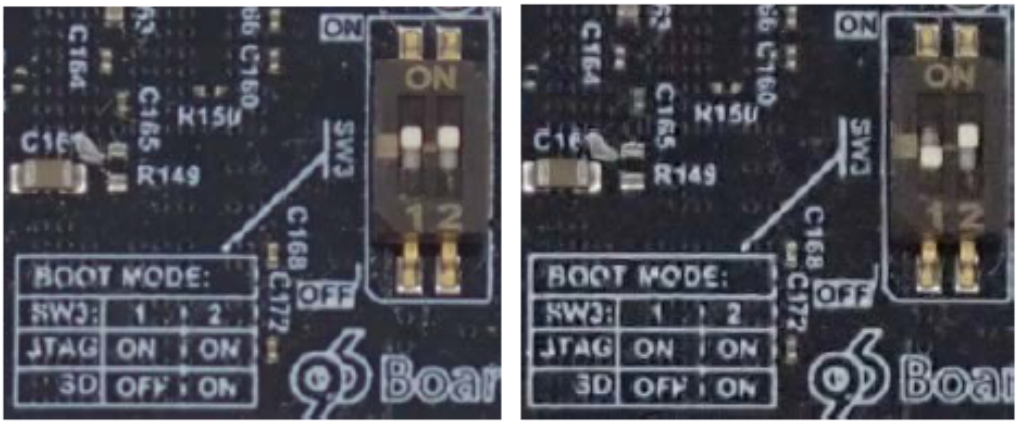

Figure 4.2: Ultra96-V2 Boot Mode DIP Switch (Boot Mode selector). Left position selects JTAG boot mode (for JTAG-based firmware loading from host), and right position selects SD card boot mode (for microSD-based boot) [43, Chapter 6, p. 49].

4.1.2 Vivado Design Suite and Vitis 2022.2

The development, synthesis, and implementation of hardware platforms and embedded software for the Zynq UltraScale+ MPSoC require a coordinated toolchain that integrates FPGA fabric configuration, processing system setup, and embedded software development. This work utilizes the AMD Xilinx Vivado Design Suite version 2022.2 for hardware platform definition and the AMD Xilinx Vitis Unified Software Platform version 2022.2 for embedded software development, compilation, and debugging [45, 44].

Vivado Design Suite 2022.2

For this study, Vivado is used to create a minimal hardware platform definition that configures the Zynq Ultrascale+ processing system .The Vivado Design Suite provides an IP Integrator interface for configuring the Zynq UltraScale+ MPSoC. The configuration process involves selecting enabled processing elements and defining inter-processor communication pathways. Key configuration steps include the following:

- **Processing System Selection:** The PS UltraScale+ IP core is instantiated and configured with the required processor cores (APU, RPU, PMU) enabled. Both Cortex-R5 cores are enabled to participate

in the experimental workloads [54].

- **Inter-Processor Interrupt (IPI) Mapping:** The IPI Master Mapping configuration defines which processor cores can send and receive interrupt messages. For this study, the IPI mapping is configured to route messages between the RPU (GEN_IPI_1 and GEN_IPI_2) and the PMU (GEN_IPI_PMU), enabling the PMU firmware to send error notifications to the Cortex-R5 application via dedicated inter-processor interrupt channels [4, 54]. This configuration is critical for the fault detection and error reporting mechanisms evaluated in the experiments.
- **Clock and Reset Configuration:** Clock frequencies and reset signal routing are configured to match the target operational requirements. The Cortex-R5 processors are configured to run at 600 MHz in lockstep mode, with synchronized clock distribution and reset assertion [3, 54].

The hardware platform is defined using the Zynq UltraScale+ MPSoC IP core configuration wizard within Vivado's IP Integrator, which generates a block design specifying the enabled processor cores, memory interfaces, and I/O peripheral assignments [54]. Once the block design is complete, Vivado synthesizes the hardware platform and exports it as an XSA (Xilinx Support Archive) file, which encapsulates all hardware-specific parameters including base addresses, interrupt assignments, and clock frequencies [44]. This XSA file is subsequently imported into Vitis to generate the Board Support Package (BSP) and configure linker scripts for software projects, ensuring consistency between hardware configuration and software expectations.

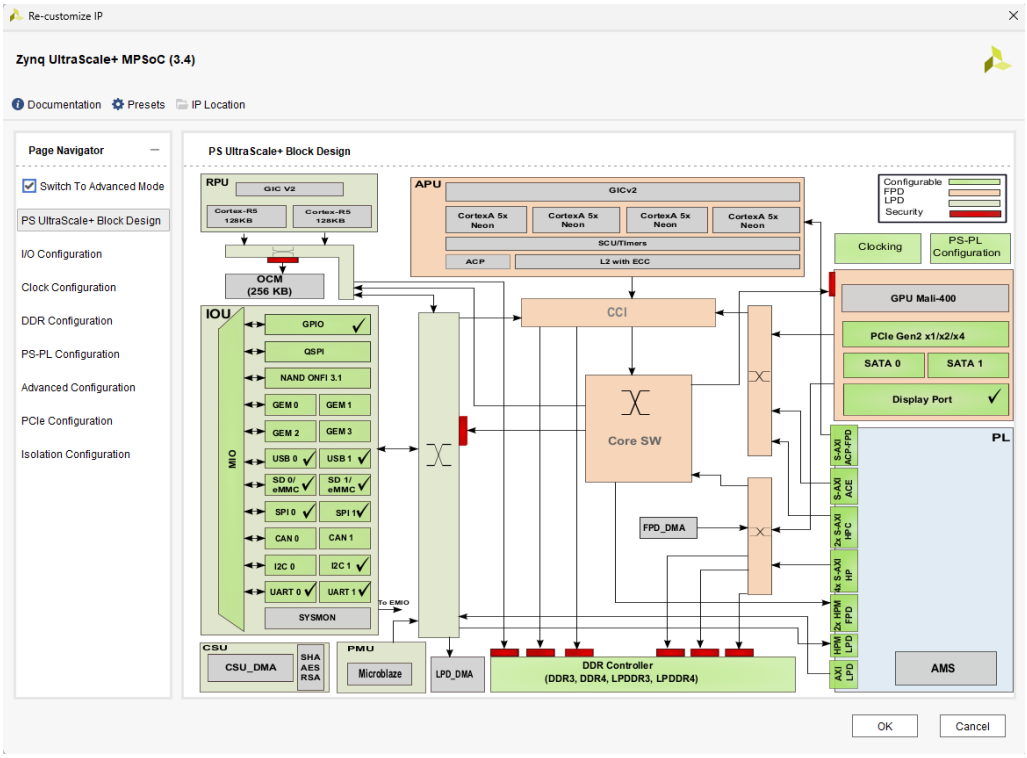

Figure 4.3: Processign System UltraScale + Block Design

Figure 4.3 shows the Zynq UltraScale+ block design generated by Vivado, illustrating the major processing elements (RPU with dual Cortex-R5 cores, APU with quad Cortex-A53 cores, GPU, and PMU) . Figure 4.4 shows the IPI Master Mapping configuration interface, where the inter-processor interrupt channels between the RPU and PMU are explicitly defined.

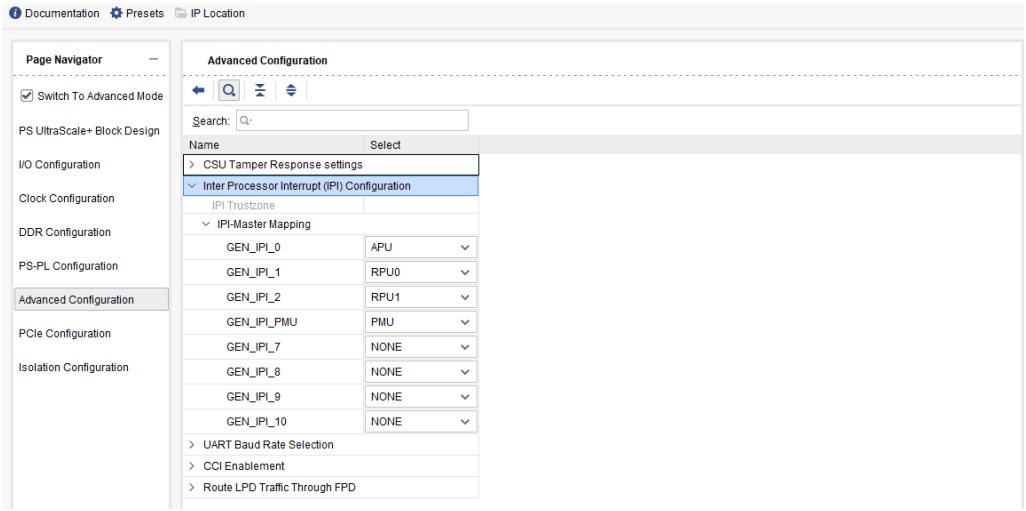

Figure 4.4: InterProcessor Interrupt Configuration

Vitis Unified Software Platform 2022.2

The Vitis Unified Software Platform provides an integrated development environment built on the Eclipse framework, offering project management, source code editing, cross-compilation, debugging, and deployment capabilities for both bare-metal and operating-system-based applications targeting Zynq UltraScale+ devices. Vitis integrates the Arm GNU Toolchain (GCC version 12.2.0) for cross-compilation of C and C++ code to the ARMv7-R instruction set architecture used by the Cortex-R5 processor [9, 44].

For this work, Vitis is configured to target the ZU3EG device and the Ultra96-V2 board support package, which includes pre-configured linker scripts, startup code, and peripheral drivers specific to the hardware platform exported from Vivado. The linker scripts define memory regions for the RPU, mapping code and data sections to appropriate physical address ranges such as the on-chip memory (OCM) or DDR memory. Modifications to these linker scripts are necessary to ensure that fault injection workloads execute entirely within OCM when validating ECC behavior, as the OCM region (`0xFFFC0000`–`0xFFFFFFFF`) provides ECC protection while DDR memory does not [4, 44].

Vitis provides the Board Support Package (BSP) generation utility, which automatically configures low-level drivers (such as `xuartps` for UART communication, `xipipsu` for inter-processor interrupts, and `xscugic` for the Generic Interrupt Controller) based on the hardware platform exported from Vivado. For bare-metal applications, the standalone BSP is used, which includes minimal runtime initialization code without operating system services. For FreeRTOS-based applications, the FreeRTOS BSP is selected, which integrates the FreeRTOS kernel version 10.4.3 with hardware-specific port layers for the Cortex-R5 processor [2].

4.1.3 FreeRTOS Real-Time Operating System

FreeRTOS is an open-source, real-time operating system kernel designed for microcontrollers and small microprocessors. Originally developed by Richard Barry and now maintained under the stewardship of Amazon Web Services, FreeRTOS has become a de facto standard in the embedded industry due to its small memory footprint, ease of integration, and permissive MIT open-source license.

FreeRTOS provides core real-time operating system functionality including preemptive task scheduling, inter-task communication primitives (queues, semaphores, mutexes, event groups), software timers, and memory management with multiple heap allocation schemes. The kernel is designed to be portable across a wide range of microcontroller architectures, with over 40 officially supported ports including the Arm

Cortex-R5 processor used in this study [35, 1, 2].

For this work, FreeRTOS version 10.4.3 is integrated into the Cortex-R5 application to provide a representative real-time software environment in which fault injection experiments are conducted. The use of FreeRTOS enables evaluation of fault detection and handling mechanisms in the context of multitasking workloads with concurrent task execution, preemptive scheduling, and inter-task synchronization, conditions that are typical of production embedded systems [17].

The FreeRTOS configuration for the Cortex-R5 application in this study includes the following key settings:

- **Scheduling:** Preemptive scheduling with time-slicing enabled, allowing multiple tasks to execute with periodic context switching based on task priorities and a configurable tick rate of 1000 Hz (1 ms tick period).
- **Memory Management:** Dynamic memory allocation using heap_4, which provides deterministic allocation and deallocation with memory coalescence to reduce fragmentation [2].
- **Inter-Task Communication:** Task notification mechanisms and queues are used to coordinate between periodic computational tasks and fault injection trigger tasks.

The integration of FreeRTOS into the fault injection experimental framework provides several advantages. First, it enables testing of lockstep and ECC fault detection mechanisms under realistic multitasking conditions where task preemption and context switching may interact with fault propagation. Second, it allows evaluation of error handling strategies that must preserve RTOS scheduler integrity and task state consistency after fault detection. Third, it facilitates modular organization of the experimental code, with separate tasks for workload execution, fault injection triggering, and error event logging.

4.2 Test Cases and Scenarios

The experimental evaluation of fault tolerance mechanisms in the Zynq UltraScale+ MPSoC is organized into three primary test cases, each designed to exercise specific hardware fault detection and handling features under controlled conditions. The test cases focus on the ECC protection mechanisms of the On-Chip Memory (OCM) and the lockstep execution mechanism of the Cortex-R5 Real-Time Processing Unit.

The experimental framework consists of three main software components running on the Ultra96-V2 platform. On the Cortex-R5 processor (R5_0), a FreeRTOS-based application executes a periodic background workload which is a simple heartbeat task that prints status messages every two seconds, or it could be a more computationally intensive application such as matrix multiplication. The specific workload is not critical to the fault injection experiments. What matters is that the processor is actively executing code when faults are injected. On the Platform Management Unit (PMU), custom firmware monitors hardware error signals from the OCM controller and RPU subsystem. When a fault is detected, the PMU firmware executes a custom error handler (`OcmEccErrHandler`) that captures diagnostic information and sends an Inter-Processor Interrupt (IPI) message to the Cortex-R5 application. The R5 application receives the IPI message, decodes the error details, and logs the event to the UART console for analysis.

Faults can be injected through two methods. The first method uses programmatic fault injection functions embedded directly in the R5 application code, allowing faults to be triggered automatically at predetermined points during execution (e.g., after a specific number of heartbeat cycles or matrix operations). The second method uses external fault injection via the Xilinx System Debugger (XSDB) console, where Tcl scripts write directly to memory or control registers to inject faults while the application is running.

In following section each test case is described , including the fault injection mechanism, the expected system behavior, and the observation method used to verify correct operation of the fault detection and handling infrastructure.

4.2.1 Correctable Error (CE) Fault Injection in OCM

Objective: The objective of this test case is to validate that the OCM ECC logic correctly detects and automatically corrects single-bit errors without software intervention, and to confirm that the PMU firmware can capture and report correctable error events to the Cortex-R5 application via IPI messaging.

Fault Injection Method: A single-bit error is introduced into a 32-bit data word stored in the OCM address space (`0xFFFC0000`–`0xFFFFFFFF`). The fault can be injected using either of two methods:

- **Programmatic Injection (C Function):** The R5 application includes a fault injection function that writes a known data pattern to a target OCM address, deliberately flips a single bit in the stored word by direct memory manipulation, and then reads the corrupted word back. When the corrupted data is read, the OCM ECC checker hardware detects the single-bit error, corrects it transparently, and sets the correctable error (CE) flag in the OCM interrupt status register (`OCM_ISR` at address `0xFF960004`).
- **External Injection (XSDB Tcl Script):** A Tcl script executed from the XSDB console writes directly to an OCM memory location to flip a single bit, then triggers a read operation. This method allows fault injection without modifying the application code, enabling interactive testing and validation during development.

When an OCM ECC error is detected, the OCM controller raises an interrupt to the PMU. The PMU handler reads `PMU_GLOBAL_ERROR_STATUS_1` and uses the `OCM_ECC_MASK` bit to identify the OCM ECC source, then calls the custom handler `OcmEccErrHandler` to process the event.

1. Reads the OCM interrupt status register (`OCM_ISR`) to determine whether the error is correctable (CE) or uncorrectable (UE).
2. If the CE flag (bit 6) is set, reads the CE First Failing Address register (`OCM_CE_FFA` at address `0xFF960040`) and the CE First Failing ECC register (`OCM_CE_FFE` at address `0xFF960044`) to capture the location and syndrome of the corrected error.
3. Constructs a 4-word IPI message payload containing the error details: Word 0 contains the API identifier (`0x0001`) indicating an OCM ECC event, Word 1 contains the `OCM_ISR` snapshot, Word 2 contains the CE first failing address, and Word 3 contains the UE first failing address (zero if not applicable).
4. Sends the IPI message to the Cortex-R5 (R5_0) using the `XPfw_IpiWriteMessage` and `XPfw_IpiTrigger` API calls from the PMU IPI manager.
5. Clears the OCM interrupt status bits by writing the read value back to `OCM_ISR` (write-1-to-clear semantics).

Expected Behavior: The ECC checker detects the single-bit error and corrects it before the faulty data is forwarded to the processor. The corrected data is returned to the R5 application, and execution continues without disruption. The OCM controller sets the correctable error (CE) flag in the `OCM_ISR` register and generates an interrupt to the PMU. The PMU executes the `OcmEccErrHandler`, captures the error details, and sends an IPI notification to the R5 application. The R5 application receives the IPI message in its interrupt handler, decodes the message payload, and logs the error information to the UART console. The logged information includes the API identifier, the OCM ISR value, and the CE first failing address, confirming that the error was detected, corrected, and properly reported.

Observation Method: The R5 application monitors incoming IPI messages via its interrupt handler. When an IPI is received, the handler reads the 4-word message buffer and checks if the API identifier (Word 0) matches the OCM ECC event code (`0x0001`). If a match is found, the handler extracts the `OCM_ISR` value (Word 1), the CE first failing address (Word 2), and prints these values to the UART console along with a timestamp or heartbeat counter. The UART console log provides a complete record of the fault injection event, including when the fault was injected, the memory address affected, and confirmation that the error

was corrected by the ECC logic. Post-experiment analysis of the UART log confirms that the CE detection, correction, and notification pipeline operated correctly.

Table 4.2: Test Case 1: OCM correctable error (CE) fault injection parameters and expected outcomes.

Parameter	Value / Description
Fault Type	Single-bit flip in OCM data word
Injection Address Range	OCM region (`0xFFFC0000`–`0xFFFFFFFF`)
Injection Methods	Programmatic (C function) or external (XSDB Tcl script)
Expected Detection	Correctable error (CE) flag (bit 6) set in `OCM_ISR` register
Expected Correction	ECC logic returns corrected data to Cortex-R5 pipeline
PMU Action	`OcmEccErrHandler` reads error registers, constructs IPI message, sends to R5_0
R5 Action	`IpiInterruptHandler` receives IPI, decodes message, logs to UART
System Impact	No application-level disruption; error transparently corrected
IPI Message Format	Word 0: API ID (`0x0001`); Word 1: `OCM_ISR`; Word 2: CE address; Word 3: UE address (0)
Observation	Monitor UART log for IPI reception, verify CE flag in logged ISR value, confirm corrected address

4.2.2 Uncorrectable Error (UE) Fault Injection in OCM

Objective: The objective of this test case is to validate that the OCM ECC logic correctly detects multi-bit errors that exceed the correction capability of the Single Error Correction Double Error Detection (SECDED) ECC code, and to confirm that uncorrectable error events trigger the custom PMU error handler and result in IPI notification to the Cortex-R5 application.

Fault Injection Method: A multi-bit error (two or more bit flips) is introduced into a 32-bit data word stored in the OCM address space. As with the correctable error test case, the fault can be injected using either programmatic C functions or external XSDB Tcl scripts:

- **Programmatic Injection (C Function):** The R5 application includes a fault injection function that writes a known data pattern to a target OCM address, flips two or more bits in the stored word, and then reads the corrupted word back. When the corrupted data is read, the OCM ECC checker hardware detects that the error exceeds the correction capability of the SECDED code and sets the uncorrectable error (UE) flag in the OCM interrupt status register (`OCM_ISR` bit 7).
- **External Injection (XSDB Tcl Script):** A Tcl script executed from the XSDB console writes directly to an OCM memory location to flip multiple bits, creating an uncorrectable error pattern.

When the uncorrectable error is detected, the OCM controller generates an interrupt (IRQ 10) to the PMU. The PMU interrupt handler identifies the OCM ECC error source and dispatches control to the `OcmEccErrHandler`, which performs the following actions:

1. Reads the OCM interrupt status register (`OCM_ISR`) and checks if the UE flag (bit 7) is set.
2. If the UE flag is set, reads the UE First Failing Address register (`OCM_UE_FFA` at address `0xFF960048`), the UE First Failing Data registers (`OCM_UE_FFD0` at address `0xFF960060` and `OCM_UE_FFD1` at address `0xFF960064`), and optionally the UE_RMW flag (bit 10) to determine if the error occurred during a sub with write operation.

3. Constructs a 4-word IPI message payload with the same format as the CE test case: Word 0 contains the API identifier (0x0001), Word 1 contains the OCM_ISR snapshot (with bit 7 set for UE), Word 2 is zero (no CE address), and Word 3 contains the UE first failing address.

4. Sends the IPI message to the Cortex-R5 and waits for acknowledgment using the poll function with a 1-second timeout.

5. Clears the OCM interrupt status bits by writing the read value back to OCM_ISR.

Expected Behavior: The ECC checker detects the multi-bit uncorrectable error and asserts the UE flag in the OCM_ISR register. Unlike correctable errors, uncorrectable errors cannot be transparently corrected, so the corrupted data may propagate to the processor depending on the timing and error handling configuration. However, the critical outcome is that the error is detected and reported. The OCM controller generates an interrupt to the PMU, which executes the OcmEccErrHandler, captures the error details including the failing address and corrupted data values, and sends an IPI notification to the R5 application. The R5 application receives the IPI message, decodes the UE-specific information (bit 7 set in Word 1, non-zero UE address in Word 3), and logs the uncorrectable error event to the UART console. The system behavior after UE detection depends on the application-level error handling policy—options include logging the error for diagnostic purposes, initiating a graceful shutdown, or triggering a system reset.

Observation Method: The R5 application IPI interrupt handler receives the error notification message and examines the OCM_ISR value (Word 1) to distinguish between correctable and uncorrectable errors. If bit 7 is set, the handler identifies the event as an uncorrectable error and logs the UE first failing address (Word 3) and corrupted data values to the UART console. The UART log provides a complete record of the uncorrectable error event, including the memory address that experienced the multi-bit fault, the data values that were corrupted, and the timestamp of the event. Post-experiment analysis confirms that the PMU-to-R5 error notification pipeline operated correctly and that the uncorrectable error was detected and reported even though it could not be corrected.

Table 4.3: Test Case 2: OCM uncorrectable error (UE) fault injection parameters and expected outcomes.

Parameter	Value / Description
Fault Type	Multi-bit flip (2+ bits) in OCM data word
Injection Address Range	OCM region (0xFFFC0000–0xFFFFFFFF)
Injection Methods	Programmatic (C function) or external (XSDB Tcl script)
Expected Detection	Uncorrectable error (UE) flag (bit 7) set in OCM_ISR register
Error Registers Read	OCM_UE_FFA, OCM_UE_FFD0, OCM_UE_FFD1
PMU Action	OcmEccErrHandler captures UE details, constructs IPI message, sends to R5_0 with ACK wait
R5 Action	IpiInterruptHandler receives IPI, identifies UE (bit 7 set), logs UE address and data
System Impact	UE interrupt to PMU; IPI notification to Cortex-R5; no transparent correction
IPI Message Format	Word 0: API ID (0x0001); Word 1: OCM_ISR (bit 7 set); Word 2: 0; Word 3: UE address
Observation	Monitor UART log for IPI reception, verify UE flag (bit 7) in logged ISR value, confirm UE address

4.2.3 Lockstep Mismatch Fault Injection via Register

Objective: The objective of this test case is to validate that the Cortex-R5 lockstep comparison logic correctly detects execution divergence between the two redundant cores and generates an appropriate fault signal

or exception. .

Fault Injection Method: The Zynq UltraScale+ MPSoC provides a memory mapped fault injection register (`RPU_ERR_INJ` at base address `0xFF9A0020`) that allows through software to deliberately introduce a state mismatch between the two lockstep Cortex-R5 cores for validation and testing purposes. The fault injection is performed programmatically from within the R5 application:

- The R5 application includes a fault injection task or timer callback that writes a specific value (e.g., `0x00000001`) to the `RPU_ERR_INJ` register at a predetermined time (e.g., after a certain number of heartbeat cycles or matrix computation iterations).
- Writing to the `RPU_ERR_INJ` register causes one of the two lockstep cores to experience an internal state corruption, leading to a mismatch in outputs that is detected by the hardware comparison logic.
- Before triggering the fault, the error injection enable register (`RPU_ERR_EN_1` at address `0xFFD805A0`) may need to be configured to enable the fault injection mechanism. After the fault is triggered, the `RPU_ERR_INJ` register is cleared by writing `0x00000000`.

This type of Faults must be injected while both cores are actively executing instructions in order for a mismatch to be detected. For this reason, the fault injection is typically triggered during the execution of a computational workload task rather than during idle periods.

Expected Behavior: The lockstep comparator hardware acts as to have detected a mismatch between the two Cortex-R5 cores and asserts an error and asserts a lockstep error signal toward the PMU error management logic. In the default Xilinx reference flow this signal can be configured to trigger resets or other recovery actions. In this work, the PMU Error Manager is configured to associate the RPU lockstep error ID `(EM_ERR_ID_RPU_LS)` with a custom error action that invokes the `RpuLsHandler` callback. The `RpuLsHandler` function logs the occurrence of the lockstep error on the PMU console and does not attempt any recovery, reset.

Observation Method: The custom exception handler on the R5 logs diagnostic information to the UART console upon detecting a lockstep mismatch exception. The logged information includes the exception type (e.g., data abort, prefetch abort, or undefined instruction). The lockstep mismatch triggers a PMU interrupt, the PMU logs the event and sends an IPI notification to the R5, which is logged to the UART console.

The lockstep fault injection test case validates the end to end operation of the redundant execution mechanism, confirming that deliberate state mismatches introduced via the `RPU_ERR_INJ` register are reliably detected by the hardware comparison logic.

Table 4.4: Test Case 3: Lockstep mismatch fault injection via `RPU_ERR_INJ` register parameters and expected outcomes.

Parameter	Value / Description
Fault Type	Lockstep core mismatch via `RPU_ERR_INJ` register write
Injection Register	`RPU_ERR_INJ` at address `0xFF9A0020`
Injection Value	`0x000000FF` (trigger lockstep mismatch event in RPU)
Enable Register	`PMU_GLOBAL.ERROREN1` at address `0xFFD805A0` (enable RPU lockstep error reporting before injection)
Injection Timing	During active execution of the FreeRTOS heartbeat task on the lockstep R5 pair
Injection Methods	Programmatic (write to `RPU_ERR_INJ` from the R5 heartbeat application)
Expected Detection	Lockstep comparator treats the injected condition as a core mismatch and asserts the RPU lockstep error towards the PMU error manager
System Response	PMU Error Manager maps `EM_ERR_ID_RPU_LS` to a custom action and invokes `RpuLsHandler` (no reset, no automatic recovery)
PMU Handler Behavior	`RpuLsHandler` logs the lockstep error ID on the PMU console and returns without modifying RPU state or issuing resets
Observation	UART logs from the PMU firmware showing invocation of `RpuLsHandler` and the reported lockstep error, confirming successful fault injection and detection

4.3 Implementation

The fault injection experiments on the Ultra96-V2 platform bring together hardware design, bootloader configuration, PMU firmware, and R5 application development. Thiw process starts in the Vivado Design Suite, where we configure the Zynq UltraScale+ MPSoC by importing the IP and creating a block diagram. After synthesis and implementation is complete in Vivado we everything as a Xilinx Support Archive (XSA) file. Essentially a ZIP file that contains all the hardware specifications including processor settings, memory layout, interrupt assignments, clock frequencies, and how components are connected. This XSA file acts as a bridge that carries the hardware design information from Vivado to the Vitis software development environment, where we write the actual application code.

When the XSA file is imported into Vitis, it automatically generates the Board Support Package (BSP). The BSP contains the hardware drivers, library configurations, and linker scripts tailored to the target platform. For this study, we configure the BSP to include several Xilinx libraries that are essential for the fault injection framework:

- **`xilffs` (FAT File System):** Provides FAT16 and FAT32 file system support for reading and writing to the microSD card, which stores the First Stage Boot Loader (FSBL), PMU firmware, and application binaries during the boot sequence.
- **`xilpm` (Power Management):** Implements the Embedded Energy Management Interface (EEMI) used by the PMU firmware to manage processor power states, coordinate inter-processor communication via IPI, and respond to system-level events such as hardware error interrupts.
- **`xilsecure` (Secure Boot):** Provides cryptographic functions for secure boot and authentication. Although these security features are not directly used in the current fault injection experiments, they are included in the BSP for completeness and future testing.

The R5 uses FreeRTOS 10.4.6 to schedule the test workload and handle periodic tasks. The software is split between two processors: the Cortex-R5 runs the FreeRTOS application that does the work and prints results and the PMU that runs custom firmware that watches for hardware errors. When a fault is injected the PMU detects it and notifies the R5 application so the event is visible in logs.

4.3.1 Cortex-R5 Application: `ipi_heartbeat_app.c`

The Cortex-R5 application serves as the primary execution environment for computational workloads and the receiver of fault notifications from the PMU. The application is structured around the FreeRTOS real-time operating system, which provides task scheduling, inter-task communication, and interrupt management services. At a high level, the application performs three main functions: (A) it initializes the IPI communication channel to receive messages from the PMU, (B) it executes a periodic heartbeat task that simulates a computational workload and (C) it responds to incoming IPI messages that contain error information about detected faults in the OCM or RPU subsystems. The complete application code is contained in the file `ipi_heartbeat_app.c`, which includes FreeRTOS task definitions, IPI driver initialization, and interrupt handler routines.

Figure 4.5 summarizes the complete error-signaling path used in the OCM ECC experiments. It starts from the execution of the Cortex-R5 application and a fault injection event, then shows how the OCM ECC logic classifies the error as correctable or uncorrectable, how the OCM controller raises System IRQ 42 and propagates it through the GIC proxy to the PMU, and finally how the PMU Error Manager invokes the custom OCM handler and forwards a condensed error report to the R5 application via IPI.

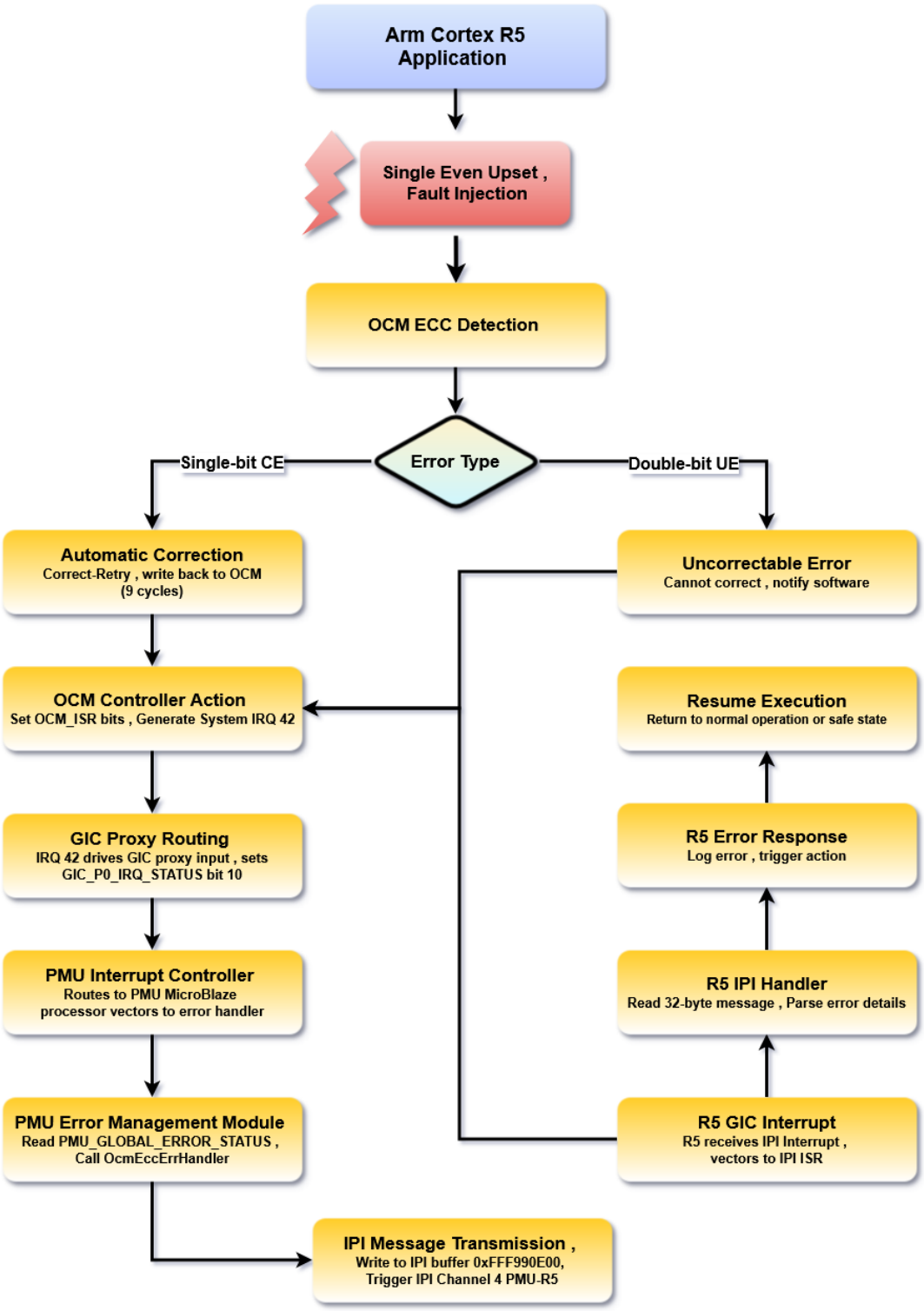

Figure 4.5: End-to-end OCM ECC fault handling and IPI notification flow between the Cortex-R5 application and the PMU firmware.

FreeRTOS Integration and Task Structure

FreeRTOS provides the basic real-time services for the R5 application, allowing multiple tasks to run with preemptive scheduling. The `main()` function performs three key steps before handing control to the kernel:

1. Initializes the IPI driver and registers the IPI interrupt handler with the GIC, so the R5 can receive messages from the PMU.
2. Creates two FreeRTOS tasks: `HeartbeatTask`, which runs the periodic workload and drives the experiments, and `IpiPollTask`, which periodically reads and prints the IPI status for debugging.
3. Initializes the Xilinx exception framework and registers a custom handler for undefined instruction exceptions.

Finally, `vTaskStartScheduler()` starts the FreeRTOS scheduler, which selects and runs the highest-priority ready task.

Once the scheduler is running, `HeartbeatTask` executes in a loop with a 2 second delay between iterations. On each iteration it increments a heartbeat counter, prints a status message over UART, and checks the `ocm_ecc_event_received` flag set by the IPI ISR when the PMU reports an OCM ECC event. If the flag is set, the task prints the full error information (API ID, OCM ISR value, CE address, UE address) and then clears the flag so that the next event can be captured cleanly.

The `IpiPollTask` is a simple support task used mainly during development. It periodically calls `XIpiPsu_GetInterruptStatus()` and prints any non-zero value it observes. This makes it easy to confirm that the IPI hardware path is alive and that messages from the PMU actually show up in the R5 IPI status registers.

Generic Interrupt Controller (GIC) Initialization via FreeRTOS

One of the most critical aspects of the R5 application is how the Generic Interrupt Controller (GIC) is initialized and managed. The interrupt controller in the Zynq UltraScale+ MPSoC routes hardware interrupts to the processor cores. When FreeRTOS starts, it brings up its own interrupt controller driver and uses it to set up interrupt priorities and decides which handler runs when an interrupt arrives. If the application tried to set up a second, separate instance of the interrupt controller (for example by calling `XScuGic\_CfgInitialize()` directly in `main()`), the two configurations would conflict, and interrupts registered on the second instance would never fire because only the FreeRTOS owned instance actually services IRQs. To avoid this problem, the R5 application uses FreeRTOS-specific functions to register interrupt handlers. The `xPortInstallInterruptHandler()` function, provided by the FreeRTOS port layer for ARM Cortex-R5, registers an interrupt handler with the FreeRTOS GIC instance. Similarly, the `vPortEnableInterrupt()` function enables a specific interrupt line in the GIC. By using these FreeRTOS-aware functions instead of directly manipulating the GIC hardware, the application ensures that all interrupt configuration is consistent with the FreeRTOS scheduler and that interrupt handlers execute correctly within the FreeRTOS interrupt context. This approach is essential for reliable IPI interrupt delivery from the PMU to the R5.

IPI Initialization: `IpiInit()` Function

The `IpiInit()` function performs the complete initialization sequence for the Xilinx IPI driver and configures the IPI channel to receive messages from the PMU. The function executes the following steps:

1. **Lookup IPI Configuration:** Calls `XIpiPsu_LookupConfig(IPI_DEVICE_ID)` to retrieve the hardware configuration for the IPI device from the BSP-generated configuration tables. This configuration includes the base address of the IPI controller (`0xFF300000`), the interrupt ID (IRQ 63), and the bit masks for each IPI channel.

2. **Initialize IPI Driver:** Calls `XIpiPsu_CfgInitialize()` to initialize the IPI driver instance with the retrieved configuration. This function sets up internal data structures and resets the IPI hardware to a known state.

3. **Reset IPI Controller:** Calls `XIpiPsu_Reset()` to perform a software reset of the IPI controller, clearing any pending messages or interrupt flags left over from previous operations.

4. **Register Interrupt Handler:** Calls `xPortInstallInterruptHandler(IPI_INT_ID,IpiInterruptHandler,IpiInstPtr)` to register the `IpiInterruptHandler` function as the handler for IPI interrupt requests. This uses the FreeRTOS port helper function to ensure the handler is registered with the FreeRTOS GIC instance.

5. **Enable GIC Interrupt Line:** Calls `vPortEnableInterrupt(IPI_INT_ID)` to enable IRQ 63 in the GIC, allowing IPI interrupts to reach the R5 processor.

6. **Enable IPI Channel:** Calls `XIpiPsu_InterruptEnable` to enable the specific IPI channel that the R5 receives messages on. The bit mask `XPAR_PSU_IPI_4_BIT_MASK` (value `0x00020000`) corresponds to the IPI channel assigned to R5_0.

The IPI initialization is performed in the `main()` function before any tasks are created or the scheduler started. If any step of the initialization fails, the function returns `pdFAIL`, and the application enters an infinite loop to prevent further execution with a non-functional IPI channel.

IPI Interrupt Handler: `IpiInterruptHandler()` Function

The `IpiInterruptHandler()` function is the interrupt service routine that runs on the R5 whenever the PMU sends an IPI message. FreeRTOS calls this handler when the IPI line assigned to the R5 fires. The handler follows a simple sequence:

1. **Read interrupt status:** It first calls `XIpiPsu_GetInterruptStatus()` to read the IPI status register and see which IPI sources have pending interrupts.

2. **Check that it came from the PMU:** It checks whether the bit for the PMU meaning (`XPAR_PSU_IPI_1_BIT_MASK`) is set. If the bit is not set, the handler does nothing further and just clears the interrupt.

3. **Read the message:** When the PMU bit is set, the handler calls `XIpiPsu_ReadMessage()` and copies the 4-word payload from the IPI buffer into `IpiMsgBuf`.

4. **Decode the message:** It then inspects the payload. Word 0 holds a simple API ID, where `0x0001` means "OCM ECC error". Word 1 carries the OCM ISR value, whose bits indicate CE (bit 6) and UE (bit 7). Words 2 and 3 carry the first failing addresses for CE and UE.

5. **Set an event flag for the task:** If the API ID shows an OCM ECC event, the handler sets the volatile flag `ocm_ecc_event_received` to 1. The `HeartbeatTask` checks this flag later in task context and prints the detailed error information there.

6. **Send an ACK to the PMU:** The handler builds a one-word response with value 0 (success) and sends it back to the PMU using `XIpiPsu_WriteMessage()`.

7. **Clear the interrupt:** Finally, it calls `XIpiPsu_ClearInterruptStatus()` with the saved status value to clear the IPI interrupt bits (write 1 to clear), allowing the next IPI event to be delivered.

The handler prints diagnostic messages at each step using `xil_printf()`, including the interrupt status, message contents (API ID, ISR value, CE address, UE address) and operation results. These messages are logged to the UART console for post-experiment analysis.

IPI Status Polling Task: `IpiPollTask()` Function

The `IpiPollTask()` is a simple FreeRTOS task that runs concurrently with the `HeartbeatTask` and periodically polls the IPI interrupt status register for debugging purposes. The task executes an infinite loop where it calls `XIpiPsu_GetInterruptStatus()` to read the current IPI status, prints the status value if it is non-zero, and then delays for about 2 seconds using `vTaskDelay`. The task has the same priority as the `HeartbeatTask` (`tskIDLE_PRIORITY + 1`), so the two tasks share CPU time with the round-robin algorithm.

The purpose of this polling task is to provide visibility into the IPI hardware state during development and debugging. If the IPI interrupt handler is not functioning correctly (for example, due to incorrect GIC initialization or interrupt routing), the polling task can still detect pending IPI messages by reading the status register directly. Under normal operation, the status register should be zero most of the time, with brief periods of non-zero status when IPI messages arrive from the PMU. If the status register shows persistent non-zero values, this indicates that IPI messages are being sent but not acknowledged, which suggests a problem with the interrupt handler or message processing logic.

Table 4.5 summarizes the main components of the R5 application and their roles in the fault detection and notification system.

Table 4.5: Main components of the Cortex-R5 application and their functions.

Component	Function
`main()`	Entry point; initializes IPI, creates tasks, starts FreeRTOS scheduler
`IpiInit()`	Initializes XIpiPsu driver, registers interrupt handler via FreeRTOS, enables IPI channel
`IpiInterruptHandler()`	ISR for IPI interrupts; reads message from PMU, decodes error info, sets event flag, sends ACK
`HeartbeatTask()`	Periodic task; executes workload, checks for error events, logs received error details
`IpiPollTask()`	Debug task; periodically polls IPI status register to verify hardware functionality
`ocm_ecc_event_received`	Volatile flag set by ISR and checked by task; signals arrival of error notification
`IpiMsgBuf[]`	4-word buffer containing IPI message: API ID, OCM ISR, CE address, UE address

4.3.2 PMU Error Manager and OCM ECC IPI Handler

This part focuses on the Platform Management Unit (PMU) firmware that detects hardware errors and decides how to react to them, emphasizing on OCM ECC faults. In general the PMU Error Manager is able to watch a wide range of error sources and can log which ones occurred. It can also reset parts of the system or call for custom event handlers. For the OCM ECC case, a custom handler collects detailed error information and sends it to the Cortex-R5 application via IPI, completing the end to end error notification path introduced in the previous subsection.

ErrorTable: central map of errors and actions

The `ErrorTable` array declared in the `xpfw_error_manager.c` file is the central lookup table that tells the PMU firmware how to treat each possible error source. Each entry corresponds to one error ID (i.e. OCM ECC, RPU lockstep error, or DDR ECC) and groups together four key properties: *a.*which hardware status bit indicates the error,*b.*what type of error it is, *c.*which action to take when it fires and *d.*which CPUs are allowed to change that configuration over IPI.

In concrete terms, every XPfw_Error_t entry in ErrorTable contains:

- **RegMask**: a bit mask that points to the corresponding bit in the PMU error status registers (either PMU_GLOBAL_ERROR_STATUS_1 or _2), so the firmware can quickly check whether this specific error is active.
- **Type**: a small code (EM_ERR_TYPE_1 or EM_ERR_TYPE_2) that tells the firmware which of the two error-status banks this error belongs to.
- **Action**: the default response when this error occurs, such as EM_ACTION_NONE (log only), EM_ACTIO N_SRST (reset), EM_ACTION_PSERR (drive the PSERR pin), or EM_ACTION_CUSTOM (call a firmware handler).
- **Handler**: a function pointer that is used only when the action is EM_ACTION_CUSTOM; this function is invoked to implement any custom reaction to the error.
- **ChngPerm**: a bit mask that encodes which masters (APU, RPU0, RPU1) are allowed to reconfigure this error via IPI commands sent to the PMU.

Most entries in the table simply use built-in handlers such as NullHandler or SwdtHandler and map to standard actions like PS error signaling or subsystem resets. But for our test cases the OCM ECC entry (EM_ERR_ID_OCM_ECC) is configured differently: its Handler is set to OcmEccErrHandler, its Action is EM_ACTION_CUSTOM, and its permissions are restricted so that only the RPU can change its configuration.

Error processing and XPfw_EmProcessError()

The rest of xpfw_error_manager.c is responsible for enabling, disabling, and processing errors at runtime based on the information stored in ErrorTable. XPfw_EmInit() runs once at startup: it disables all actions for all error IDs, snapshots and clears any stale error bits, and then enables the global error signals so that new events will be captured cleanly. Then, XPfw_EmSetAction() is used to configure how each error behaves, by enabling the appropriate reset, interrupt, or PSERR path, or by enabling the interrupt path for custom handlers.

The key runtime function is XPfw_EmProcessError(), which is called whenever the PMU reports that either a TYPE_1 or TYPE_2 error has occurred. Its operation can be described in a few steps:

1. First it selects the correct PMU error-status register (PMU_GLOBAL_ERROR_STATUS_1 forTYPE_1 or TYPE_2 and reads its current value into ErrRegVal. For debugging, it also prints this raw register value so that the developer can see which bits are set.
2. Then , it acknowledges the errors at the hardware level by writing ErrRegVal back to the same register, using the standard ”write-1-to-clear”. This ensures that the PMU does not repeatedly signal the same event.
3. It walks through all entries in ErrorTable and for each error who Type matches the given ErrorType, it checks whether the corresponding bit in ErrRegVal is set. If the bit is set, it means that this particular error has just occurred.
4. Finally , for every active error, it ORs the relevant bits into the ErrorLog array, which keeps a software log of the errors that have been seen. If the error is configured with EM_ACTION_CUSTOM and has a non NULL handler, XPfw_EmProcessError() calls that handler as an action.

The XPfw_EmProcessError() acts like a dispatcher: it decodes the hardware error-status register, records which error IDs were involved, and then triggers the right action by calling the handler associated with each custom error. For OCM ECC, this is exactly how the PMU ends up calling OcmEccErrHandler() when the OCM ECC bit is set, which then bridges the PMU world to the R5 application through IPI.

EM module setup and OCM ECC custom handler

The `xpfw_mod_em.c` file wraps the Error Manager in a PMU firmware module and wires it into the core event and IPI infrastructure. The `ModEmInit()` function creates the EM module and then installs a configuration handler (`EmCfgInit`), an event handler (`EmEventHandler`), and an IPI handler (`EmIpiHandler`), so that the module can react to hardware errors and also handle configuration requests from other processors.

During configuration, `EmCfgInit()` registers the EM module for the two error events:

- `XPFW_EV_ERROR_1`
- `XPFW_EV_ERROR_2`

and calls `XPfw_EmInit()` to reset the error logic, and then iterates over `ErrorTable` to apply the default actions for each error ID.

When a hardware error occurs, the PMU core raises one of the error events that `EmEventHandler()` listens to. For `XPFW_EV_ERROR_1` the handler calls `XPfw_EmProcessError(EM_ERR_TYPE_1)` and for `XPFW_EV_ERROR_2` it calls `XPfw_EmProcessError(EM_ERR_TYPE_2)`. This is the link between the low-level PMU error status bits and the highlevel encoded behavior in `ErrorTable` including the custom OCM ECC path.

The central point in the PMU customization is the `OcmEccErrHandler()` function. This routine is called by the Error Manager whenever the OCM ECC error bit is set, and it translates the raw OCM ECC registers into a compact IPI message for the R5. Its behavior can be summarized as follows:

1. First reads a snapshot of all relevant OCM ECC registers: the interrupt status (`OCM_ISR`), the first failing address and ECC code for correctable errors and the first failing address and data words for uncorrectable errors. These reads capture the full context of the error at the moment it is detected.
2. Then it has some prints/debug messages to the PMU console, showing the OCM ISR value and if present, the details of any CE or UE event (addresses and data).
3. It clears the active OCM ECC interrupt bits by writing the snapshot of `OCM_ISR` back to the `OCM_ISR` register (”write 1 to clear”).
4. It builds a 4-word IPI payload with a simple format:
    a. Word 0 is an API ID (`EM_IPI_API_OCM_ECC = 0x0001`).
    b. Word 1 is the OCM ISR snapshot.
    c. Word 2 is the CE first failing address.
    d. Word 3 is the UE first failing address.
5. Sends the above payload to the R5-0 core though `XPfw_IpiWriteMessage()` which writes the message and then though `XPfw_IpiTrigger()` which triggers its sending, targeting the mask `EM_IPI_DEST_R5_0`. If either the write or the trigger fails, it prints an error and returns without waiting for an acknowledgement.
6. Finally, the IPI trigger succeeds, it prints a status message, then optionally waits for an acknowledgement from the R5 using `XPfw_IpiPollForAck()` with a timeout. On success, it logs that the R5 acknowledged the message. On timeout, it prints an error indicating that no ACK was received within the expected time.

With this custom handler a raw OCM ECC event can be turned into a structured notification that the R5 can consume in task context. Combined with the R5-side `IpiInterruptHandler()` and `HeartbeatTask()`, it completes the loop: hardware detects an ECC error, the PMU classifies and reports it via IPI, and the R5 application logs and reacts to the fault in a controlled way.

4.3.3 Fault Injection Scripts and RPU Lockstep Error Triggering

Fault injection can be performed using two approaches: Tcl scripts executed in the XSCT console within Vitis, or custom C functions running directly on the target processor. Tcl scripts provide a convenient way to inject faults from the host debugger without recompiling or reloading applications.Tcl scripts are ideal for iterative testing. On the other hand custom C functions offer programmatic control within the application runtime. Custom function are more efficient in cases where faults can be injected as part of automated test sequences.Both methods target the same OCM fault injection registers and RPU error injection mechanisms but provide different levels of flexibility and integration.

Correctable and Uncorrectable ECC Error Injection in OCM (Tcl Scripts)

The process of the Tcl scripts that were used to inject both correctable (CE) and uncorrectable (UE) ECC errors into the OCM involve the following steps:

1. Enable the ECC UE interrupt in the OCM Interrupt Enable Register (OCM_IEN) at address `0xFF96 000C` to ensure the system will report any detected ECC errors.
2. Set the Fault Injection Data Register (OCM_FI_D0) at `0xFF96004C` to flip the desired bits (bit 0 and bit 1 for UE, bit 0 for CE) in the next OCM access.
3. Clear the Fault Injection Counter Register (OCM_FI_CNTR) at `0xFF960074` so that the next access triggers the fault.
4. Write data to OCM at address `0xFFFE0000` to trigger the ECC logic.
5. Read back the data from OCM to confirm the error event.
6. Check the OCM Interrupt Status Register (ISR) to verify the type of error detected (CE or UE).

This process allows for controlled and repeatable testing of the system's response to ECC errors in OCM, ensuring the error management module functions as expected.

RPU Lockstep Error Injection (C Code)

The C code is used to trigger a lockstep error in the RPU by manipulating the RPU_ERR_INJ register. The process involves the following steps:

1. Enable lockstep error injection by writing `0xC0` to the RPU_ERR_INJ register at address `0xFFD805A0`.
2. Check the ERROR_STATUS_1 register at `0xFFD80530` to confirm the error injection is enabled.
3. Trigger the lockstep error by writing `0xFFF` to the RPU_ERR_INJ register at `0xFF9A0020`.
4. Immediately reset the RPU_ERR_INJ register to zero to prevent continuous error generation.
5. The error handler function is called, which logs the event and initiates an RPU reset if necessary.

This sequence allows for testing the system's response to lockstep errors, ensuring the error management module can detect and recover from such fault.

5. Experimental Results and Conclusions

This section presents and analyzes the experimental results obtained from fault injection experiments conducted on the Zynq UltraScale+ MPSoC, focusing on the Cortex-R5 Real-Time Processing Unit (RPU) operating in lockstep mode. Three representative test cases are examined in detail, corresponding to correctable ECC faults, uncorrectable ECC faults, and processor-level lockstep faults injected through the dedicated error injection register `RPU_ERR_INJ`. The objective is to observe and analyze system behavior across different fault severity levels, spanning from recoverable memory errors to unrecoverable processor execution mismatches.

Before introducing fault conditions, it is important to establish the baseline system behavior and initialization sequence observed during application startup.

Figure 5.1 illustrates the system boot sequence and initial execution state prior to fault injection. The First Stage Boot Loader (FSBL) is executed on the RPU, followed by the initialization of the Platform Management Unit (PMU) firmware. During this phase, the Error Management (EM) module is initialized, and ECC protection for the On-Chip Memory (OCM) is enabled, with detected ECC events reported via interrupts to the Platform Management Unit for software-level handling.

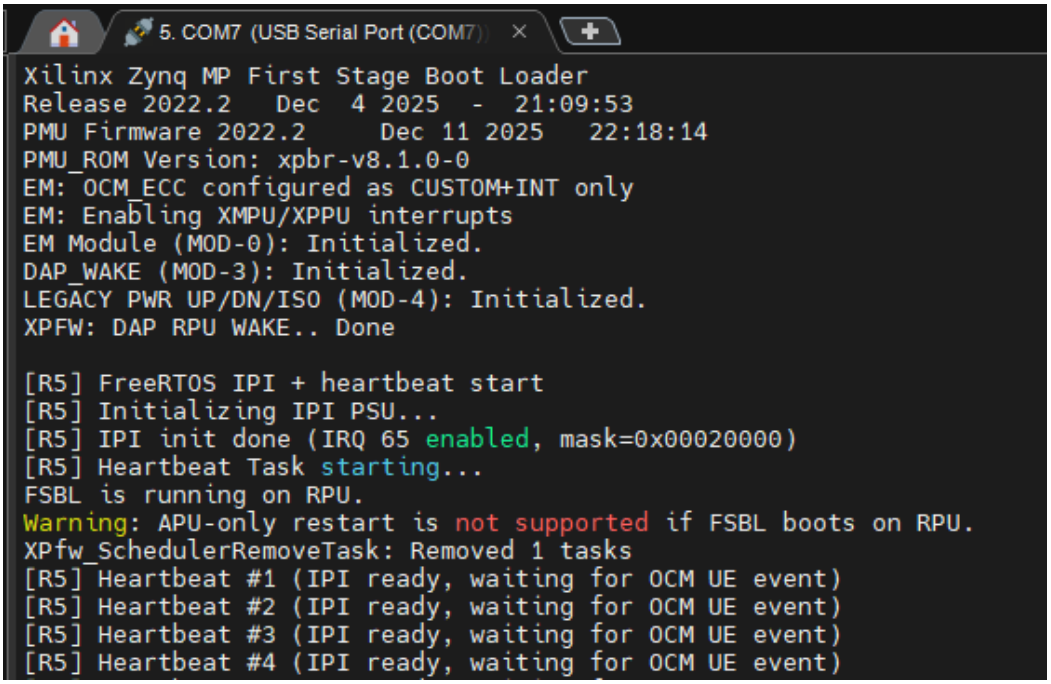

Figure 5.1: FreeRTOS HeartBeat Application Start

Subsequently, the FreeRTOS application is launched on the R5 core. The application initializes the Inter-Processor Interrupt (IPI) infrastructure, enabling IRQ 65 for PMU-to-RPU communication, and starts a periodic heartbeat task. The heartbeat output, which repeatedly reports that the system is "waiting for OCM UE event", serves as a continuous liveness indicator and confirms that the RPU remains operational, responsive, and synchronized with the PMU prior to fault injection.

This stable baseline state provides a reference point against which the effects of injected faults can be evaluated.

5.1 Test Case 1: Correctable ECC Error (CE)

The first test case evaluates the system response to a correctable ECC fault injected into the On-Chip Memory (OCM). Correctable ECC errors typically correspond to single-bit memory faults, which can be detected and corrected transparently by the ECC logic before corrupted data propagates to software-visible state.

As shown in Figure 5.2, the Platform Management Unit (PMU) reports an OCM ECC error event with an interrupt status register (ISR) value of `0x00000040`. This ISR bit is explicitly associated with an OCM correctable ECC error condition. Importantly, this value differs from the ISR bit used to signal uncorrectable ECC errors, which is reported as `0x00000080`. The distinction between these ISR values provides clear architectural evidence that the observed event corresponds to a correctable, rather than uncorrectable, memory fault.

A second key indicator confirming the correctable nature of the event is the absence of uncorrectable error metadata in the PMU output. No fault address or syndrome registers (such as `FFA`, `FFD0`, or `FFD1`) are populated during this event. In the Zynq UltraScale+ MPSoC architecture, these registers are only updated when the ECC logic is unable to correct the fault and must report detailed diagnostic information. Their absence therefore confirms that the ECC mechanism successfully corrected the fault entirely in hardware.

Following detection, the PMU reports the correctable ECC event through the Error Management (EM) module and issues an inter-processor interrupt (IPI) to the RPU. This demonstrates that correctable errors follow the same notification path as more severe fault types, ensuring that software is made aware of the event for diagnostic or logging purposes.

On the RPU side, the IPI interrupt service routine is invoked, reports the interrupt status, and completes execution successfully. The RPU explicitly acknowledges the event back to the PMU, confirming bidirectional fault signaling between the management and processing subsystems. Finally , no reset or execution interruption occurs as a result of this event.

The FreeRTOS heartbeat task continues to execute normally throughout the fault event, with no observable disruption in timing or control flow. Immediately after the event is handled, the system returns to its steady state operation, continuing to report that it is ready and awaiting further fault conditions.

Overall, this test case demonstrates that correctable ECC errors are detected, corrected, and reported in a controlled and non-disruptive manner. The fault is fully contained within the memory subsystem, while software execution and real-time behavior remain unaffected. This behavior aligns with the intended role of ECC in dependable system design, where transient memory faults are mitigated without sacrificing system availability.

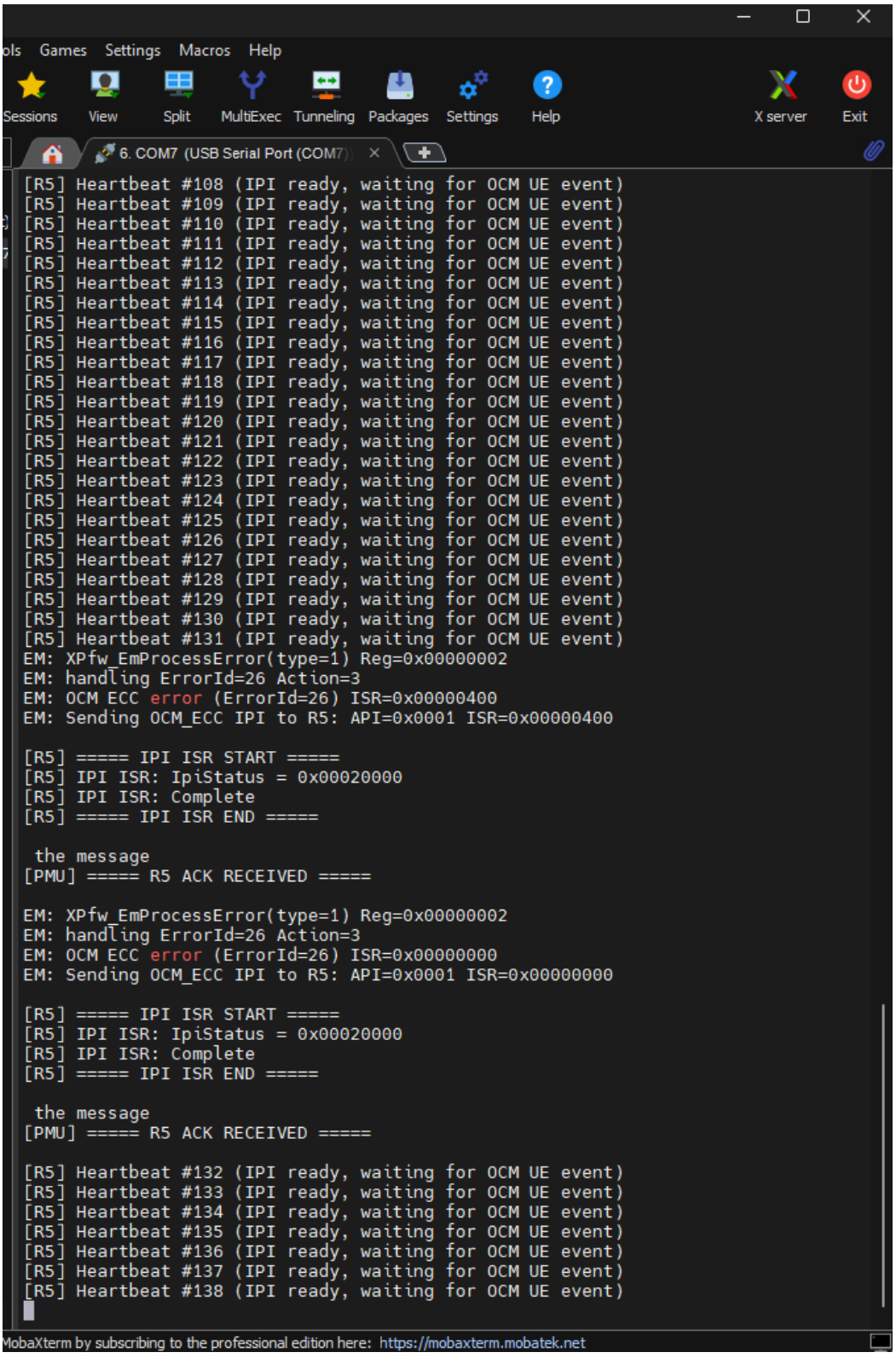

Figure 5.2: Test Case 1: OCM Correctable Error Event

5.2 Test Case 2: Uncorrectable ECC Error (UE)

The second test case evaluates the system response to an uncorrectable ECC fault injected into the On-Chip Memory (OCM). Uncorrectable ECC errors typically arise from multi-bit memory faults that exceed the correction capability of the ECC logic and therefore cannot be transparently repaired in hardware.

As shown in Figure 5.3, the Platform Management Unit (PMU) reports an OCM ECC error event with an interrupt status register (ISR) value of `0x00000080`. This ISR bit is explicitly associated with an uncorrectable ECC error condition and is distinct from the correctable ECC interrupt bit (`0x00000040`) observed in Test Case 1. The asserted ISR value therefore provides the first clear architectural indication that the detected fault is uncorrectable.

In contrast to the correctable error case, the PMU output additionally includes a detailed uncorrectable error report, explicitly marked with a `UE` field. This report contains fault metadata registers such as the faulting address and syndrome information (e.g., `FFA`, `FFD0`, and `FFD1`). These registers are only populated when the ECC logic is unable to correct the fault and must therefore expose diagnostic information to higher system layers. Their presence unequivocally confirms that the injected fault resulted in an uncorrectable ECC condition.

Following detection of the uncorrectable ECC error, the PMU escalates the event by issuing an inter-processor interrupt (IPI) to the RPU. The use of the same IPI signaling path as in the correctable error case demonstrates that both fault types are reported through a unified notification mechanism, while still preserving severity differentiation through the ISR encoding and associated metadata.

On the RPU side, the IPI interrupt service routine is entered, reports the interrupt status, and completes execution successfully. The RPU explicitly acknowledges the event back to the PMU, as indicated by the confirmation message logged by the PMU firmware. This bidirectional handshake between the PMU and RPU confirms that the uncorrectable fault has been detected, propagated, and acknowledged across system management and processing domains.

Importantly, despite the increased severity of the fault, no immediate reset of the RPU is triggered in this test configuration. The FreeRTOS application resumes execution after servicing the interrupt, and the heartbeat task continues to operate normally. This behavior indicates that, while the fault is classified as uncorrectable at the memory level, the system is configured to allow software-level awareness and potential mitigation rather than enforcing an automatic hardware reset.

Overall, this test case demonstrates that uncorrectable ECC errors are reliably detected, fully characterized, and explicitly reported to software. Unlike correctable errors, uncorrectable faults generate detailed diagnostic information and are treated as higher-severity events, yet they can still be handled in a controlled manner that preserves system availability and enables higher-level fault management strategies.

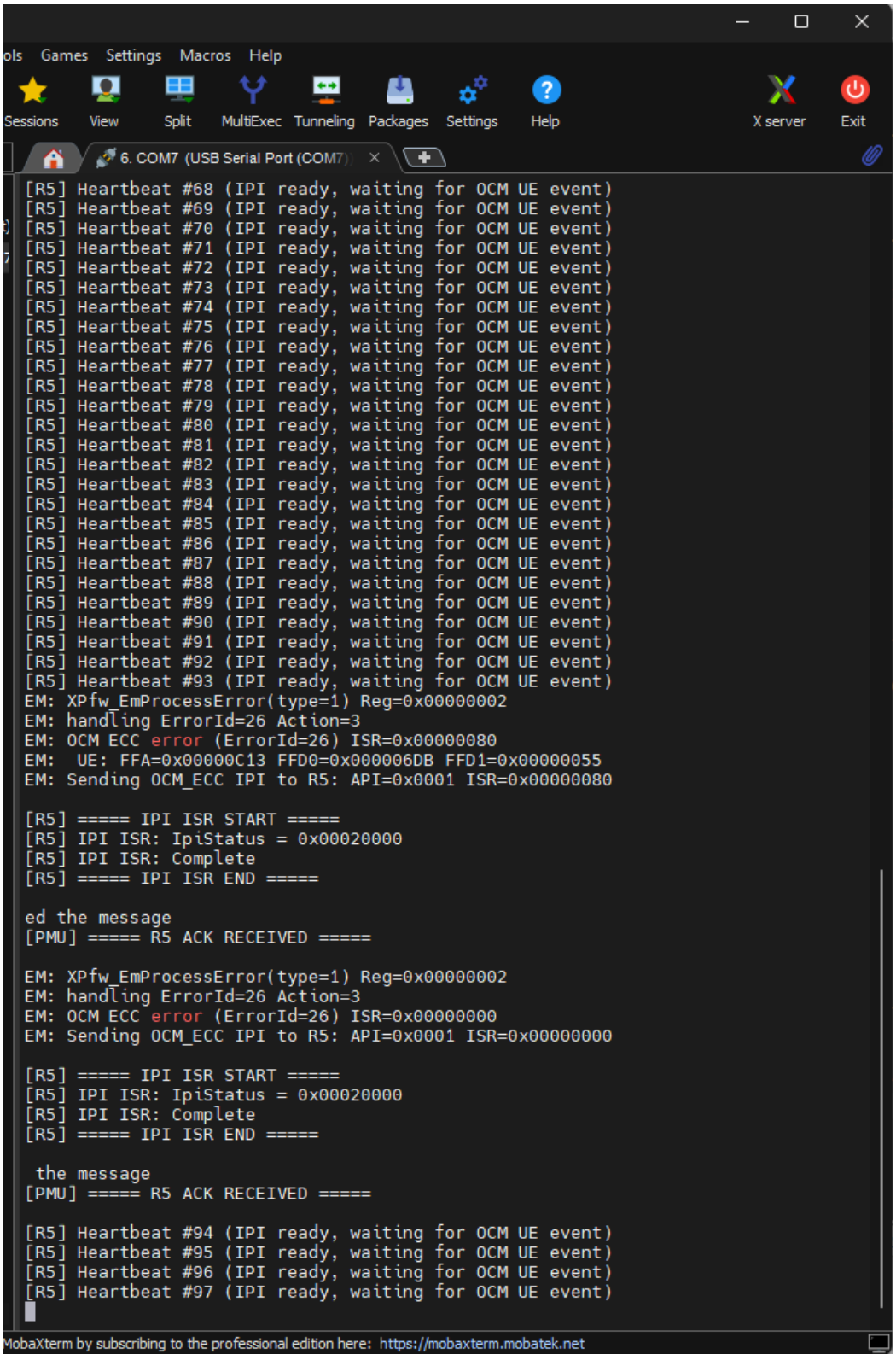

Figure 5.3: Test Case 2: OCM Uncorrectable Error Event

5.3 Test Case 3: Uncorrectable ECC Error (UE)

The third test case evaluates the system response to a processor-level fault introduced by deliberately injecting a lockstep execution mismatch in the Cortex-R5 Real-Time Processing Unit (RPU). Unlike the previous test cases, which target memory-level faults handled by ECC mechanisms, this test focuses on architectural fault detection within the processor execution pipeline itself.

Prior to fault injection, the system operates in a stable baseline state, as indicated by the continuous output of the FreeRTOS heartbeat task. The repeated heartbeat messages confirm that the RPU is executing normally, FreeRTOS scheduling is active, and no ECC-related faults or PMU interventions are present at the time of injection.

The lockstep fault is injected using a sequence of register writes issued through a TCL script. First, the RPU lockstep error injection mechanism is enabled by writing to the corresponding error injection control register. Subsequently, a trigger value is written to the lockstep injection register, forcing a divergence between the two redundant execution pipelines operating in lockstep mode. The trigger is then cleared, ensuring that the fault is transient rather than persistent. Architecturally, this sequence induces a mismatch between the lockstep cores, representing a worst-case processor fault scenario, as it directly violates the fundamental assumption of identical execution in redundant pipelines.

Following fault injection, the Platform Management Unit (PMU) immediately reports the event as an *RPU Lock-Step Error*, identified by a dedicated error identifier. This message indicates that the lockstep comparator has detected a divergence between the redundant cores. Unlike ECC-related faults, this error is architectural in nature and cannot be corrected or masked by hardware logic or software intervention.

Upon detection of the lockstep mismatch, the PMU initiates an immediate reset of the RPU subsystem. The reset sequence includes acknowledgment from internal interfaces, followed by a controlled release of the RPU from reset. This behavior is mandatory for lockstep faults, as a divergence in processor execution state invalidates the correctness of all subsequent computations. Allowing execution to continue would violate functional safety guarantees and could lead to unpredictable system behavior.

After the reset sequence completes, the RPU reboots and restarts execution from a known-safe state. The FreeRTOS application is reloaded, the IPI infrastructure is reinitialized, and the heartbeat task resumes execution. The successful restart of the application confirms that the reset was performed in a controlled and deterministic manner, without deadlock or undefined behavior.

This test case demonstrates a clear distinction between memory-level faults and processor-level faults. While ECC-related errors permit continued execution under controlled conditions, lockstep execution mismatches are treated as unrecoverable by design. The enforced reset mechanism ensures fail-safe recovery and prevents the propagation of corrupted processor state. The observed behavior validates the role of lockstep execution and PMU supervision in enforcing architectural safety policies that cannot be overridden by software.

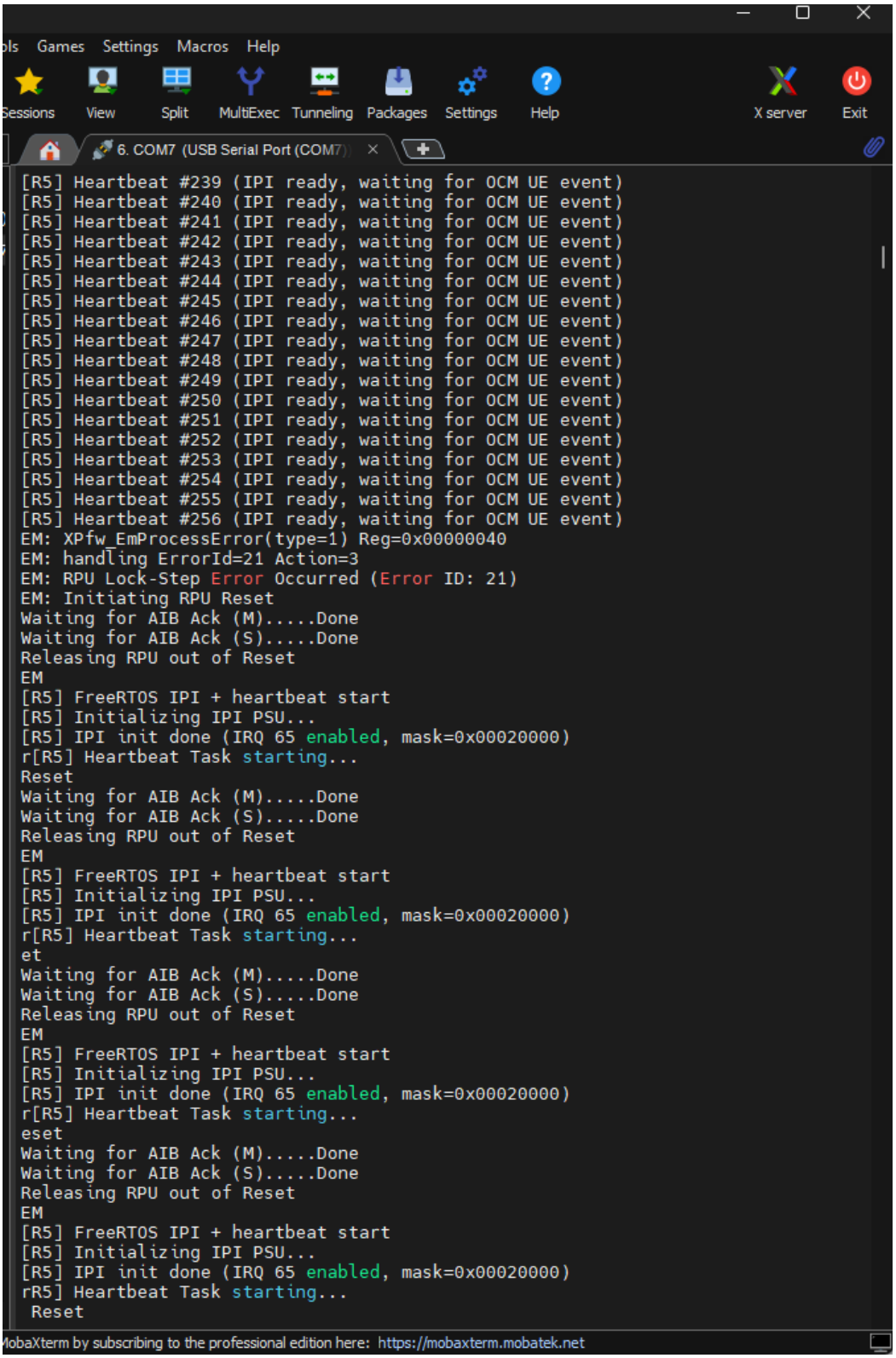

Figure 5.4: Test Case 3: RPU Lockstep Error Injection Event

5.4 Conclusions and Future Work

This work presented an experimental investigation of fault detection and handling mechanisms in the Zynq UltraScale+ MPSoC, focusing on the Cortex-R5 Real-Time Processing Unit operating in lockstep mode. Through a series of controlled fault injection experiments conducted on real hardware, the behavior of the system was observed under different fault classes, ranging from recoverable memory errors to unrecoverable processor execution faults.

Three representative fault scenarios were examined. In the first test case, correctable ECC errors in the On-Chip Memory were injected and analyzed. The results demonstrated that such faults are detected and transparently corrected by the hardware ECC logic, while still being reported to the software layer through interrupt-based signaling. System execution continued uninterrupted, and real-time behavior was preserved, validating the role of ECC in improving reliability without impacting availability.

The second test case focused on uncorrectable ECC errors. In this scenario, the ECC logic detected faults that exceeded its correction capability and generated detailed diagnostic information, including fault address and syndrome data. These errors were escalated to the Platform Management Unit and reported to the RPU via inter-processor interrupts. Despite the higher severity of the fault, the system remained operational in the selected configuration, allowing software-level awareness and potential mitigation strategies to be applied. This behavior highlights the flexibility of the architecture in supporting controlled fault handling policies beyond immediate hardware resets.

The third test case addressed processor-level faults by injecting a lockstep execution mismatch in the Cortex-R5 cores. Unlike memory-related faults, lockstep mismatches represent architectural failures that invalidate processor execution state. Experimental results confirmed that such faults are detected by the lockstep comparator and immediately handled by the Platform Management Unit through enforced RPU reset. Execution was halted and restarted from a known-safe state, demonstrating deterministic and fail-safe recovery behavior that cannot be overridden by software.

Taken together, these results illustrate a clear hierarchy of fault severity and corresponding system responses. Correctable memory faults are masked and logged, uncorrectable memory faults are detected and escalated for software awareness, and processor-level lockstep faults trigger mandatory architectural recovery. The experiments confirm that the Zynq UltraScale+ MPSoC provides a well-defined and robust framework for fault detection, isolation, and handling, combining hardware mechanisms with software-visible signaling to support dependable real-time system design.

While this work focused on targeted fault injection scenarios and observation of system-level behavior, several directions for future work can be identified.

One possible extension is the exploration of more complex fault models, including persistent or repeated faults, as well as fault injection under varying workload conditions. Evaluating system behavior under sustained fault conditions could provide further insight into long-term reliability and recovery robustness.

Another promising direction involves extending the fault handling logic at the software level. In this work, software primarily acknowledged and logged fault events. Future work could investigate application-level mitigation strategies, such as task reconfiguration, graceful degradation, or controlled shutdown in response to uncorrectable errors.

Additionally, the interaction between the Real-Time Processing Unit and other processing subsystems, such as the Application Processing Unit or programmable logic, could be explored in the context of coordinated fault handling. This would allow investigation of system-wide fault propagation and containment in heterogeneous MPSoC environments.

Finally, future studies could incorporate external fault injection methods or radiation-based testing to validate the observed behavior under more realistic fault conditions. Such experiments would further strengthen confidence in the applicability of the presented results to safety-critical and high-reliability domains.

Overall, the experimental framework and results presented in this dissertation provide a solid foundation for continued research into dependable MPSoC architectures and fault-tolerant real-time systems.

A. Implementation Listings

A.1 RPU FreeRTOS Application (ipi_heartbeat_app.c)

Listing A.1: RPU FreeRTOS IPI Heartbeat Application with PMU-to-RPU ECC Notification

```
/*
 * RPU FreeRTOS application implementing:
 *  - Periodic heartbeat task
 *  - IPI-based communication with the PMU
 *  - Reception and handling of OCM ECC error notifications
 *
 * The application demonstrates how ECC events detected by the PMU
 * are propagated to the Cortex-R5 processor via IPI and handled
 * at the software level.
 */

#include "FreeRTOS.h"
#include "task.h"

#include "xil_printf.h"
#include "xparameters.h"
#include "xipipsu.h"
#include "xil_exception.h"

/* ================================================================
 * IPI configuration
 * ================================================================ */
#define IPI_DEVICE_ID       XPAR_XIPIPSU_0_DEVICE_ID
#define IPI_INT_ID          XPAR_XIPIPSU_0_INT_ID
#define IPI_RECV_MASK       XPAR_PSU_IPI_4_BIT_MASK
#define IPI_PMU_MASK        XPAR_PSU_IPI_3_BIT_MASK

#define IPI_MSG_SIZE        8U

static XIpiPsu IpiInstance;
static u32 IpiMsgBuf[IPI_MSG_SIZE];
static u32 IpiRespBuf[IPI_MSG_SIZE];

static volatile u32 ocm_ecc_event_received = 0;

/* ================================================================
 * IPI interrupt handler
 * ================================================================ */
void IpiInterruptHandler(void *CallbackRef)
{
    XIpiPsu *IpiPtr = (XIpiPsu *)CallbackRef;
    u32 IpiStatus;
```

```
    xil_printf("\r\n[R5] === IPI ISR START ===\r\n");

    IpiStatus = XIpiPsu_GetInterruptStatus(IpiPtr);

    if ((IpiStatus & XPAR_PSU_IPI_1_BIT_MASK) != 0U) {

        if (XIpiPsu_ReadMessage(IpiPtr,
                                IPI_RECV_MASK,
                                IpiMsgBuf,
                                IPI_MSG_SIZE,
                                XIPIPSU_BUF_TYPE_MSG) == XST_SUCCESS) {

            if ((IpiMsgBuf[0] & 0xFFFFU) == 0x0001U) {
                xil_printf("[R5] OCM ECC event received via IPI\r\n");
                ocm_ecc_event_received = 1;
            }

            /* Send ACK back to PMU */
            IpiRespBuf[0] = 0U;
            XIpiPsu_WriteMessage(IpiPtr,
                                 IPI_PMU_MASK,
                                 IpiRespBuf,
                                 1,
                                 XIPIPSU_BUF_TYPE_RESP);
        }
    }

    XIpiPsu_ClearInterruptStatus(IpiPtr, IpiStatus);

    xil_printf("[R5] === IPI ISR END ===\r\n\r\n");
}

/* ============================================================
 * IPI initialization
 * ============================================================ */
static BaseType_t IpiInit(void)
{
    XIpiPsu_Config *CfgPtr;

    CfgPtr = XIpiPsu_LookupConfig(IPI_DEVICE_ID);
    if (CfgPtr == NULL) {
        return pdFAIL;
    }

    if (XIpiPsu_CfgInitialize(&IpiInstance,
                              CfgPtr,
                              CfgPtr->BaseAddress) != XST_SUCCESS) {
        return pdFAIL;
    }

    XIpiPsu_Reset(&IpiInstance);
```

```
    extern BaseType_t xPortInstallInterruptHandler(uint8_t,
                                                   XInterruptHandler,
                                                   void *);
    extern void vPortEnableInterrupt(uint8_t);

    xPortInstallInterruptHandler((uint8_t)IPI_INT_ID,
                                 IpiInterruptHandler,
                                 &IpiInstance);

    vPortEnableInterrupt((uint8_t)IPI_INT_ID);

    XIpiPsu_InterruptEnable(&IpiInstance, IPI_RECV_MASK);

    return pdPASS;
}

/* ============================================================
 * Heartbeat task
 * ============================================================ */
static void HeartbeatTask(void *pvParameters)
{
    static u32 heart_count = 0;

    xil_printf("[R5] Heartbeat task started\r\n");

    for (;;) {
        vTaskDelay(pdMS_TO_TICKS(2000));
        heart_count++;

        xil_printf("[R5] Heartbeat #%lu\r\n", heart_count);

        /* Inject lockstep error on 100th heartbeat */
        /*
        if (heart_count == 100) {
            xil_printf("[R5] >>> Injecting RPU lockstep error <<<\r\n");
            Xil_Out32(0xFFD805A0, 0xC0);
            Xil_Out32(0xFF9A0020, 0xFF);
            Xil_Out32(0xFF9A0020, 0x00);
        }
        */

        if (ocm_ecc_event_received) {
            xil_printf("[R5] Handling OCM ECC event\r\n");
            ocm_ecc_event_received = 0;
        }
    }
}

/* ============================================================
 * Main
 * ============================================================ */
```

```
int main(void)
{
    xil_printf("[R5] FreeRTOS IPI heartbeat application start\r\n");

    if (IpiInit() != pdPASS) {
        xil_printf("[R5] IPI initialization failed\r\n");
        for (;;) {}
    }

    __asm volatile ("CPSIE i");

    xTaskCreate(HeartbeatTask,
                "Heartbeat",
                512,
                NULL,
                tskIDLE_PRIORITY + 1,
                NULL);

    Xil_ExceptionInit();
    Xil_ExceptionEnable();

    vTaskStartScheduler();

    for (;;) {}
}
```

A.2 PMU Firmware Error Management Handlers

A.2.1 OCM ECC Handler with PMU-to-RPU IPI Notification

Listing A.2: PMU Firmware OCM ECC Error Handler: capture metadata and notify the RPU via IPI

```
/**
 * OCM ECC (CE/UE) error handler for EM_ACTION_CUSTOM
 * Called by the Error Manager when OCM ECC error is detected
 *
 * This handler:
 * 1. Captures OCM error details
 * 2. Sends IPI message to R5 with error info
 * 3. Waits for ACK from R5
 */
void OcmEccErrHandler(u8 ErrorId)
{
    u32 Isr, CeFfa, CeFfe, UeFfa, UeFfd0, UeFfd1;
    u32 Msg[4];
    s32 Status;
    const u32 TimeOut = 1000U;  /* 1 second timeout */

    /* 1. Snapshot OCM ECC status and failing info */
    Isr    = XPfw_Read32(OCM_ISR);
    CeFfa  = XPfw_Read32(OCM_CE_FFA);
    CeFfe  = XPfw_Read32(OCM_CE_FFE);
    UeFfa  = XPfw_Read32(OCM_UE_FFA);
    UeFfd0 = XPfw_Read32(OCM_UE_FFD0);
    UeFfd1 = XPfw_Read32(OCM_UE_FFD1);

    XPfw_Printf(DEBUG_ERROR,
                "EM: OCM ECC error (ErrorId=%d) ISR=0x%08lx\r\n",
                ErrorId, Isr);

    if ((Isr & OCM_ISR_CE_MASK) != 0U) {
        XPfw_Printf(DEBUG_ERROR,
                    "EM:  CE: FFA=0x%08lx FFE=0x%08lx\r\n",
                    CeFfa, CeFfe);
    }

    if ((Isr & OCM_ISR_UE_MASK) != 0U) {
        XPfw_Printf(DEBUG_ERROR,
                    "EM:  UE: FFA=0x%08lx FFD0=0x%08lx FFD1=0x%08lx\r\n",
                    UeFfa, UeFfd0, UeFfd1);
    }

    /* 2. Clear OCM interrupt bits (write-1-to-clear) */
    XPfw_Write32(OCM_ISR, Isr);

    /* 3. Build IPI message for R5 with error info */
    Msg[0] = (u32)EM_IPI_API_OCM_ECC;  /* 0x0001 */
    Msg[1] = Isr;
```

```
    Msg[2] = CeFfa;
    Msg[3] = UeFfa;

    /* 4. Send IPI message to R5-0 */
    XPfw_Printf(DEBUG_PRINT_ALWAYS,
                "EM: Sending OCM_ECC IPI to R5: API=0x%04x ISR=0x%08lx \r\n",
                EM_IPI_API_OCM_ECC, Isr);

    Status = XPfw_IpiWriteMessage(EmModPtr, EM_IPI_DEST_R5_0,
                                  Msg, (u8)(sizeof(Msg) / sizeof(Msg[0])));

    if (Status != XST_SUCCESS) {
        XPfw_Printf(DEBUG_ERROR,
                    "EM: Failed to write IPI message to R5 (Status=%ld)\r\n",
                    Status);
        return;
    }

    /* 5. Trigger the IPI */
    Status = XPfw_IpiTrigger(EM_IPI_DEST_R5_0);

    if (Status != XST_SUCCESS) {
        XPfw_Printf(DEBUG_ERROR,
                    "EM: Failed to trigger IPI to R5 (Status=%ld)\r\n",
                    Status);
        return;
    }

    XPfw_Printf(DEBUG_PRINT_ALWAYS,
                "EM: OCM_ECC IPI sent to R5, waiting for ACK...\r\n");

    /* 6. Wait for ACK from R5 */
    XPfw_Printf(DEBUG_PRINT_ALWAYS,
                "\r\n[PMU] ===== OCM_ECC IPI SENT, WAITING ACK =====\r\n");
    Status = XPfw_IpiPollForAck(EM_IPI_DEST_R5_0, TimeOut);

    if (Status == XST_SUCCESS) {
        XPfw_Printf(DEBUG_PRINT_ALWAYS,
                    "EM: OCM_ECC: R5 acknowledged the message\r\n");
    } else {
        XPfw_Printf(DEBUG_ERROR,
                    "EM: OCM_ECC: R5 ACK timeout (waited %lu ms)\r\n",
                    TimeOut);
    }
    XPfw_Printf(DEBUG_PRINT_ALWAYS,
                "[PMU] ===== R5 ACK RECEIVED =====\r\n\r\n");
}
```

A.2.2 RPU Lockstep Error Handler and Fail-Safe Reset

Listing A.3: PMU Firmware RPU Lockstep Error Handler: report and reset the RPU

```
/**
 * Example Error handler for RPU lockstep error
 *
 * This handler should be called when an R5 lockstep error occurs
 * and it resets the RPU gracefully
 */
void RpuLsHandler(u8 ErrorId)
{
    XPfw_Printf(DEBUG_ERROR,"EM: RPU Lock-Step Error Occurred "
            "(Error ID: %d)\r\n", ErrorId);
    XPfw_Printf(DEBUG_ERROR,"EM: Initiating RPU Reset \r\n");
    if (XST_SUCCESS != XPfw_ResetRpu()) {
        XPfw_Printf(DEBUG_ERROR,"EM: Error in Reset RPU\r\n");
    }
}
```

A.3 OCM Error Injection Functions

A.3.1 OCM Correctable Error injection

Listing A.4: OCM Correctable Error Injection

```
void inject_ocm_CE(void)
{
    xil_printf("\r\n[OCM] Injecting a Correctable ECC error...\r\n");

    /* 1) Program FI to flip a single data bit */
    Xil_Out32(OCM_FI_D0, 0x00000001U);  /* flip bit0 in lower 32 bits */
    Xil_Out32(OCM_FI_D1, 0x00000000U);
    Xil_Out32(OCM_FI_D2, 0x00000000U);
    Xil_Out32(OCM_FI_D3, 0x00000000U);
    Xil_Out32(OCM_FI_SY, 0x00000000U);

    /* 2) Counter = 0 -> next access injects fault */
    Xil_Out32(OCM_FI_CNTR, 0x00000000U);

    /* 3) Use C[0][0] in OCM as test word */
    volatile u32 *addr = (u32 *)&matrix_C[0][0];
    u32 test_val = 0xDEADBEEFU;

    Xil_Out32((UINTPTR)addr, test_val);
    u32 read_val = Xil_In32((UINTPTR)addr);

    if (read_val == test_val) {
        xil_printf("  CE injection: value corrected to 0x%08lX (bit0 was flipped).\r\n
                read_val);
    } else {
        xil_printf("  CE injection: unexpected read 0x%08lX (wrote 0x%08lX)\r\n",
                read_val, test_val);
    }
```

```
    u32 isr = Xil_In32(OCM_ISR);
    xil_printf("  OCM_ISR after CE = 0x%08lX\r\n", isr);
}
```

A.3.2 OCM Unorrectable Error injection

Listing A.5: OCM Uorrectable Error Injection

```
void inject_ocm_UE(void)
{
    xil_printf("\r\n[OCM] Injecting an Uncorrectable ECC error into C[%d][%d]...\r\n"
               C_FAULT_ROW, C_FAULT_COL);

    //volatile u32 *addr = (u32 *)OCM_UE_TEST_ADDR;

    /* Address of C[row][col] in OCM */
    volatile u32 *addr = (u32 *)&matrix_C[C_FAULT_ROW][C_FAULT_COL];

    /* 0) Clear old CE/UE bits (optional for repeated tests) */
    Xil_Out32(OCM_ISR, OCM_ISR_CE_MASK | OCM_ISR_UE_MASK);

    /* 1) Flip two bits in the next word */
    //Xil_Out32(OCM_FI_D0, 0x00000003U);  /* bits 0 and 1 */
    Xil_Out32(OCM_FI_D0, 0x00000001U);  /* bits 0 and 1 */
    Xil_Out32(OCM_FI_D1, 0x00000000U);
    Xil_Out32(OCM_FI_D2, 0x00000000U);
    Xil_Out32(OCM_FI_D3, 0x00000000U);
    Xil_Out32(OCM_FI_SY, 0x00000000U);

    /* 2) ECC enabled already; arm FI counter */
    Xil_Out32(OCM_FI_CNTR, 0x00000000U);

    /* 3) Write + flush, then read to trigger ECC */
    Xil_Out32((UINTPTR)addr, 0x12345678U);
    Xil_DCacheFlushRange((UINTPTR)addr, 4U);

    u32 read_val = Xil_In32((UINTPTR)addr);
    xil_printf("  Read value back: 0x%08lX\r\n", read_val);

    /* 4) Check ISR bits */
    u32 isr = Xil_In32(OCM_ISR);
    xil_printf("  OCM_ISR after UE = 0x%08lX\r\n", isr);

    if ((isr & OCM_ISR_UE_MASK) != 0U) {
        xil_printf("  ==> UE bit set (bit7).\r\n");
    } else if ((isr & OCM_ISR_CE_MASK) != 0U) {
        xil_printf("  ==> Only CE bit set (bit6).\r\n");
    } else {
        xil_printf("  ==> No ECC bits set.\r\n");
    }
}
```

B. TCL Fault Injection Scripts

B.1 OCM Correctable Error Injection Script : Test case 1

Listing B.1: TCL script used to trigger an OCM Correctable Error Event

```
;# Enable ECC UE interrupt in OCM_IEN
mwr -force 0xFF96000C [expr 1<<6]

;# Write to Fault Injection Data 0 Register OCM_FI_D0
;# Errors will be injected in the bits which are set, here its bit0, bit1
mwr -force 0xFF96004C 3

;# Enable ECC and Fault Injection
mwr -force 0xFF960014 1


# Clear the Count Register : OCM_FI_CNTR
mwr -force 0xFF960074 0

;# Now write data to OCM for the fault to be injected
# Since OCM does a RMW for 32-bit transactions, it should trigger error here
mwr -force 0xFFFE0000 0x1234

;# Read back to trigger error again
mrd -force 0xFFFE0000
```

B.2 OCM Uncorrectable Error Injection Script : Test case 2

Listing B.2: TCL script used to trigger an OCM Uncorrectable Error Event

```
;# Enable ECC UE interrupt in OCM_IEN
mwr -force 0xFF96000C [expr 1<<7]

;# Write to Fault Injection Data 0 Register OCM_FI_D0
;# Errors will be injected in the bits which are set, here its bit0, bit1
mwr -force 0xFF96004C 3

;# Enable ECC and Fault Injection
mwr -force 0xFF960014 1


# Clear the Count Register : OCM_FI_CNTR
mwr -force 0xFF960074 0

;# Now write data to OCM for the fault to be injected
# Since OCM does a RMW for 32-bit transactions, it should trigger error here
```

```
mwr -force 0xFFFE0000 0x1234

;# Read back to trigger error again
mrd -force 0xFFFE0000
```

B.3 RPU Lockstep Fault Injection Script: Test case 3

Listing B.3: TCL script used to trigger an RPU lockstep mismatch fault

```
mwr 0xFFD805A0 0xC0       ; Enable RPU Lockstep Error injection
mwr 0xFF9A0020 0xFF       ; Trigger lockstep fault
mwr 0xFF9A0020 0x00       ; Clear the trigger
```

C. Fault Injection via ACTLR Manipulation

C.1 ACTLR Instruction Fetch Experiment

This appendix documents an experimental investigation into microarchitectural fault behavior on the Cortex-R5 processor by manipulating execution ordering and instruction fetch behavior. The experiment focuses on the use of the Secondary Auxiliary Control Register (ACTLR) to influence pipeline ordering and cache behavior, and employs a carefully crafted self-modifying code sequence to provoke instruction stream inconsistency between redundant lockstep cores.

Specifically, selected ACTLR fields related to execution ordering and error control were examined in the context of lockstep execution. By intentionally omitting synchronization barriers and modifying an instruction immediately before execution, the experiment attempts to induce divergent instruction streams between the two lockstep pipelines. This scenario represents a worst-case architectural fault, as lockstep correctness fundamentally relies on identical instruction fetch, decode, and execution behavior across both cores.

According to the Cortex-R5 Technical Reference Manual Chapter 4, modifications to ACTLR fields affecting execution behavior must be followed by explicit instruction synchronization barriers (ISB) to ensure consistent architectural state across the pipeline. Violating this requirement may lead to undefined or implementation-dependent behavior, particularly in safety-critical configurations such as lockstep operation [10].

Table C.1: Relevant ACTLR Fields Considered in the Experiment

Bit(s)	Name	Function
[22]	DCHE	Disables hard-error recovery in cache logic when set
[20:16]	Dual-Issue Controls	Controls instruction issue ordering and execution behavior
[5:3]	CEC	Cache error control for parity and ECC handling

Listing C.1: Self-Modifying Code Experiment without Instruction Synchronization Barrier

```
.text
.global InjectSelfModTest
.type InjectSelfModTest, %function

.extern test_val

InjectSelfModTest:
    // r4 = &test_val, r5 = value to store
    LDR     r4, =test_val
    LDR     r5, =0xDEADBEEF

    // r0 = address of target instruction
    ADR     r0, target_instruction

    // Small delay to allow prefetch in one pipeline
    NOP
    NOP
    NOP
```

```
    // Overwrite instruction just-in-time
    LDR     r1, =0xE5845000         // STR r5, [r4]
    STR     r1, [r0]

    // Intentionally omit ISB / DSB

target_instruction:
    NOP                             // Patched at runtime

    BX      lr
```

D. Live Debugger Attachment and Runtime Observation

D.1 Purpose and Context

During the experimental evaluation presented in Chapters 4 and 5, a live debugger was attached to the target system to observe runtime behavior, register state, and execution flow while fault injection experiments were performed. The debugger attachment was used strictly for observation and control purposes and did not modify the fault handling logic or execution paths under test.

The primary objectives of attaching a live debugger were to verify correct program loading on the Cortex-R5 core, confirm interrupt and exception handling behavior, and correlate console output with internal processor state during fault events.

D.2 How to Attach live Debugger

Create a new Debug Configurations selecting Single Application Debug. Choose the Debug type to be *Attach to running target* and the Connection *Local*:

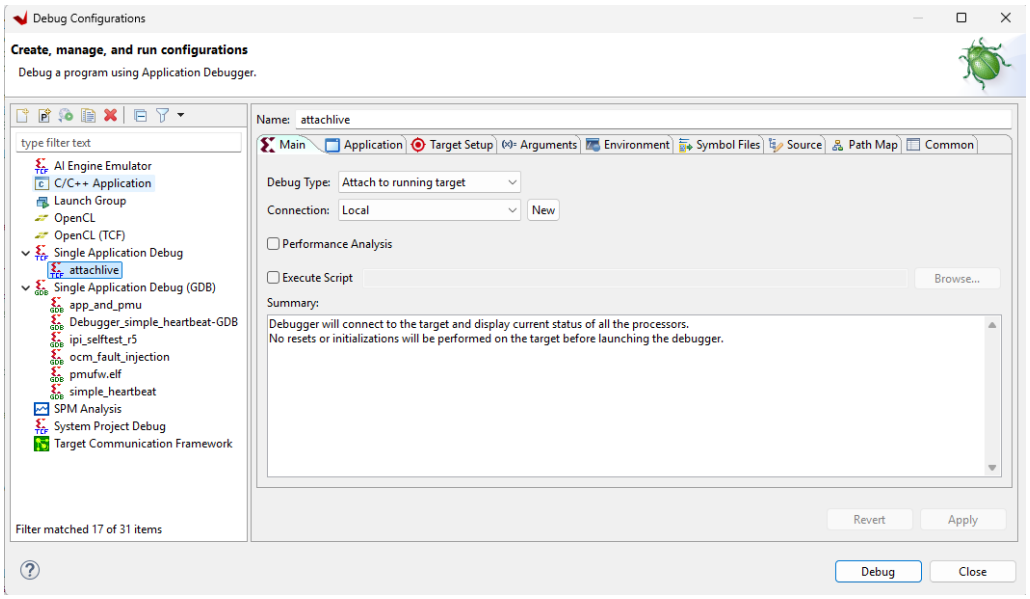

Figure D.1: Creating new Debug Configuration

In the Symbol Files tab press *Add* and add the path to the .elf application that is going to be executed:

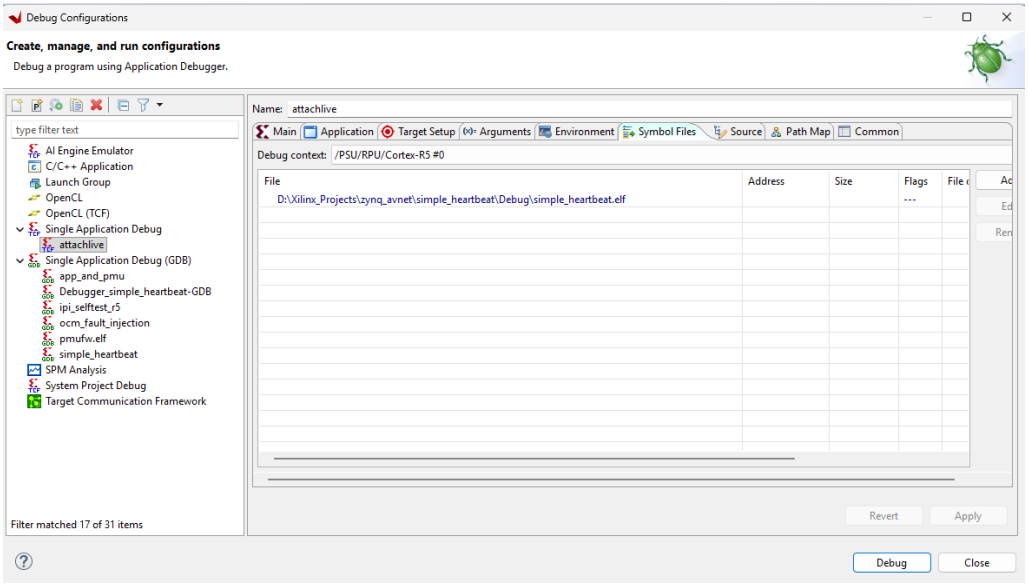

Figure D.2: Path to the .elf executable

In the Path Map tab press *Add* and import the path to the application's root folder:

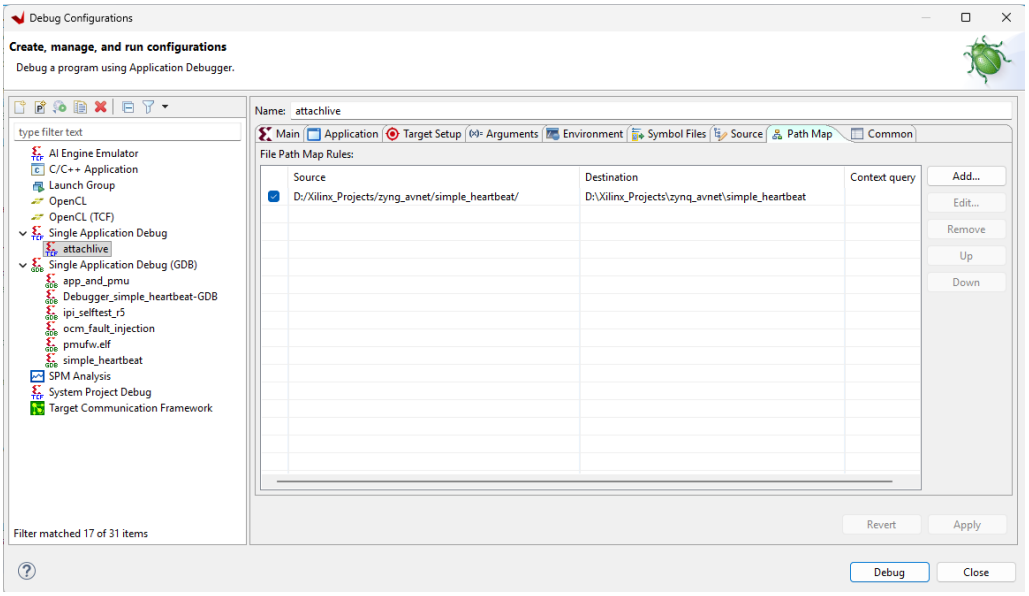

Figure D.3: Path to the application's root folder

Run or Debug the application and switch , if it is not done automatically , to Debug view. For this example the application boots from the SD card . You should be able to see something like this :

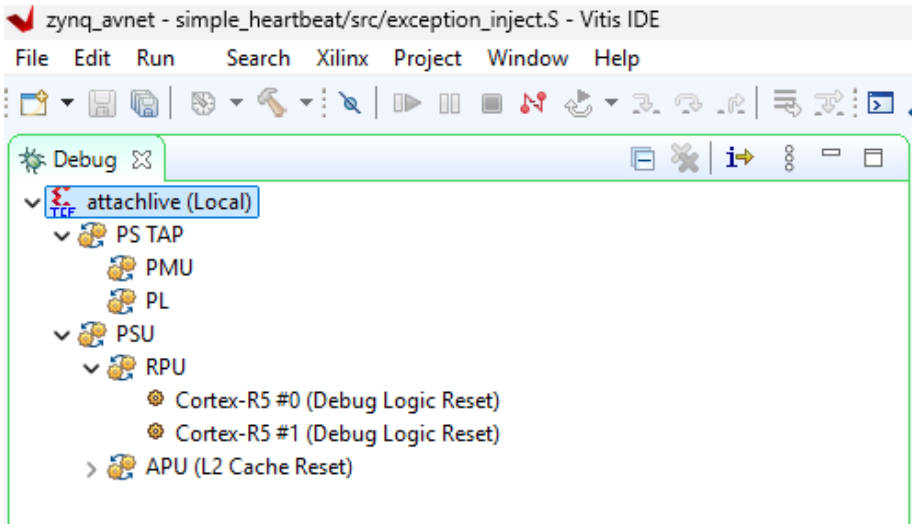

Figure D.4: Debug View : RPU in Debug Logic Reset

If we try to connect from the xsct console we will also view that the RPU is help in Debug Logic Reset :

```
xsct% connect
tcfchan#2
xsct% targets
  1  PS TAP
     2  PMU
     3  PL
  5* PSU
     6  RPU
        7  Cortex-R5 #0 (Debug Logic Reset)
        8  Cortex-R5 #1 (Debug Logic Reset)
     9  APU (L2 Cache Reset)
       10  Cortex-A53 #0 (APU Reset)
       11  Cortex-A53 #1 (APU Reset)
       12  Cortex-A53 #2 (APU Reset)
       13  Cortex-A53 #3 (APU Reset)
xsct%

xsct%
```

Figure D.5: xsct console : RPU in Debug Logic Reset

To overcome this issue and be able to suspend the rpu we must clear the `RST_LPD_DBG (CRL_APB)` register. So from the available targets we select the PSU and run the command `mwr 0xFF5E0240 0x0` :

```
xsct% targets 5
xsct% targets
  1  PS TAP
     2  PMU
     3  PL
  5* PSU
     6  RPU
        7  Cortex-R5 #0 (Debug Logic Reset)
        8  Cortex-R5 #1 (Debug Logic Reset)
     9  APU (L2 Cache Reset)
       10  Cortex-A53 #0 (APU Reset)
       11  Cortex-A53 #1 (APU Reset)
       12  Cortex-A53 #2 (APU Reset)
       13  Cortex-A53 #3 (APU Reset)
xsct% mwr 0xFF5E0240 0x0
xsct% targets
  1  PS TAP
     2  PMU
     3  PL
  5* PSU
     6  RPU
        7  Cortex-R5 #0 (Running)
        8  Cortex-R5 #1 (Lock Step Mode)
     9  APU (L2 Cache Reset)
       10  Cortex-A53 #0 (APU Reset)
       11  Cortex-A53 #1 (APU Reset)
       12  Cortex-A53 #2 (APU Reset)
       13  Cortex-A53 #3 (APU Reset)
xsct%
xsct%
```

Figure D.6: Clear `CRL_APB` register

And now we are able to suspend the R5_0 core :

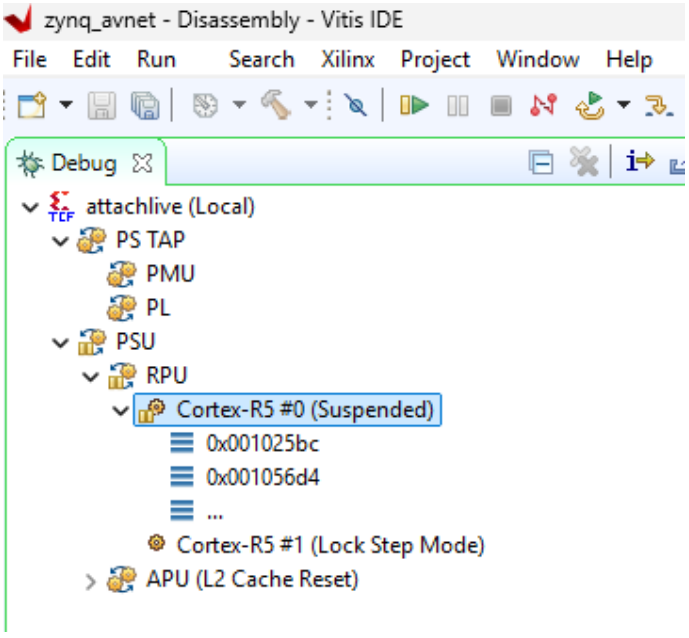

Figure D.7: Pause of thr R5_0 core

Bibliography

[1] Amazon Web Services. *FreeRTOS: Operating System for Microcontrollers*. https://aws.amazon.com/freertos/. 2023.

[2] Amazon Web Services, Inc. *FreeRTOS Kernel*. https://www.freertos.org/. Version 10.4.3. 2023.

[3] AMD Xilinx. *Zynq UltraScale+ Device Technical Reference Manual*. UG1085 v2.4. 2023. URL: https://docs.amd.com/go/en-US/ug1085-zynq-ultrascale-trm.

[4] AMD-Xilinx. *Zynq UltraScale+ MPSoC Software Developer Guide*. Version v2024.2. UG1137. Advanced Micro Devices, Inc. Jan. 2024. URL: https://docs.amd.com/go/en-US/ug1137-zynq-ultrascale-mpsoc-swdev-en-us-2024.2-1.pdf.

[5] Jordan D. Anderson, Jennings C. Leavitt, and Michael J. Wirthlin. "Neutron Radiation Beam Results for the Xilinx UltraScale+ MPSoC." In: *2018 IEEE Radiation Effects Data Workshop (REDW)*. 2018, pp. 1–5. DOI: 10.1109/NSREC.2018.8584297.

[6] *Arm Cortex-R5 Technical Reference Manual*. DDI0460C. Revision r1p1. Arm Ltd. 2018.

[7] ARM Limited. *ARM Cortex-R5 and Cortex-R5F Technical Reference Manual*. ARM DDI 0460C. ARM Limited. Feb. 2011.

[8] ARM Limited. *ECC Schemes for ARM Cortex-R5 Processor*. Tech. rep. Appendix C of ARM DDI 0460C. ARM Limited, 2011.

[9] Arm Limited. *Arm GNU Toolchain*. https://developer.arm.com/Tools%20and%20Software/GNU%20Toolchain. Version 12.2.0. 2023.

[10] ARM Ltd. *ARM Cortex-R5 Technical Reference Manual*. DDI0460C. 2011. URL: https://developer.arm.com/documentation/ddi0460/c.

[11] A. Avizienis et al. "Basic concepts and taxonomy of dependable and secure computing." In: *IEEE Transactions on Dependable and Secure Computing* 1.1 (2004), pp. 11–33. DOI: 10.1109/TDSC.2004.2.

[12] Avizienis and Kelly. "Fault Tolerance by Design Diversity: Concepts and Experiments." In: *Computer* 17.8 (1984), pp. 67–80. DOI: 10.1109/MC.1984.1659219.

[13] Inc. Avnet. *Ultra96-V2 Single Board Computer Hardware User's Guide*. Tech. rep. LIT# Ultra96-V2-HW-User-Guide-v1.3. 2021. URL: https://www.avnet.com/wps/wcm/connect/onesite/b85b9556-0b2a-42b3-ad6a-8dcf3eac1ff9/Ultra96-V2-HW-User-Guide-v1_3.pdf.

[14] P. Balasubramanian, K. Prasad, and Nikos E. Mastorakis. "A Fault Tolerance Improved Majority Voter for TMR System Architectures." In: *CoRR* abs/1605.03771 (2016). arXiv: 1605.03771. URL: http://arxiv.org/abs/1605.03771.

[15] Robert C. Baumann. "Radiation-Induced Soft Errors in Advanced Semiconductor Technologies." In: *IEEE Trans. on Device and Materials Reliability* 5.3 (2005), pp. 305–316. DOI: 10.1109/TDMR.2005.853449.

[16] Manuel Cabañas-Holmen et al. "Predicting the Single-Event Error Rate of a Radiation Hardened by Design Microprocessor." In: *IEEE Transactions on Nuclear Science* 58 (2011), pp. 2726–2733. URL: https://api.semanticscholar.org/CorpusID:29850237.

[17] Cardinal Peak. *What is FreeRTOS and What is it Good For?* https://www.cardinalpeak.com/blog/what-is-freertos-and-what-is-it-good-for. 2021.

[18] Louise Crockett et al. *Exploring Zynq MPSoC: With PYNQ and Machine Learning Applications*. English. Strathclyde Academic Media , eBook (Online Edition), Apr. 2019. ISBN: 0992978750.

[19] Louise H. Crockett et al. *Exploring Zynq MPSoC: With PYNQ and Machine Learning Applications*. Strathclyde Academic Media, 2019. ISBN: 978-0992978754.

[20] *DO-178C: Software Considerations in Airborne Systems and Equipment Certification*. RTCA, 2011.

[21] *DO-254: Design Assurance Guidance for Airborne Electronic Hardware*. RTCA, 2000.

[22] R. Dobrin, S. Punnekkat, and H. Hansson. "ISO 26262 ASIL-Oriented Hardware Design Framework for Safety-Critical Automotive Systems." In: *2019 IEEE International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA)*. Aug. 2019, pp. 1–10.

[23] R. C. Harrington et al. "Exploiting SEU Data Analysis to Extract Fast SET Pulses." In: *IEEE Transactions on Nuclear Science* 66.6 (2019), pp. 932–937. DOI: 10.1109/TNS.2019.2913498.

[24] M.-C. Hsueh, T. K. Tsai, and R. K. Iyer. "Fault Injection Techniques and Tools." In: *IEEE Computer* 30.4 (Apr. 1997), pp. 75–82.

[25] *IEC 61508: Functional Safety of Electrical/Electronic/Programmable Electronic Safety-Related Systems*. International Electrotechnical Commission, 2010.

[26] *ISO 26262: Road Vehicles – Functional Safety*. International Organization for Standardization, 2018.

[27] Daisuke Kobayashi. "Scaling Trends of Digital Single-Event Effects: A Survey of SEU and SET Parameters and Comparison With Transistor Performance." In: *IEEE Transactions on Nuclear Science* 68.2 (2021), pp. 124–148. DOI: 10.1109/TNS.2020.3044659.

[28] Tuo Li et al. "Processor Design for Soft Errors: Challenges and State of the Art." In: *ACM Comput. Surv.* 49.3 (Nov. 2016). ISSN: 0360-0300. DOI: 10.1145/2996357. URL: https://doi.org/10.1145/2996357.

[29] Linaro. *96Boards Consumer Edition Specification*. https://www.96boards.org/specifications/. 2023.

[30] J. Mach et al. "Lockstep Replacement: Fault-Tolerant Design." In: *IEEE Transactions on Computers* 74.2 (Feb. 2025), pp. 437–450.

[31] S. McNeil. *Isolation Methods in Zynq UltraScale+ MPSoCs*. XAPP1320 (v1.1). Xilinx. May 2020.

[32] S. Mitra et al. "Logic soft errors in sub-65nm technologies design and CAD challenges." In: *Proceedings. 42nd Design Automation Conference, 2005.* 2005, pp. 2–4. DOI: 10.1145/1065579.1065585.

[33] Shubu Mukherjee. *Architecture Design for Soft Errors*. Morgan Kaufmann Publishers/Elsevier, 2008. ISBN: 9780123695291. URL: https://books.google.gr/books?id=nh1GlAEACAAJ.

[34] Shubu Mukherjee. "Chapter 1 - Introduction." In: *Architecture Design for Soft Errors*. Burlington: Morgan Kaufmann, 2008, pp. 1–41. ISBN: 978-0-12-369529-1. DOI: https://doi.org/10.1016/B978-012369529-1.50003-3. URL: https://www.sciencedirect.com/science/article/pii/B9780123695291500033.

[35] OSRTOS. *FreeRTOS - Open Source Real-Time Operating System*. https://www.osrtos.com/rtos/freertos/. 2023.

[36] Emre Ozer et al. "Error Correlation Prediction in Lockstep Processors for Safety-Critical Systems." In: *2018 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. 2018, pp. 737–748. DOI: 10.1109/MICRO.2018.00065.

[37] Behrooz Parhami. *Dependable Computing: A Multilevel Approach*. Draft, forthcoming book. Santa Barbara, CA, USA: Publisher TBD, TBD. URL: http://www.ece.ucsb.edu/~parhami.

[38] Michael Rogenmoser et al. "Hybrid Modular Redundancy: Exploring Modular Redundancy Approaches in RISC-V Multi-core Computing Clusters for Reliable Processing in Space." In: *ACM Trans. Cyber-Phys. Syst.* 9.1 (Jan. 2025). ISSN: 2378-962X. DOI: 10.1145/3635161. URL: https://doi.org/10.1145/3635161.

[39] SlideShare Contributor. *Memory ECC - The Comprehensive of SEC-DED*. SlideShare. Accessed: December 25, 2025. 2017. URL: https://www.slideshare.net/slideshow/memory-ecc-the-comprehensive-of-secded/82022057.

[40] I. Sommerville. *Software Engineering*. International Computer Science Series. Pearson, 2011. ISBN: 9780137053469. URL: https://books.google.gr/books?id=l0egcQAACAAJ.

[41] Synopsys Inc. *ASIL D ISO 26262 Compliance for Automotive SoCs*. Technical Article. July 2022. URL: https://www.synopsys.com/articles/asil-d-systematic-iso-26262-compliance.html.

[42] Synopsys Inc. *Error Correction Code (ECC) in DDR Memories*. Technical Article. Oct. 2020. URL: https://www.synopsys.com/articles/ecc-memory-error-correction.html.

[43] *Ultra96-V2 Single Board Computer Hardware User's Guide*. Version 1.3. Avnet, Inc. 2021.

[44] *Vitis Unified Software Platform Documentation: Embedded Software Development*. UG1400. Version 2022.2. AMD Xilinx. 2022.

[45] *Vivado Design Suite User Guide: Getting Started*. UG910. Version 2022.2. AMD Xilinx. 2022.

[46] Dagan White. *Considerations Surrounding Single Event Effects in FPGAs, ASICs, and Processors*. White Paper WP402. Version v1.0.1. Xilinx, Mar. 2012. URL: http://www.xilinx.com.

[47] Wikipedia contributors. *ECC memory*. Wikipedia, The Free Encyclopedia. Accessed: December 2025. URL: https://en.wikipedia.org/wiki/ECC_memory.

[48] Wikipedia contributors. *Redundancy (engineering)*. Wikipedia, The Free Encyclopedia. Accessed: December 2025. URL: https://en.wikipedia.org/wiki/Redundancy_(engineering).

[49] Xilinx Inc. *PMU Firmware Guide*. Xilinx Inc. 2019. URL: https://xilinx-wiki.atlassian.net/wiki/spaces/A/pages/18842508/PMU+Firmware.

[50] Xilinx Inc. *Zynq UltraScale+ MPSoC Register Reference*. UG1087. Xilinx Inc. 2022.

[51] Xilinx, Inc. *Zynq UltraScale+ MPSoC Technical Reference Manual*. Version v2.4. December 21, 2023. 2023. URL: https://docs.xilinx.com/v/u/en-US/ug1085-zynq-ultrascale-trm.

[52] *Zynq UltraScale+ Device Technical Reference Manual*. v2.4. UG1085. Accessed online. AMD Xilinx. Dec. 2023.

[53] *Zynq UltraScale+ MPSoC Data Sheet: Overview*. DS925 (v1.6). AMD/Xilinx. May 2024. URL: docs.amd.com.

[54] *Zynq UltraScale+ MPSoC Processing System Configuration*. Vivado IP Catalog Documentation, Version 2022.2. AMD Xilinx. 2022.

[55] *Zynq UltraScale+ MPSoC Software Developer Guide*. v2024.2. UG1137. AMD Xilinx. Jan. 2024.