ALMA MATER STUDIORUM
UNIVERSITÀ DI BOLOGNA

DEPARTMENT OF
ELECTRICAL, ELECTRONIC, AND INFORMATION ENGINEERING
"GUGLIELMO MARCONI" - DEI

SECOND CYCLE DEGREE

ELECTRONICS FOR INTELLIGENT SYSTEMS, BIG-DATA
AND INTERNET OF THINGS
INGEGNERIA ELETTRONICA

TLB-ECC OPTIMIZATION FOR CVA6 RELIABILITY

Dissertation in
LAB of Digital Electronics M

Supervisor

Prof. Davide Rossi

Co-Supervisors

Riccardo Tedeschi
Dr. Yvan Tortorella

Defended by

Atena Ehsanfar

Graduation Session / July/ 2025

Academic Year 2024/2025

Acknowledgements

First, I would like to sincerely thank Prof. Luca Benini for giving me the opportunity to work within his research group. Being part of such a professional and inspiring academic environment has been a valuable experience during my thesis journey, and I truly appreciate his support throughout the development of this work.

I would like to express my profound appreciation to Prof. Davide Rossi for granting me the opportunity to carry out my thesis within his professional research team and for proposing a compelling and meaningful topic that closely aligned with my research interests.

I am especially grateful to my co-supervisors, Riccardo Tedeschi and Dr. Yvan Tortorella, whose constant support, insightful feedback, and technical guidance were truly essential at every step of this journey. Thank you for your patience and for always being there whenever I needed guidance.

I extend my heartfelt appreciation to my family, especially my sisters, Azita and Aida, for their love and support throughout this period, particularly during the times I struggled with illness and the uncertainty of completing my degree on time. Although they were far away, they have always remained close to my heart, and their continuous encouragement gave me the strength to persevere and stay focused on my goal.

I am also deeply thankful to my friends, Amirhossein and Mahsa, for being by my side through both the joyful and challenging moments of this journey. Your support, thoughtful presence, and willingness to simply listen made even the hardest days easier to face.

I am especially grateful to Alberto, whom I met during one of the most difficult times in my life. Your presence brought light into a heavy year; you stood by me with patience, care, and belief when I needed it most, and for that, I am truly thankful.

Abstract

Translation Lookaside Buffer (TLB) is an essential part of the memory management unit (MMU) that speeds up virtual-to-physical address translation. Due to its frequent access, it is especially prone to soft errors, which can lead to incorrect translations, degraded system performance, or even critical failures.

Starting from a baseline TLB extended with suboptimal Single-Error Correction, Double-Error Detection (SECDED) mechanism, this thesis proposes alternative architectural solutions that preserve the TLB reliability capabilities while improving the TLB area and timing metrics at the cost of slower correction mechanism due to TLB entries invalidation in case of fault detection during translation process.

This work ensures real-time delivery of corrected data to the MMU while persistently updating TLB entries to enhance reliability, reduce correction overhead, and limit soft error propagation. Additionally, performance is improved by removing TLB and ECC logic from the MMU's critical path, easing timing constraints during translation.

To achieve this aim, five solutions have been implemented: Detection Only mode, Full parallel correction using separate encoder per each TLB entry, Counter-based solution for correction, Arbiter-based solution for correction and the last solution is about Performance Enhancement based on using arbiter and counter, separately. The proposed TLB modifications are implemented within a 64-bit RISC-V core named CVA6 capable of booting Linux, integrated into an open-source SoC called Cheshire. The design was validated through PPA analysis on GlobalFoundries 22nm (GF22) technology. Based on the result, there is a 5.3% and 8.14% increment in frequency for the arbiter-based correction mechanisms and counter-based solutions including performance enhancement respectively, compared to the baseline TLB.

However, a negligible increase in CVA6 area has been observed for both arbiter-based (1.25%) and counter-based solutions (0.41%).

Contents

1.Introduction

As embedded systems, low-power computing platforms, and high-performance processors become increasingly widespread across industries such as automotive, aerospace, healthcare, and consumer electronics, ensuring the reliability of these systems has emerged as a crucial design concern.

At the same time, the RISC-V has seen rapid adoption in safety-critical domains, thanks to its open Instruction Set Architecture (ISA) and readily available open-source processor IP cores [1].

A major advantage of RISC-V is its flexibility, which allows customization of both ISA and hardware to satisfy specific reliability requirements. As a result, RISC-V is gaining traction in systems where dependable operation is essential [2].

Furthermore, as CMOS technology advances, driven by the need for greater functionality, lower power consumption, ever-smaller transistor dimensions and operating voltages, significant challenge emerges, particularly in memory applications. The escalating demand for on-chip storage has led to increasingly compact embedded memories. However, this technological scaling, combined with reduced supply voltages, makes memory cells highly susceptible to radiation-induced soft errors, compromising their reliability [3].

Ensuring memory system reliability involves not only correcting transient errors but also maintaining performance and energy efficiency. To this end, designers commonly integrate Error-Correcting Codes (ECC) such as Hamming codes, into memory controllers, which detect and correct errors. Over time, however, memories may accumulate faults, necessitate periodic re-reading and scrub to prevent multiple error accumulation. In such designs, memory controllers primarily focus on high data throughput, relying heavily on ECC logic and delegating more complex fault-tolerance tasks to the operating system [4].

Also, the Translation Lookaside Buffer (TLB) emerges as one of the vulnerable architectural structures. Acting as a cache for recent virtual-to-physical address translations, the TLB is accessed frequently during memory operations which result in being particularly susceptible to Single Event Upsets (SEUs). When soft errors occur within TLB, they can lead to hard faults, silent data corruption, or even system-wide freeze by corrupting translation entries [5].

A single-bit error in an entry can alter the stored virtual page number, potentially leading to incorrect matching behavior. This alteration can result in false negatives, where legitimate entries are missed. More critically, it can cause false positives, where an incorrect match is made. Such errors can have severe consequences, including silent data corruption, system crashes, or hard faults, as the processor may execute unintended instructions instead of the correct ones [6].

Different methods have been proposed to protect TLBs against errors. When by using a parity bit [7] single-bit faults in the TLB can be detected. Upon detection of an error, the corresponding TLB entry is invalidated which as result memory management unit (MMU) is supposed to feed the TLB with correct data again. Since this recovery operation can be costly in terms of performance, more efficient approaches such as ECCs have been introduced [8], [9].

To mitigate the delay caused by error checking and correction, one solution [8] employs ECC in a backup copy and uses periodic scrubbing to refresh and correct errors. Another alternative [9] adapts ECC logic to enable fast error detection, initiating the correction process only when an error is confirmed or using scrubbing mechanisms to maintain reliability [6].

However, traditional ECC implementations often focus on transient correction only: they correct errors combinationally during read access but do not update the corrected values in TLB. As a result, the same error must be re-corrected each time the entry is accessed, which leads to several inefficiencies like Redundant decoding cycles, Increased energy consumption, and Critical path delay due to ECC logic operating in line with the read pipeline.

This thesis addresses these challenges by proposing a set of architectural enhancements for TLB extended with ECC which is the baseline TLB for this work. These enhancements ensure that corrected entries are persistently updated in TLB entries, as well, thereby eliminating repeated correction cycles and reducing ECC logic pressure.

The methodology is built upon five distinct strategies:

1. **Detection-Only Mode**,
   Having no correction and just invalidate the detected errored entries
2. **Full parallel correction using separate encoder per each TLB entry**,
   Having capability of correcting several entries with 1-bit error simultaneously
3. **Counter-based correction**,
   Using counter module and a finite state machine (FSM) to sequentially scan and correct corrupted entries using the decoder's output.
4. **Round-robin arbiter-based correction**,
   Using round-robin arbiter module to select the entry with 1-bit error to be corrected
5. **Performance Enhancement techniques applied to the latter two**.
   Bypassing ECC when MMU wants to read TLB to improve the timing, Since ECC decoding and correction paths lie directly on the critical path. During this phase, no correction is allowed

In this work an open-source RISC-V CVA6- based (64-bit) Cheshire SoC has been used, and an analysis of mentioned solutions have been done on GlobalFoundries 22nm (GF22) technology under the slow-slow (SS) process corner at 0.72 V and –40 °C.

2. State of the Art

2.1 Fault Taxonomy and System Vulnerabilities

Faults are the underlying cause of errors and failures in digital systems. A fault is latent defect or anomaly in hardware or software [10]. When a fault alters the system's internal state in an unintended way, it results in an error. If the error propagates and causes the system to deviate from its specified behavior, it leads to failure [10].

2.1.1 Classification of Faults in Hardware Systems

Faults in hardware systems can be classified by duration and stability, behavioral impact and physical origin and manifestation.

a) Duration and Stability

Permanent faults are persistent issues in electronic systems that require physical repair or replacement to resolve. They can occur due to the aging of components over time or stem from defects introduced during manufacturing [10]. A common type is a stuck-at fault, which can result from problems like electromigration, where the continuous flow of electrons gradually moves atoms in a conductor, eventually leading to open circuits or shorts [10], [11].

Transient faults are temporary disruptions in electronic systems. commonly referred to as soft errors, may result in a bit flip within a combinational circuit, known as a Single Event Transient (SET) or a bit alteration in a memory element, termed a Single Event Upset (SEU) [12]. One common reason is the effect of radiation, which modern systems are more vulnerable to, because of the continuous scaling down of semiconductor technology [13], [14].

Depending on the amount of charge disturbance brought in by the radiation event it can create SEUs, where cosmic rays or alpha particles momentarily flip bits in memory cells without causing permanent damage or as more complex variant is multi-bit Upsets (MBUs), where a single energetic particle can simultaneously alter multiple bits within the same data word, particularly in SRAM or DRAM [15].

As CMOS technology continues to scale down, the overall soft error rate in microprocessors is shaped by two opposing trends. On one hand, the SEU rate per bit in sequential cells tends to decrease due to reduced cell size and lower supply voltage. On the other hand, the total number of bits in modern processors is rapidly increasing due to architectural advancements like larger caches, expanded memory, and multicore integration. As a result, even though each individual bit becomes less susceptible, the total likelihood of soft errors across the entire processor rises with each generation [16].

Beyond radiation, electrical noise, crosstalk, electromagnetic interference (EMI), and power disturbances can also induce transient errors. Furthermore, environmental effects, such as extreme temperatures, can contribute to these faults by leading to issues like leakage, threshold drift, or unstable timing margins within the system [17], [18], [19].

Intermittent faults lead the system to alternate unpredictably between correct behavior and malfunctioning states [20].

b) Effect on System Behavior

The effect of faults on system behavior can vary significantly depending on their nature. Benign faults typically render a unit inactive or stuck in a known state, leading to predictable system behavior [10]. These faults are often recoverable through simple mechanisms like reset or redundancy. In contrast, malicious faults are far more complex and dangerous. They produce misleading or contradictory outputs-such as a sensor sending inconsistent values to different parts of the system-which can undermine trust and system integrity [10].

c) Physical faults

They can originate from both internal and external sources. Internally, they may result from gradual physical degradation, such as shifts in transistor threshold voltage, aging effects, or physical defects like short circuits and open connections [21]. Externally, these faults can be triggered by environmental stressors including electromagnetic interference (EMI), mechanical vibrations, radiation (such as cosmic rays) [21], or extreme temperature variations. In addition to these naturally occurring issues, faults may also be introduced by human activity. These human-made faults include design-related errors such as flawed logic, layout inaccuracies, or incomplete specifications, as well as operational mistakes like improper usage, incorrect maintenance procedures, or configuration error in hardware or firmware [22].

2.1.2 Safety Standards

Safety standards play a crucial role in guiding the design and validation of fault-tolerant embedded systems. Among these, IEC 61508:2010 [23] serves as a foundational framework for the functional safety of electrical, electronic, and programmable electronic systems, defining safety integrity levels (SILs) based on reliability metrics and diagnostic coverage to prevent system failure under predefined conditions. Building upon this, ISO 26262:2018 [24] tailors IEC 61508 to the automotive domain, introducing Automotive Safety Integrity Levels (ASILs) that consider severity, exposure, and controllability to assess the risks associated with each function. In the realm of microcontroller-based consumer and industrial devices, IEC60730 [25] sets out requirements for automatic electrical controls used in households and similar applications. This standard ensures embedded systems include self-diagnostic and fault response mechanisms, especially for safety-critical tasks such as appliance regulation or motor control. [25] highlights that the integration of such safety frameworks is not only necessary for meeting regulatory compliance but also for enabling dependable operation of modern, software-intensive embedded systems like autonomous platforms.

Another important standard relevant to low-level device safety is IEC 60747- a series of specifications for the design, fabrication, and reliability testing of semiconductor devices, including integrated circuits. It outlines parameters such as electrical characteristics, thermal behavior, electromagnetic compatibility (EMC), and environmental conditions, all of which significantly influence the performance and lifespan of integrated components. For insights into reliability assessment strategies and stress testing protocols, particularly in power semiconductors, relevant studies on humidity reliability are often referenced [26].

2.2 Fault Tolerance Techniques

Ensuring fault tolerance is essential, especially for mission-critical applications. Although faults are often transient, commonly referred to as soft errors rather than permanent defects, they can still significantly disrupt system behavior, potentially leading to malfunctions or crashes in contemporary electronic devices [27].

To mitigate these risks, researchers have developed a wide range of fault-tolerant strategies. These approaches span from modifications in chip materials and fabrication processes to various design-level solutions, including hardware-based, software-based, or combine both like hybrid techniques [28], [29].

2.2.1 Hardware based solutions

a) Technological Techniques

At the lowest abstraction level, **Radiation-Hardened (Rad-Hard) technologies** provide fault tolerance by modifying the silicon-level design to resist radiation-induced soft errors. These techniques operate at the semiconductor fabrication level and include process-based methods such as doping adjustments and specialized materials [14], [21].

While Rad-Hard solutions effectively reduce the need for architectural fault tolerance in application-specific SoCs, they are often limited by factors such as high cost, reduced availability, and significant increases in CVA6 area. Furthermore, these approaches are commonly associated with older technology nodes, making them less compatible with high-performance, modern chip designs despite their improved resilience to soft errors [19].

Building on this foundation, **layout-based hardening techniques** like **LEAP-DICE** further enhance soft error resilience by optimizing transistor placement and spatial separation to mitigate multi-bit upsets (MBUs). These methods focus on improving immunity through physical layout design rather than changes in manufacturing processes, offering a balance between reliability and design flexibility [16], [30], [27].

b) Circuit-Level Techniques

At the circuit structure level, **redundant cell architectures** such as **DICE**, **BISER**, **SEILA**, and **BCDMR** have been extensively employed to improve fault tolerance in memory elements. The **DICE** (Dual Interlocked storage Cell) design uses interlocked feedback among four or more storage nodes to prevent single-event upsets from altering stored data.[31] **BISER** (Built-in Soft Error Resilience) and **BCDMR** (Built-in Current-Based Detection and Masking Redundancy) enhance robustness through scan-latch-based redundancy and current-mode logic [32]. Similarly, **SEILA** utilizes self-error detection with integrated latch architecture to improve recovery from soft errors [16].

Finally, **node-level hardening techniques**, including **resistive and capacitive hardening**, are applied at the electrical property level of memory and logic circuits. These approaches strengthen the charge-holding capacity of storage nodes or introduce resistance to stabilize bit values, making them less susceptible to transient faults. While effective in increasing resilience, such fine-grained modifications can lead to added design complexity and potential trade-offs in speed and power efficiency. [16]

c) Microarchitectural Techniques

One of the techniques of this category is **Hardware redundancy** which involves duplicating or triplicating system components to enhance reliability, typically used where failure is unacceptable. While effective, it increases area, power, and cost. It comes in three forms: static (all units operate in parallel), dynamic (spares activate upon failure), and hybrid (a combination of both for balanced fault tolerance) [10].

Modern systems adopt hardware redundancy at multiple design levels, particularly through modular redundancy strategies. Multi-core and many-core processors offer a high performance and energy-efficient computing solution, but their increased sensitivity to soft errors due to miniaturization raises reliability concerns. To address this, a fault-tolerant strategy named N-Modular Redundancy and M-Partitions (NMR-MPar) is introduced [26]. This method leverages both redundancy and partitioning to ensure reliable operation in mixed-critical systems. By dividing tasks across different partitions that execute independently, NMR-MPar enhances fault isolation and allows critical functions to run securely on multi-/many-core platforms [26]. Commonly used approaches are Double Modular Redundancy (DMR) and Triple Modular Redundancy (TMR). In these schemes, multiple copies of a critical module process the same inputs, and a voter circuit determines the correct output. This ensures correct execution even in the presence of faults and can be implemented at varying levels of granularity. Meanwhile, Hybrid Modular Redundancy (HMR) combines traditional TMR and DMR techniques, selectively deploying them across different parts of a RISC-V multi-core cluster. For example, safety-critical cores may use TMR with majority voting, while less critical subsystems use DMR. This strategy has been adopted in fault-tolerant RISC-V clusters for aerospace applications due to its balance between fault coverage and efficiency [19].

As an example of the methods mentioned above, it can be pointed out fine-grained TMR within a RISC-V core, as demonstrated in the STRV project [2], replicates circuitry and performs voting after each register to ensure correctness. Similarly, Gkiokas and

Schoeberl's design [39] uses TMR across the fetch, decode, and execute stages in a 5-stage dual-issue core, passing results to a majority voter before memory and write-back stages.

To guarantee consistency, the inputs to the duplicated or triplicated cores are tightly synchronized so that each core processes identical data. Their outputs are connected to dedicated checkers or majority voters, which verify correctness and trigger recovery actions if discrepancies are detected. When operating in this configuration, the grouped cores function as a single virtual processing unit within the system.

Although this setup protects the cores themselves, the checkers and majority voters remain susceptible to soft errors. To address this vulnerability, it is assumed that additional safeguards are implemented, such as ECC on interconnect buses, which are monitored at the protocol level within the protected core domain. These measures help ensure the integrity of control, instruction, and data paths, even in the presence of transient faults.

In contrast, coarse-grained modular redundancy involves duplicating entire cores, such as in Dual-Core Lockstep (DCLS) or Triple-Core Lockstep (TCLS) architectures, where entire processor blocks operate in parallel with boundary-level checkers and voters to detect and mask faults [10], [19].

Other structural redundancy models include M-of-N system, which continues to operate correctly as long as at least M out of N modules remain functional [10]. Lastly, duplex systems employ two identical modules in parallel; if a mismatch is detected, the faulty unit is excluded from operation. Although duplexing provides less fault coverage than full NMR, it is a cost-effective solution in systems where detecting single errors with minimal hardware overhead is sufficient [10], [23].

Furthermore, one of the most fundamental strategies for soft error mitigation is **information redundancy**, where additional bits are embedded within the original data stream to enable detection and correction of bit-level faults. Rather than focusing on hardware replication or circuit hardening, this technique employs mathematical encoding to identify inconsistencies in data caused by soft errors like Single Event Upsets (SEUs) [10].

Information redundancy is a widely used strategy to mitigate soft errors in processor storage components, such as caches, memories, and register files. Additionally, such codes are also applied selectively in computation components like arithmetic logic units (ALUs), where bit-level correctness is critical [16].

At its core, information redundancy utilizes coding theory, which formulates how original data can be protected using a combination of data bits and redundant bits. These redundant bits are strategically derived from the original data using logical operations (such as XOR) and can later be used to detect and potentially correct bit errors during readback or data processing [10].

Error-correcting codes are categorized in separable and non-separable codes [10]. Separable codes organize the data and redundancy (parity) bits into clearly distinct fields (Figure 2.1). That is, the original data is preserved in its entirety, and the redundant bits are appended separately. This structure simplifies decoding, as the data can be accessed directly while the redundancy is processed in parallel. Most traditional ECC, including Hamming-based ECC used in memory systems, fall into this category. This coding structure is advantageous in timing-critical components like TLBs, where fast access is crucial and decoding must be efficient to avoid pipeline stalls [10], [7].

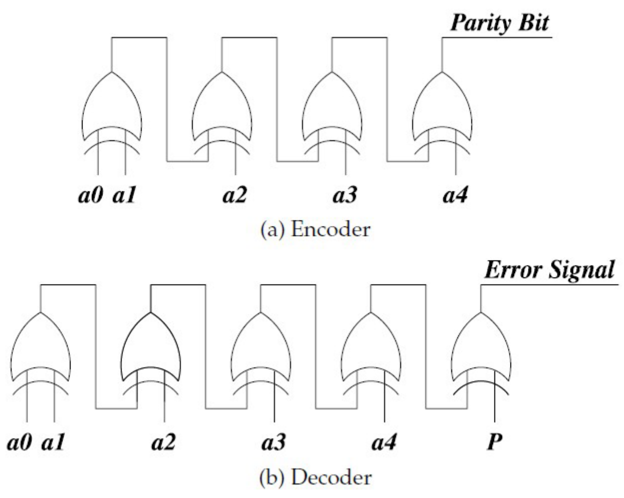


Figure 2.1: Even parity encoding and decoding circuits. Adapted from [10].

Non-separable codes, on the other hand, intermix data and redundancy bits. In such schemes, the encoded word does not explicitly preserve the original data layout. While this can sometimes lead to more compact codes or optimized implementations, it increases decoding complexity since the original data must be reconstructed from the entire codeword [10].

Among the simplest forms of separable coding is the parity bit. A parity code appends a single bit to a data block to make the number of 1's either even (even parity) or odd (odd parity). This allows single-bit errors to be detected by re-computing the parity and comparing it to the stored parity bit [10]. While simple parity is primarily a detection method, more advanced parity-based strategies can also support error correction. For instance, per-byte parity assigns a separate parity bit to each byte, increasing the overhead but improving coverage, especially when errors affect distinct bytes [10]. Overlapping parity schemes, which compute parity across both rows and columns of a data matrix, offer a practical solution for both error detection and correction. They can identify the exact faulty bit at the intersection of affected parity checks, a technique that is especially relevant in RAID architecture and memory arrays [10], [43].

To address the limitations of simple parity and provide correction capabilities, more robust coding schemes like Error-Correcting Codes (ECC) are employed. At the system level, the most widely used techniques for safeguarding memory elements are ECC and parity [56].

d) System-Level Techniques

Here, it can be mentioned to **System-level redundancy** which improves reliability without modifying internal processor architecture. ARM Cortex-R processors support Dual-Core Lockstep **(DCLS)** [40], where two cores run in sync and can switch to independent execution during reset. Fault recovery, however, is left to the surrounding SoC and often relies on resets or checkpointing, which can be problematic for real-time systems like spacecraft computers that demand minimal downtime [41].
To address this, [42] proposed a Triple-Core Lockstep (TCLS) system using Cortex-R5, where an assistance unit coordinates three cores, enabling majority voting and real-time fault correction. The assistance unit uses Rad-Hard technology, while the cores use standard processes with ECC and modular redundancy. When a fault is detected, the affected core resets and reloads its state from memory, ensuring continued execution.
A similar approach is used in the AURIX™ TriCore™ Microcontroller [45] which also applies TCLS and reset-based fault recovery. These techniques improve reliability at the system level without extensive architectural changes [19].

The other solution can be **Data replication** is a fundamental fault-tolerance technique that enhances system reliability by duplicating information, providing physical redundancy as opposed to coding-based methods like ECC [10]. Unlike mathematical redundancy, replication allows direct comparison of data copies and enables fault masking through majority voting or substitution, making it especially suitable for systems with strict timing constraints or legacy architectures where coding is impractical. Complete data replication involves storing two or more identical copies of an entire data block, allowing recovery in case of corruption, though it significantly increases storage requirements and is thus typically reserved for critical control registers or non-time-sensitive backups. In contrast, read-only data replication is used in systems where data is mostly read and seldom updated; here, only a single master copy is written, while reads can be cross-verified among replicas to detect inconsistencies, making it ideal for protecting configuration data or instruction caches [10].

2.2.2 Software-based solutions

Reliability can also be enhanced through software-level solutions. One common approach involves executing the same application on multiple threads—either in duplicate [33], [34], running in parallel or in sequence. Although this method is effective in addressing faults within the data path, it may not detect or recover from faults occurring in the processor's internal control logic, which can cause threads to hang [35], [20].

a) Software redundancy particularly in complex systems that rely heavily on critical embedded software [19]. It offers an essential layer of protection beyond hardware-level safeguards by specifically addressing faults that originate from software bugs, compiler errors, or incorrect logic. Two prominent techniques employed in software redundancy are: N-version programming, where multiple independently developed versions of a program are executed concurrently, and their outputs are then compared. Alternatively, recovery blocks involve a primary algorithm whose result is subjected to an acceptance test; if it fails, a backup algorithm is invoked, allowing for graceful recovery from errors [10], [19].

b) Software-Based Self-Testing (SBST), in addition to runtime redundancy, uses software routines to verify embedded processors by activating and detecting faults through normal execution paths. It's widely applied in automotive-grade microcontrollers and RISC-V cores, especially where external test access is limited. SBST supports at-speed, in-field, and periodic testing, requiring no external equipment, and can detect both permanent and transient faults with high coverage. However, traditional SBST relies on outdated fault models and demands intensive simulations [36]. Recent approaches improve this by using fault-independent techniques and structural insights, enabling automated test generation, better fault coverage, and minimal performance impact, as shown as an example in tests in [37], [36].

c) Predictive checkpointing and **rollback mechanisms** periodically save the processor state, allowing the system to roll back to a valid state when errors are detected [10].

d) Time redundancy is another software-based fault-tolerance technique that relies on re-executing operations to detect or correct transient errors [10]. For example, a computation might be performed multiple times, and the results then compared. This method is particularly effective against transient faults, as these temporary errors are unlikely to recur in the exact same way during a subsequent re-execution. Time redundancy is often combined with information redundancy techniques to further enhance error detection and correction [10], [20].

These redundancy techniques are primarily used in the processor's pipeline components, such as the instruction fetch/decode/execute units, ALUs, or control logic. While these methods introduce performance or area overheads, they are effective for ensuring execution correctness across the full instruction flow [16].

2.3 ECC

As discussed before, ECC belongs to the Information redundancy category and provides a more efficient way to protect static data. It ensures strong error resilience while using significantly less hardware resources [23].

ECC plays a crucial role in embedded systems designed to meet safety standards like IEC 60730 Class C or IEC 61508 SIL2 and above [46]. While it's technically possible to reach these compliance levels without hardware ECC, doing so would require a significant amount of additional software logic [46]. Integrating ECC memory simplifies this process by boosting diagnostic coverage above 90%, making it much easier for systems to satisfy strict safety requirements [46]. Additionally, ECC can enhance security, as it may help detect signs of hardware tampering through unexpected error patterns [46].

The widely used technique in ECC is the Hamming Single Error Correction (SEC) code, which is capable of correcting single-bit errors using simple and fast encoding and decoding logic, all while requiring minimal redundancy. A practical example of this implementation can be found in [47].

Hamming ECC consists of two processes: encoding and decoding. The encoding process with Hamming ECC involves adding the appropriate parity bits to the original data. The Hamming code process begins with a block of data bits, which determines the overall size of the encoded word. Based on specific bit positions, parity bits are computed by checking fixed combinations of the data bits. These parity bits are then inserted into predefined positions within the data block, forming the encoded message that includes both data and redundancy. Once transmitted, the receiver performs error detection by recalculating the parity bits using the same formulation as in the encoding process and comparing them to the received ones. If any mismatch is found, it indicates the presence of a bit of an error. When this occurs, the receiver uses the pattern of mismatched parity bits-known as the syndrome-to identify the position of the faulty bit and correct it (flip it). This mechanism enables single-bit error correction with minimal logic and is the foundation of SEC (Single Error Correction) codes [48], [49].

Extended Hamming codes enhance basic SEC functionality by also providing double-error detection (DED). This is achieved by adding one extra parity bit that covers the entire encoded word, ensuring parity across all bits, including the data and the original Hamming parity bits. A practical implementation example can be found in [50].

The notation commonly used to describe the characteristics of an ECC is written as (n, k, m), where n represents the total length of the code word, k is the length of the original data, and m indicates the number of check bits added to enable error detection and correction [51]. An additional fundamental parameter in ECC theory is the minimum Hamming distance, denoted as d_{min}. This distance is defined as the number of bit positions in which two code words of the same length differ. In practical terms, it reflects the minimum number of bit alterations required to transform one valid code word into another, or alternatively, the fewest number of errors that could corrupt a valid word while still yielding another valid but incorrect word. The maximum number of faults that an ECC scheme can detect is equal to $d_{min} - 1$, while the maximum number of correctable errors is given by the integer part of $(d_{min} - 1)/2$. For instance, if the minimum Hamming distance d_{min} is 0, the code cannot detect or correct any errors. On the other hand, a code with d_{min}= 3 can detect up to two errors and correct one [51].

Hsiao SEC-DED codes are an optimized variant of Extended Hamming codes [52], designed to offer improved efficiency in hardware implementations. These codes are classified as optimal minimum odd-weight column codes, meaning that every column in their parity check matrix contains an odd number of ones. This structure enables reliable double-error detection while maintaining the ability to correct single-bit errors.

One of the key advantages of Hsiao codes lies in their simplified detection logic, which translates into lower latency, reduced silicon area, and decreased power consumption when compared to traditional Hamming SEC-DED codes. Hsiao codes are widely used in the protection of register files, memory arrays, and configuration registers, particularly in systems that require lightweight and fast error correction mechanisms [52]. Hsiao ECC codes have been shown to reduce the latency of conventional Hamming ECC implementations by approximately 8.5% [49].

Following the above discussion, also ECC takes advantage of syndrome decoding. If the syndrome equals zero, no error is detected. If the syndrome is non-zero, its pattern identifies the location of the bit error - allowing the system to correct it in the case of a single-bit fault. In Hamming codes, the syndrome directly maps to the index of the erroneous bit, enabling single-bit error correction. The calculation is commonly performed using a parity-check matrix in modulo-2 arithmetic [10].

Additionally, a global parity bit-computed over the entire codeword-is included in some ECC designs to determine whether the total parity has been altered. If the syndrome indicates an error but the global parity remains valid, this discrepancy suggests the presence of an uncorrectable multi-bit error that did not affect overall parity, helping to prevent mis-correction and ensuring system reliability [10].

To illustrate a real-world application of ECC, the RISC-V Rocket and BOOM processor cores were enhanced with a configurable ECC-protected memory component [53]. In this implementation, ECC plays a critical role by detecting and correcting single-bit errors and detecting double-bit errors in various memory structures, including caches and buffers. The integration of ECC ensures continuous system operation without data corruption, all while maintaining performance and keeping hardware overhead within acceptable limits [53]. As another application, it can be mentioned to the paper by [19] demonstrates the use of ECC in space engineering to protect memory systems from radiation-induced errors. ECC is employed to detect and correct faults caused by cosmic rays, enhancing system reliability. Results show that ECC significantly improves fault tolerance, making it essential for reliable operation in space environments [19]. Also, ECC deployment is shown in the work by [54] where an error detection and correction system were implemented for semiconductor memory applications. The design includes both encoder and decoder components, demonstrating how ECC can significantly enhance system reliability and speed while reducing resource usage.

In addition to its widespread use in memory arrays and processor caches, ECC plays a vital role in protecting structures such as the Translation Lookaside Buffer (TLB), which is responsible for accelerating virtual-to-physical address translation in modern processors. Given the high access frequency and critical role of TLB entries, even single-bit errors can lead to severe system faults or silent data corruption. Integrating ECC within the TLB enables detection and correction of such errors with minimal performance overhead. Hsiao codes are used to safeguard TLB entries, ensuring immediate correction of one-bit errors through syndrome-based logic while supporting detection of more severe faults. This approach enhances overall system reliability without requiring costly duplication or complex rollback mechanisms.

The next chapter will begin by introducing the concept and functionality of the TLB, followed by a discussion on how ECC can be applied to enhance its reliability.

3. Background

3.1 TLB

To introduce Translation Lookaside Table, it is needed to introduce first Virtual Memory and Virtual Machines.

At any given moment, modern computers typically executing multiple processes, each utilizing their own independent address space. Allocating a complete address space worth of physical memory to every process would be highly resource-intensive, especially since many processes only occupy a small portion of their allocated address space. To address this inefficiency, systems employ virtual memory, which segments physical memory into blocks and distributes these segments among active processes. This method necessitates a protection mechanism to ensure that each process can only access the memory blocks assigned to it, preserving memory isolation and security [20]. A key benefit of virtual memory is its ability to reduce the startup time of programs, as it eliminates the need to load all program data and code into physical memory before execution begins [20]. Figure 3.1 illustrates how virtual memory pages are mapped to physical memory for a program divided into four pages. Beyond efficient memory sharing and protection, virtual memory also facilitates the process of relocating programs. This feature, known as relocation, enables a program to be executed from any location within physical memory. The program's placement can be dynamically adjusted-whether in physical memory or on disk-simply by altering the virtual-to-physical address mapping. Historically, before virtual memory became widespread, processors relied on relocation registers to achieve this flexibility [20].

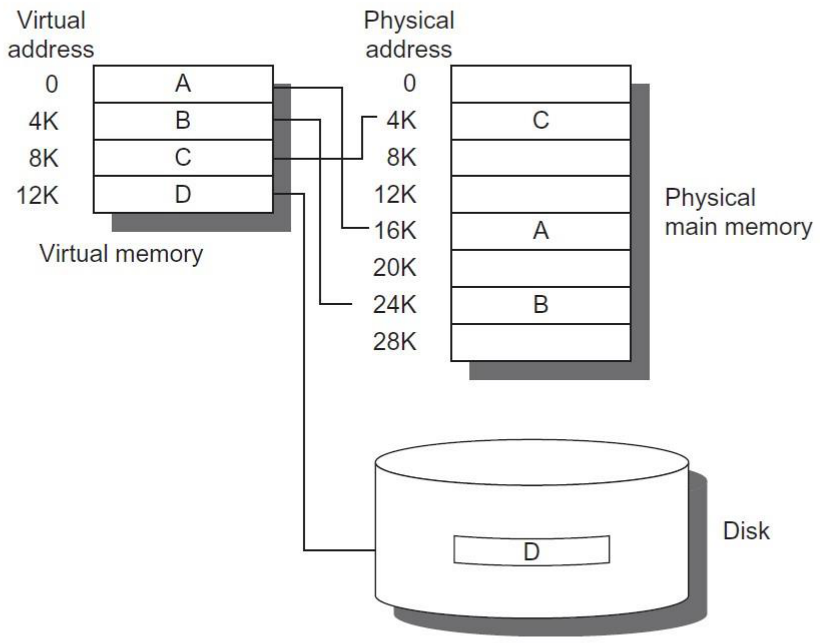


Figure 3.1: The logical program in its contiguous virtual address space is shown on the left. It consists of four pages, A, B, C, and D. The actual location of three of the blocks is in physical main memory and the other is located on the disk. Adapted from [20].

This approach functions as a translation mechanism that converts Virtual Page Numbers (VPNs) into their corresponding Physical Page Numbers (PPNs). The translation is handled through software using a structured data format known as a page table, which maintains all the necessary mapping information. Each time memory access occurs, the page table identifies the exact physical memory location associated with the requested virtual page [20]. When a page is present in physical memory, the Page Table Entry (PTE) holds the corresponding physical page number along with several status bits. These bits provide important metadata about the page, such as whether it has been modified (dirty), or is executable. On the other hand, if the page is not currently in memory, the PTE will typically point to its location in the swap space on disk. This allows the operating system to retrieve the page when needed, ensuring efficient memory management even when physical memory is limited [55]. Paged virtual memory implies that each memory access logically involves at least two separate operations: the first access is required to retrieve the physical address from the page table, and the second access is then used to obtain the actual data from memory. This two-step process effectively doubles the time it takes to complete a single memory reference.

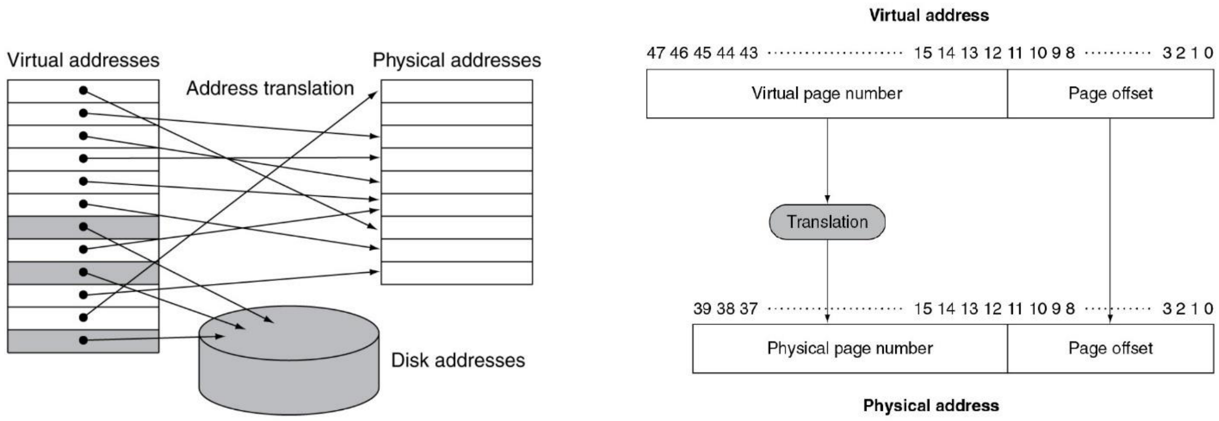


Figure 3.2: Virtual Address Translation. Adapted from [55].

As can be seen in Figure 3.2, the two main items for virtual address translation are Page number and Page offset. The **"page number"** refers to the portion of a virtual or physical address that indicates which page in the memory hierarchy is being addressed. It is used as an index into the page table. The page table maps virtual pages to physical pages, allowing the system to determine the corresponding physical page for a given virtual page. Furthermore, the **"page offset"** refers to the lower-order bits of a virtual or physical address that indicate the position of a specific byte within a memory page. It is kept unchanged during the address translation process. It determines the byte's position within the page, helping the system locate the specific data within the physical page [55].

Virtual machines (VMs), a concept nearly as longstanding as virtual memory itself, have seen renewed interest in recent years due to several compelling factors [20]. The growing demand for enhanced isolation and security in modern computing environments, alongside the shortcomings in security and stability of traditional operating systems, has contributed significantly to this resurgence. Additionally, the need to allow multiple unrelated users to share a single physical machine-common in cloud computing and data center operations- has further driven the adoption of VMs. Technological advancements, particularly the substantial improvements in processor performance, have also made the performance overhead introduced by VMs much more tolerable [20].

VMs provide an entire system-level environment that replicates the functionality of the binary instruction set architecture (ISA). Typically, the virtual machine supports the same ISA as the host hardware, ensuring compatibility and efficient execution. Unlike conventional computing platforms where a single operating system has exclusive control over all hardware resources, virtual machines allow multiple operating systems to coexist and share the same physical resources.

VMs offer two additional advantages that hold substantial commercial value. They provide an abstraction layer that enables complete software stacks-including older operating systems like DOS-to operate independently on the same physical hardware. This versatility allows for various scenarios, such as running outdated OS versions for legacy application support, using stable releases for production environments, or deploying newer versions for testing purposes. On the hardware level, VMs enable different software environments to run concurrently on a single machine, minimizing the need for multiple servers and enhancing overall resource efficiency. The isolation between VMs also improves system reliability, as each virtual machine functions separately [20].

In the context of virtual memory, retrieving the appropriate entry from the page table can be a time-consuming and resource-demanding task, as it requires a procedure known as a page table walk. This cost would be far too dear. The location of page tables (PTs) plays a critical role in the efficiency of virtual memory systems. Given that the space required for page tables is directly proportional to the size of the address space, the number of users, and inversely proportional to the page size, the memory footprint of PTs can become substantial. Due to this large space requirement, storing page tables in processor registers is not feasible. As a practical solution, operating systems typically store page tables in the main memory. However, this approach introduces performance overhead, as each memory access now requires an additional reference to fetch the page table entry before retrieving the actual data. Consequently, this mechanism effectively doubles the number of memory accesses for each operation. The solution is to rely on the principle of locality; if the accesses have locality, then the address translations for the accesses must also have locality [55].

To mitigate this issue, processors commonly include a specialized cache called the Translation Lookaside Buffer (TLB), which is intended to accelerate the address translation process. While much smaller in size compared to the full-page table, the TLB stores the most recently accessed virtual pages, enabling faster retrieval and reducing the need for frequent page table walks [5]. TLB is a part of the MMU and use only for PTE not normal usage but can take advantage of all methods to improve cache performance. The number and structure of TLBs can vary based on the specific CPU make and model. Many modern processors are equipped with more than one TLB, and some even implement multiple levels of TLBs, like the hierarchical design of memory caches. This multi-level TLB architecture helps reduce the occurrence of TLB misses and plays a crucial role in minimizing memory access latency.

The TLB maintains a record of Virtual Page Numbers (VPNs) along with associated entries in a Random Access Memory (RAM) that stores the corresponding Physical Page Numbers (PPNs). In virtual memory systems, the address space is divided between instructions and data, resulting in the implementation of two distinct TLBs: the Instruction TLB (ITLB) for instruction addresses and the Data TLB (DTLB) for data addresses [5].

A TLB entry functions similarly to a cache entry, where the tag contains parts of the virtual address, and the data section includes the physical page number, protection information, a valid bit, and typically a use bit and a dirty bit. These bits are managed by the operating system, which modifies them within the page table and then invalidates the related TLB entry to ensure consistency. When the entry is subsequently reloaded from the page table, the TLB receives an accurate and updated set of bits. TLBs are generally implemented as fully associative or set-associative structures, allowing entries to store any mapping or a specific subset of mappings within the buffer [20].

As Figure 3.3 shows, whenever a translation is needed, the TLB performs a parallel comparison between the input data and all stored entries to swiftly identify a matching result. When it receives a VPN along with an Address Space Identifier (ASID) from the Program Counter (PC), it searches for a corresponding entry. ASID is essential for distinguishing between memory pages that share the same virtual address but belong to different processes, ensuring correct address translation and process isolation [5].

When a process switch occurs, the TLB might need to be flushed because the virtual addresses for the new process will likely map to different physical addresses, ASID allows the TLB to cache entries for multiple processes simultaneously by tagging each entry with an ASID, avoiding unnecessary flushes. A match leads to a TLB hit, retrieving the corresponding PPN from RAM.

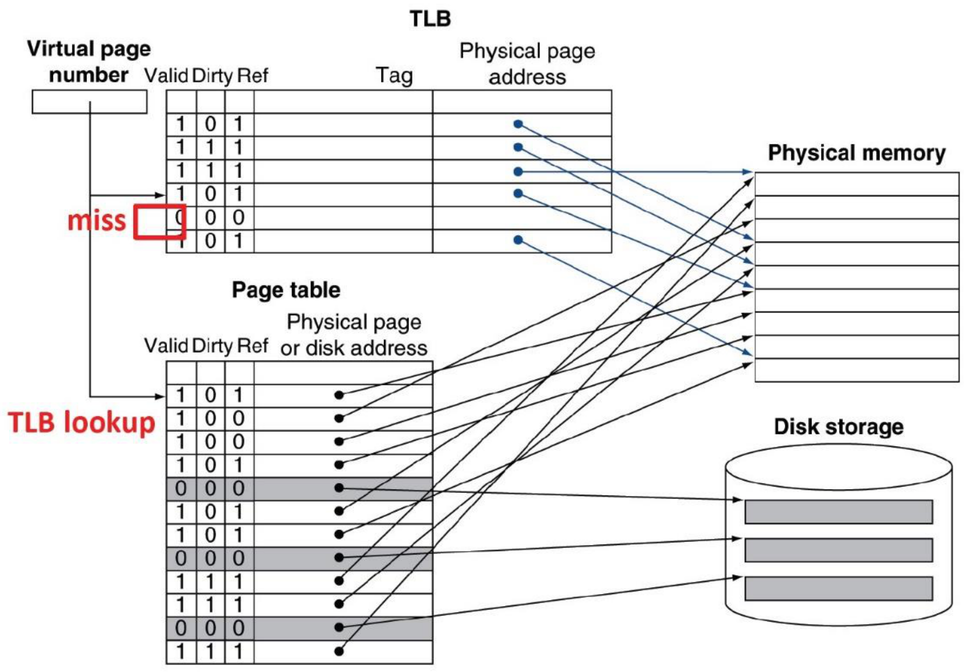


Figure 3.3: Fast translation using a TLB. Adapted from [55].

When the TLB does not contain the required entry, miss happens, it initiates a search through the page table, and the appropriate entry is fetched from memory using the PTW. If the page is either not present in memory or resides on a disk, a page fault is triggered, leading to an interruption being handled by the operating system. In response, the OS retrieves the page from disk and loads it into memory, subsequently updating the TLB. During this update, a replacement algorithm is employed to determine which existing TLB entry should be replaced. This entire process increases system latency and introduces performance overhead due to the additional runtime operations involved [5].

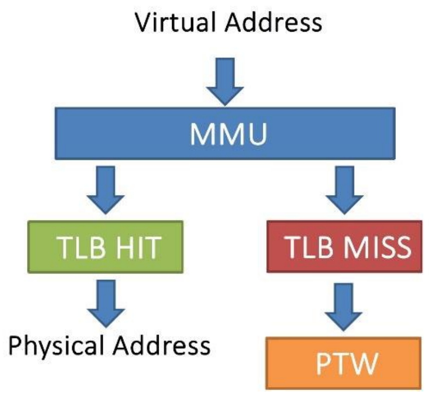


Figure 3.4: MMU using virtual address. Adapted from [55].

As has been shown in figure 3.4, MMU sends the translation request to TLB and in case of facing a Miss, it starts a PTW. The MMU is a vital hardware component necessary for enabling advanced operating systems such as Linux. In RISC-V architecture, the MMU supports multiple address translation modes, one of the most used being the SV39 scheme, which employs a 39-bit virtual address space and page-based memory translation. SV39 utilizes a three-level page table hierarchy to efficiently manage large address spaces. The first level maps 1-gigabyte pages, the second level handles 2-megabyte pages, and the third level manages standard 4-kilobyte pages, providing a scalable and hierarchical structure for address translation. Additionally, the MMU supports a configurable number of TLB entries, allowing system designers to tailor the buffer to specific performance requirements. To enhance efficiency during TLB misses, a hardware page table walker is integrated, which autonomously traverses the page table entries and facilitates rapid retrieval of the required mapping (Figure 3.5).

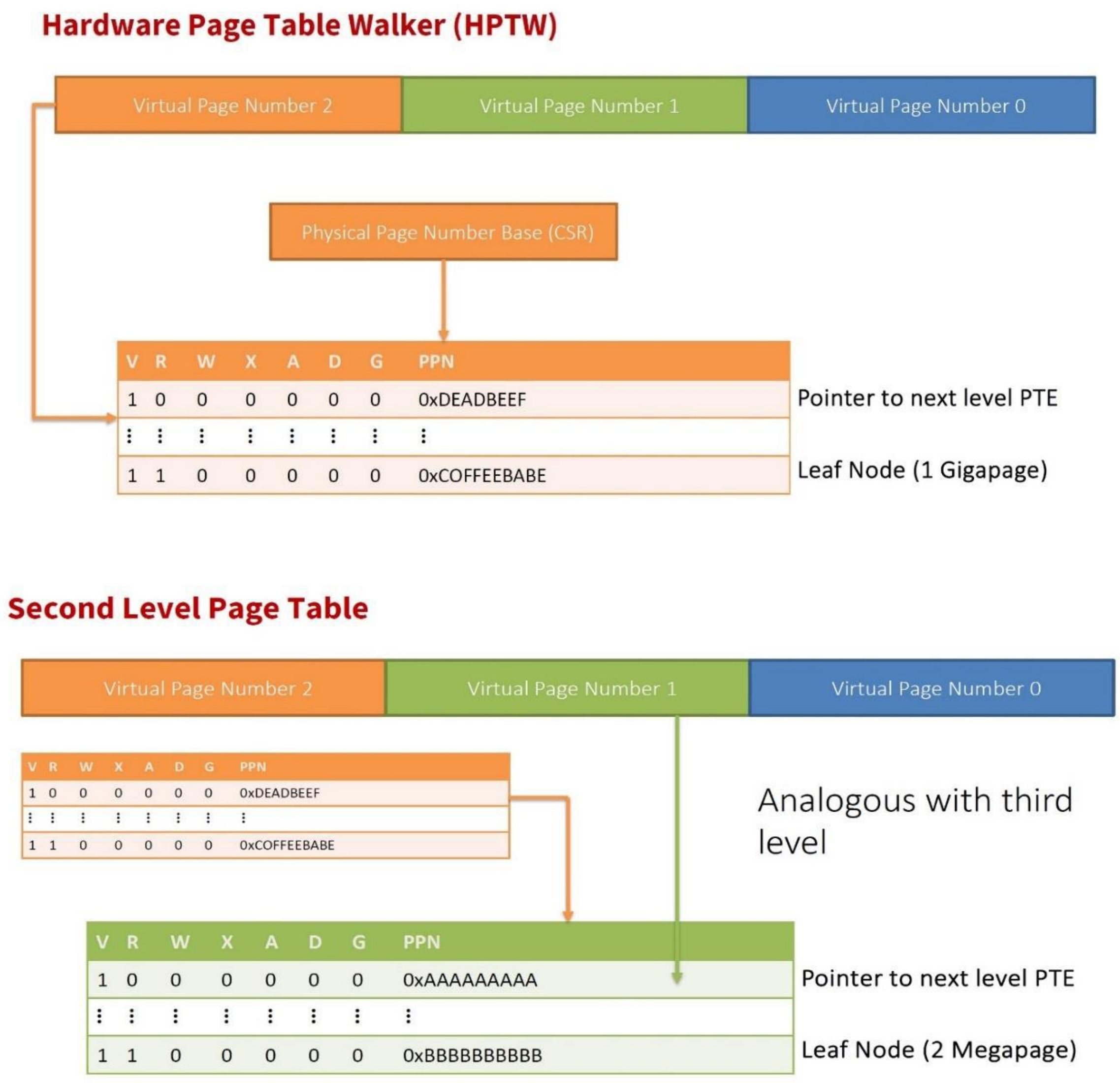


Figure 3.5: A hardware page table walk. Adapted from [55].

Page tables can be organized using one of two main approaches: a single-level structure or a multi-level structure. The multi-level page table, as implemented in architectures like RISC- V, is particularly effective in conserving memory by managing address translations more efficiently. During each memory operation in processors equipped with physical caches, a translation from virtual to physical addresses must take place.

Like other memory components, TLBs are vulnerable to radiation-induced faults known as SEUs. These soft errors can modify the contents of one or more memory cells, classified as Single Bit Upsets (SBUs) or Multiple Cell Upsets (MCUs), respectively. With continued advancements in technology leading to smaller memory cells that retain less charge, the susceptibility to Multi-Bit Upsets (MBUs) has significantly increased. SEUs can lead to operational faults in the TLB, such as false hits or false misses. A false hit, also referred to as a false positive, occurs when a corrupted TLB entry incorrectly matches the input, resulting in the use of an incorrect physical page and potentially causing system failures or data corruption. In contrast, a false miss, or false negative, takes place when a valid TLB entry is altered by an error, preventing a successful match. This forces the system to fall back on a page table walk, thereby increasing memory access latency.

Given the TLB's frequent usage, even though it occupies a relatively small portion of the system's memory, protecting it against such errors is critical. As hardware continues to scale down in size, the risk of multi-bit soft errors grows, making robust error protection mechanisms for the TLB more essential than ever [5].

Several techniques have been proposed to enhance the reliability of TLB-based structures against soft errors, and they can be categorized into three main approaches: circuit-level modifications, data duplication, and external error management modules. One solution focuses on ECC-equipped TLBs, utilizing multi-bit error protection schemes such as Hsiao or SEC-DED codes with specialized match-line designs to enhance resilience—this is demonstrated in [5]. Some proposals apply refresh-based fault tolerance mechanisms at the TLB level to recover from soft errors periodically [57]. In the realm of content addressable memory (CAM) organization, NAND match-line structures are used to reduce the vulnerability to soft errors, as explored in [58], while error-correcting match schemes have been integrated into CAMs for robust address matching, as shown in [59]. To handle errors probabilistically, scrubbing intervals are dynamically tuned to minimize soft error impact in Ternary Content Addressable Memory (TCAM) devices [60].

Further enhancements in cache-related CAM structures are also seen in [61], which

discusses CAM-based tags in highly associative caches to tolerate soft errors. Additionally, error detection and correction codes are integrated into CAM arrays for reliable data retrieval [62], and some techniques propose the use of Bloom filters combined with interleaved parity to reduce error probability, as introduced in [63]. A more robust alternative combines data duplication with error detection codes to protect CAMs, ensuring redundancy at the module level [7].
Lastly, TLB-specific strategies like parity error recovery mechanisms have also been suggested to correct soft errors in TLB entries [64].
The ECC used throughout this work is based on the Hsiao SEC-DED code which is used a syndrome-based decoder capable of correcting all single-bit errors and detecting double-bit errors. Hsiao codes reduce logic complexity by minimizing the number of XOR gates required for parity generation and error correction. In this design, ECC is applied to the TLB tag, valid bit, and content arrays as shown in Figures 3.6, 3.7 and 3.8.

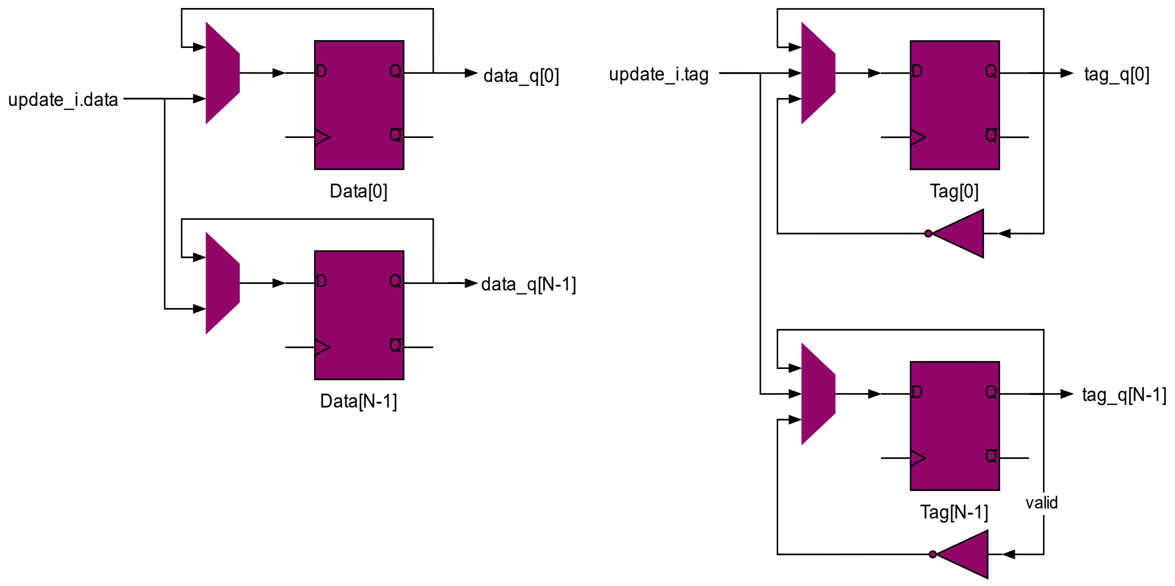


Figure 3.6: TLB without ECC protection. Adapted from [65].

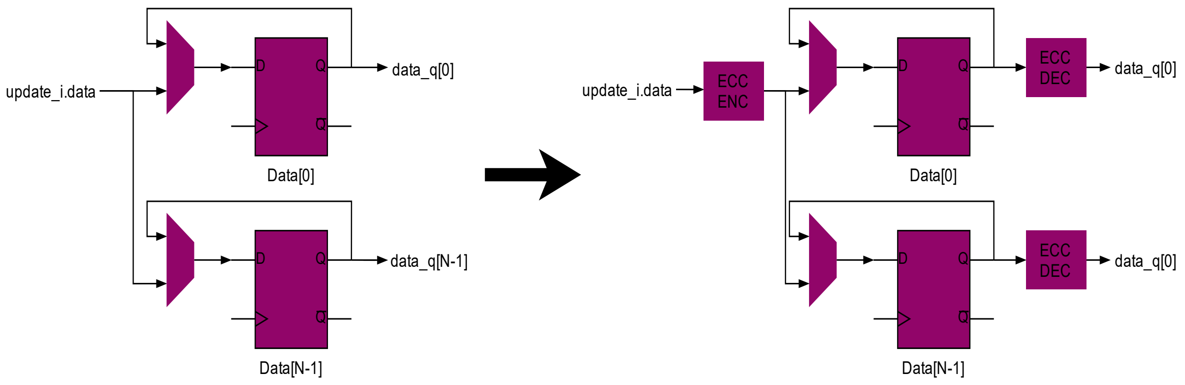


Figure 3.7: TLB with adding ECC to TLB content. Adapted from [65].

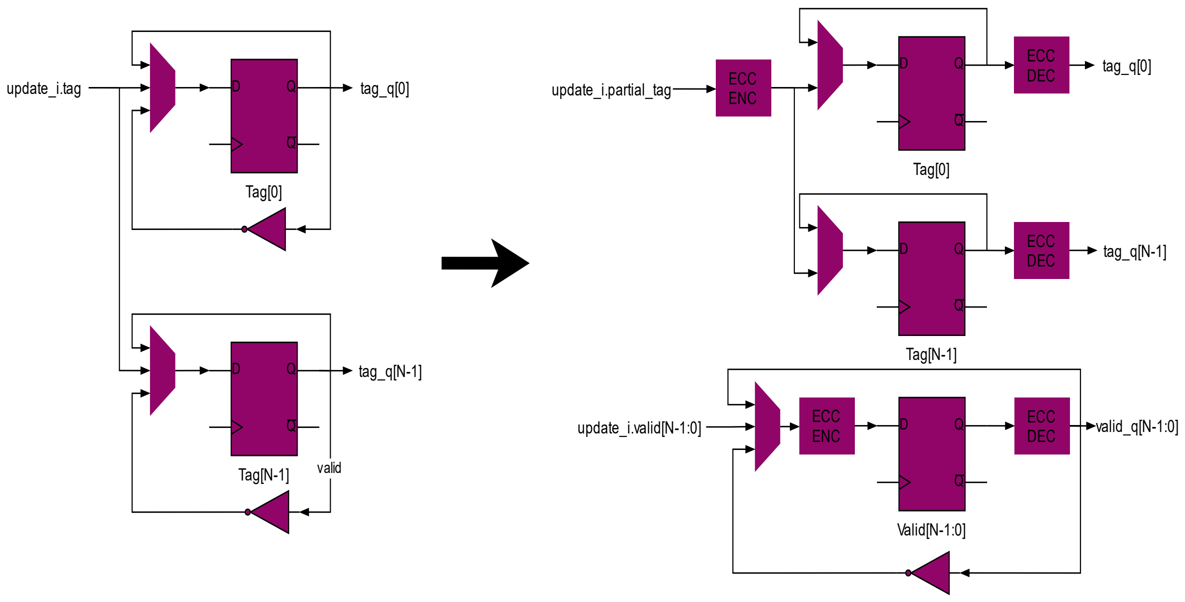


Figure 3.8: TLB with adding ECC to tag and valid bit. Adapted from [65].

As mentioned before, a key functional component of the MMU is the page table walker. The primary role of the PTW is to divide a virtual address based on the specific translation topology and scheme in use, such as Sv39, and then resolve it into a physical address by traversing the hierarchical page table structures. Once the nested translation process is complete, the PTW updates the TLB with the resolved PTEs along with the current Address Space Identifier (ASID) and the Virtual Machine Identifier (VMID).

An additional optimization is implemented by storing the page size used for translation- such as 4KiB, 2MiB, or 1GiB within the same TLB entry, along with the respective permission bits for each stage [56].

Taking advantage of ECC, the translation is encoded using the Hsiao code and then enters TLB. In each cycle the ECC decoder checks all entries in parallel and computes the syndrome to detect potential errors. If a 1-bit error is found, the decoder can identify and even correct the corrupted tag value combinationally. However, if the entry is not being accessed by the MMU in that cycle, the corrected tag is not written back into the TLB memory, and no permanent correction occurs. The decoder simply detects the error but does not update the stored tag. This means the same error may be encountered again in future cycles unless an explicit correction mechanism is implemented. Only when the MMU uses the entry in the same cycle, the corrected tag is used for translation, but even in that case, the corrected value is not saved - it is used transiently, without modifying the original TLB entry.

The absence of a mechanism to permanently update the corrected value in the TLB memory leads to several inefficiencies and reliability concerns. First, if the underlying bit error remains uncorrected in the stored entry, the same fault will persist and must be detected and corrected repeatedly in every subsequent cycle that accesses the same entry. This repeated correction not only increases the workload of the ECC logic but also unnecessarily consumes additional power and processing time, which could otherwise be avoided by a one-time permanent fix. Moreover, persistent soft errors that are not corrected at the source increase the risk of propagating incorrect address translations if, for any reason, the correction mechanism fails or is bypassed. This undermines the dependability and performance of the address translation process. Therefore, ensuring that corrected entries are updated directly within the TLB is crucial for improving system efficiency, reducing redundant operations as well as latency, and enhancing overall fault tolerance [4].

3.2 PULP PLATFORM

The PULP Platform is a collaborative initiative led by the Energy-efficient Embedded Systems (EEES) group at the University of Bologna together with the Integrated Systems Laboratory (IIS) at ETH Zurich, with the primary goal of developing open-source RISC-V hardware. Both the processor and the complete system employed in this work are part of the PULP Project [64].

3.3 RISC-V

Recent developments in computing architecture have introduced the RISC-V instruction set architecture (ISA), which has now surpassed 10 billion shipped cores [56]. Unlike traditional proprietary ISAs, RISC-V stands out by offering a free, open standard with a modular and highly flexible extension framework, making it adaptable for a wide range of applications- from lightweight microcontrollers to high-performance supercomputers. The RISC-V privileged specification also includes hardware-level support for virtualization through the Hypervisor extension, which was officially ratified in late 2021 [56].
In its design, the RISC-V privileged architecture specifies that virtual addresses are translated into physical addresses by navigating a multi-level radix-tree page table. There are four defined topologies for this translation:

- Sv32 uses a two-level hierarchy for 32-bit virtual address spaces
- Sv39 employs a three-level tree for 39-bit spaces
- Sv48 extends this to a four-level hierarchy for 48-bit addresses
- Sv57 utilizes a five-level tree for 57-bit virtual address spaces

At each level, entries can either point to another level of the table (non-leaf) or directly to the final physical address translation (leaf). These pointers, along with access permissions, are stored in PTEs that are either 32-bit wide for RV32 or 64-bit wide for

RV64 architectures. Although RISC-V commonly uses a 4 KiB page size, its flexible hierarchy supports larger super pages, such as 2 MiB and 1 GiB in Sv39-to alleviate pressure on the TLB and improve address translation efficiency [56].

3.4 Cheshire

Cheshire is a highly energy-efficient, Linux-capable, 64-bit RISC-V host platform designed to flexibly integrate heterogeneous domain-specific accelerators (DSAs) [66]. Its modular architecture features a fully configurable interconnect, a wide range of optional peripherals, and an integrated direct memory access (DMA) engine to efficiently handle data transfers between the host and connected accelerators. At its core, Cheshire utilizes the 64-bit CVA6 application-class processor, which provides all the necessary hardware to independently boot and run a general-purpose operating system such as Linux. This includes RISC-V-compliant core-local and platform-level interrupt controllers, standard I/O interfaces for accessing external storage and peripherals, and an interface for off-chip DRAM, which is essential because typical embedded Linux systems require 8-16 MB of memory, often exceeding on- chip capacity [66].

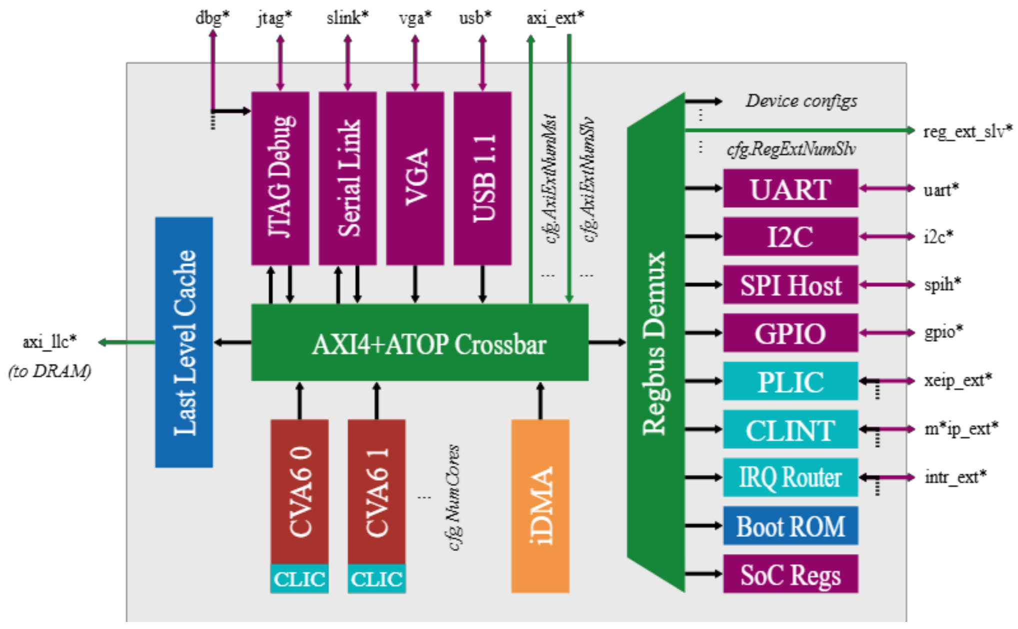


Figure 3.9: Architecture of the Cheshire platform. Adapted from [66].

As shown in Figure 3.9, architecture centers on a main AXI4 crossbar interconnect with ATOP (AXI5 atomic operations) support, which connects the CVA6 cores, off-chip RPC DRAM via a configurable last-level cache (LLC), DSAs, and other internal components. The crossbar's address width, data width, and the number of manager and subordinate ports can all be tailored to meet specific system bandwidth and addressing requirements. Simpler devices that do not require burst or out-of-order transactions are connected via a lightweight, extensible Regbus demultiplexer, which minimizes the crossbar's area and power footprint. The Regbus operates with a fixed 32-bit data width, providing efficient communication for lower-complexity peripherals and configuration interfaces.

Debug capabilities are supported by a RISC-V-compliant debug module with a JTAG transport interface, enabling real-time external debugging of the CVA6 processor and any additional RISC-V harts integrated in DSAs.

In terms of computing capacity, Cheshire can be configured with up to 31 CVA6 cores, each capable of running Linux and maintaining coherence through a self-invalidation mechanism.

RISC-V atomic operations are supported via a custom user-channel-based AXI4 extension. By default, Cheshire is set up with hypervisor support and the Core Local Interrupt Controller (CLIC) enabled, and each core acts as an independent AXI4 manager within the crossbar interconnect.

The platform includes a comprehensive suite of peripherals to meet diverse application requirements. These include standard I/O interfaces such as UART, I2C, SPI hosts, and GPIO modules. All peripherals seamlessly connect through either the AXI4 or Regbus interconnects and are compatible with standard Linux drivers, ensuring straightforward software integration.

The interconnect architecture supports up to 16 external AXI4 manager ports and up to 16AXI4 and Regbus subordinate ports, offering extensive flexibility for integration with broader system-on-chip (SoC) memory infrastructures.

3.5 CVA6

The CVA6 is an application-class RISC-V processor core developed as part of the PULP platform and is currently maintained by the OpenHW Group (Figure 3.10).
The CVA6 is an open-source, 64-bit RISC-V processor core featuring a 6-stage, single-issue, in-order pipeline. It is fully capable of running Linux and implements the RV64GC instruction set architecture variant. Additionally, it supports SV39 virtual memory through an integrated MMU, provides three privilege levels (Machine, Supervisor, and User), and includes Physical Memory Protection (PMP) functionality [64],[67]. It features a configurable size, distinct TLBs for instructions and data, a hardware page table walker, and branch prediction mechanisms, including a branch target buffer and a branch history table [68].
The MMU in CVA6 features two compact, fully associative TLBs: a Ll DTLB for data and a Ll ITLB for instructions. Each TLB can store up to l6 entries and fully supports flush operations with the ability to filter by ASID and virtual address.
To enable nested translation, the Ll DTLB and ITLB are designed to handle two translation stages, incorporating access permission checks and Virtual Machine Identifiers (VMIDs). Each TLB entry contains both the VS-Stage and G-Stage PTEs along with their respective permissions. The lookup process combines the translation sizes from both stages; for example, if the VS-stage provides a 4KiB translation and the G-stage provides a 2MiB translation, the resulting effective translation size will be 4KiB [56].
The CVA6 pipeline is organized into six stages as can be seen in Figure 3.10: two Instruction Fetch (IF) stages, followed by Instruction Decode (ID), Instruction Issue (IS), Instruction Execution (IE), and Writeback (WB). The IF unit is capable of fetching either a single 32-bit instruction or two l6-bit compressed instructions per cycle from the instruction cache. Retrieved instructions are temporarily stored in an 8-entry instruction buffer before proceeding to the ID stage. Additionally, the IF stage includes basic branch prediction mechanisms to help reduce pipeline stalls and maintain efficient instruction flow [69].

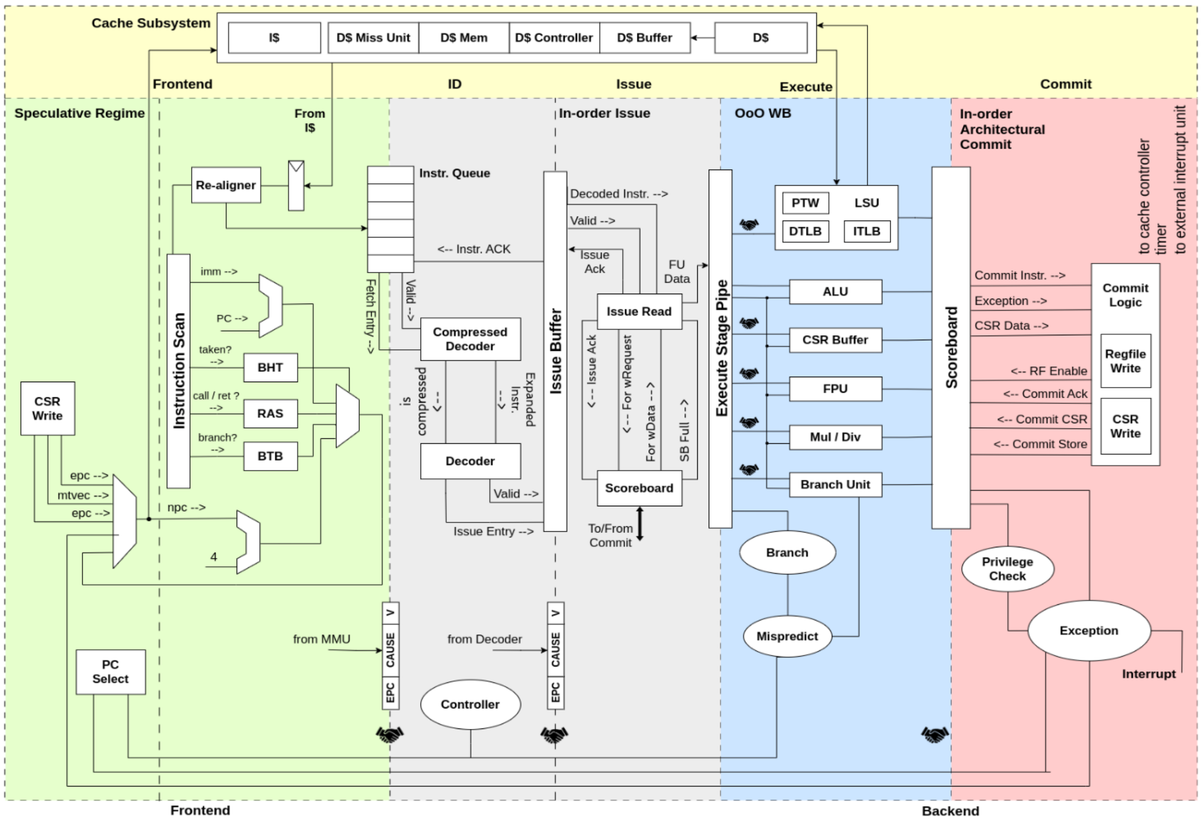


Figure 3.10: Architecture of CVA6. Adapted from [58].

4. Architecture and Implementation of Correction Mechanism

4.1 Concept

As discussed in the preceding chapter, the TLB functions as a cache that stores recently used translations required for converting virtual addresses, as issued by the MMU, into corresponding physical addresses. When a requested virtual address is already present in the TLB, a hit signal is generated, allowing the MMU to proceed with minimal latency. Conversely, if the entry is not available, a TLB miss occurs, prompting the MMU to perform a PTW to retrieve the appropriate mapping and subsequently update the TLB with the obtained result.

During each cycle, an ECC mechanism scans all TLB entries in parallel, detecting and correcting any single-bit errors to ensure that the data is reliable and prepared for accurate address translation. The processes of encoding and decoding in ECC introduce additional latency, thereby impacting the overall performance of protected storage in terms of read and write operations. This overhead becomes particularly detrimental in processor pipelines where the associated storage components are accessed frequently, resulting in a notable slowdown in pipeline efficiency [16]. Correcting corrupted entries within the TLB is essential to alleviate the computational burden on the ECC unit and enhance overall system reliability. In pursuit of this objective, five distinct correction strategies are explored in this section: (1) Detection-Only Mode, (2) Full parallel correction using separate encoder per each TLB entry, (3) Counter-Based Correction, (4) Arbiter-Based Correction, and (5) Correction mechanisms integrated with performance enhancement, specifically applied to the latter two approaches.

All the proposed approaches are fundamentally based on the initial step of notifying the TLB about the presence of an error. In the baseline TLB design with ECC, illustrated in Figure 4.1, the ECC mechanism can detect and correct errors; however, it does not update the TLB with the corrected data. Consequently, it becomes necessary to signal

TLB when an error has occurred. This can be achieved by utilizing the decoder output signal (err_o), which leverages the syndrome bits to indicate whether a single-bit or multi-bit error has been detected.

```
assign single_error = ^syndrome_o; // parity of syndrome

assign err_o[0] = single_error; // 1 if single error

assign err_o[1] = ~single_error & (|syndrome_o);
// 1 if not single but some syndrome bits → multi-bit error
```

The ECC employed in this study is derived from the Hsiao Single Error Correction–Double Error Detection (SEC-DED) code, utilizing a syndrome-based decoder that can accurately correct any single-bit error and detect the presence of double-bit errors. The interface between the ECC unit and the TLB is facilitated by two modules: hsiao_ecc_enc and hsiao_ecc_dec. These modules handle the transmission of raw valid bits, tag fields, and content bits from the MMU to the encoder, which then produces the encoded data. This encoded data is subsequently passed to the decoder, which generates output signals corresponding to the original fields. In addition to these outputs, the decoder also produces an err_o signal, which indicates the presence of an error. While the original TLB design does not utilize the err_o signal, the proposed solutions incorporate it by assigning its value to four newly defined flags—invalidate_tag, invalidate_pte, invalidate_gpte, and invalidate_valid. These flags are designed to capture the error notification and identify faults in the tag, content, or valid bits of the TLB entries.

The occurrence of an error can be observed when the corresponding bit for the affected TLB entry within the error-detection flags transitions to a value of 1 or 2 within the same clock cycle.

There are two primary operational modes for handling errors: entries identified as erroneous can either be invalidated or corrected. Depending on the selected mode, the validity bit of the tags is assigned accordingly to reflect whether an entry remains valid or is marked for invalidation. It can be set to zero to invalidate the entry or assigned the value from the decoder's valid-bit output (valid_dec) to retain it. If the corrupted entry is the same entry which is required for address translation, the output signal lu_hit_o, which determines whether a matching entry exists in the TLB for the virtual address provided by the MMU, will be deserted (set to zero), indicating a translation miss. In the case of two-bit errors, the ECC mechanism lacks the capability to perform correction. Consequently, if an entry contains more than a single-bit error in either the tag or content fields, the system handles this scenario by invalidating the affected entry. This is achieved by setting its tags' validity bit to zero, thereby ensuring that the corrupted entry is excluded from future address translations.

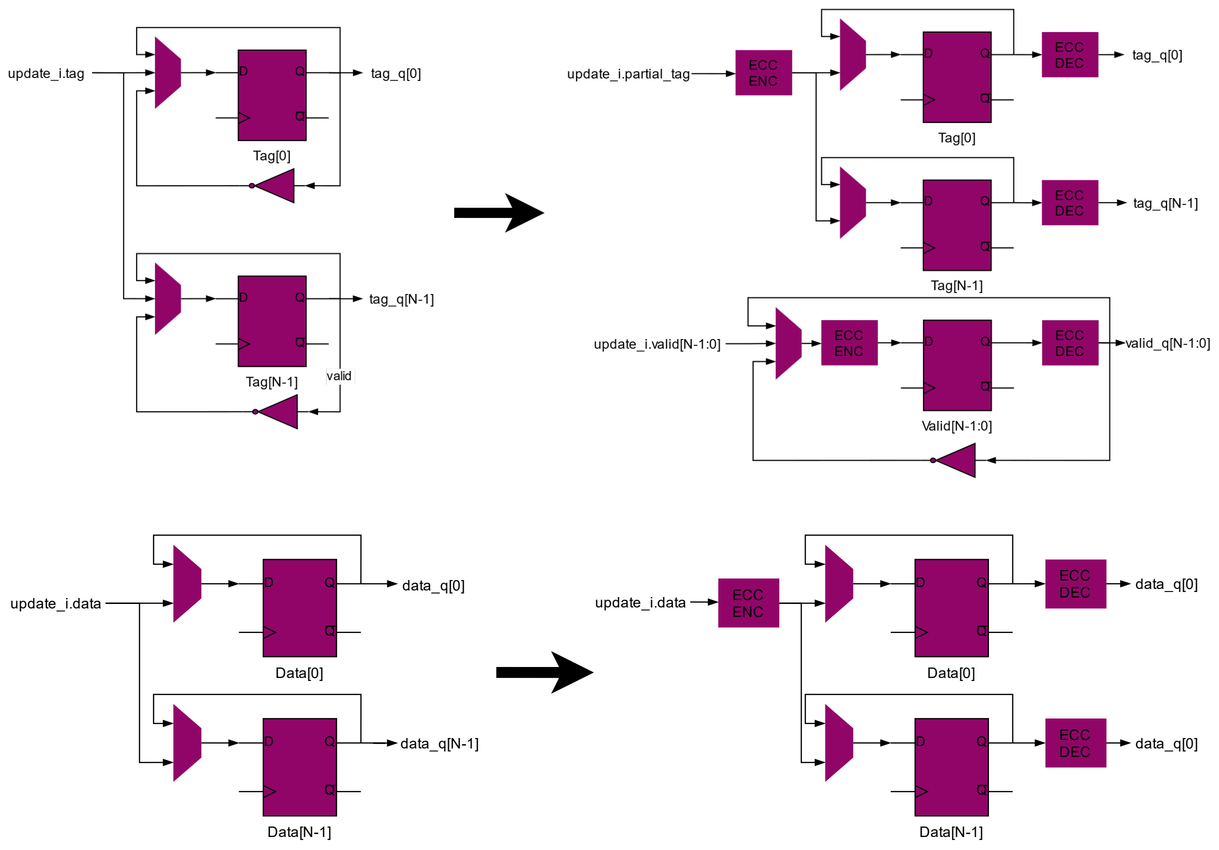


Figure 4.1: TLB extended with ECC

Regarding the handling of valid bits, it is important to note that in the combinational feedback loop, the decoder output is continuously fed into the encoder of the valid bits on each cycle. As a result, any single-bit error in the valid field is automatically corrected in the subsequent cycle, with the corrected data being reintegrated into the TLB. However, in the presence of a two-bit error within the valid bits, the decoder output being a collective representation of all entries, cannot isolate the specific corrupted entry. Therefore, under such conditions, the only viable approach is to flush the entire TLB and rely on PTW to repopulate the entries with accurate data.

With the foundational concepts of error detection and correction now established, the subsequent sections will systematically present and analyze each of the proposed approaches in detail.

4.2 Detection Only

In this approach, error correction is deliberately omitted, thereby eliminating the need for any additional architectural modifications. Any error detected in either the tag or content field leads to the immediate invalidation of the affected entry. Should the MMU subsequently require that entry for address translation, a PTW is triggered to retrieve the necessary information.

While this approach eliminates the continuous time and energy overhead associated with performing correction in every cycle, it introduces inefficiencies in scenarios where errors occur in entries needed by the MMU. In such cases, a PTW must be initiated, incurring additional latency and energy consumption. To address this trade-off, it is proposed to implement a mechanism capable of correcting at least single-bit errors (as illustrated in Figure 4.2), thereby improving energy efficiency and performance. The first correction-based approach is discussed in the following section.

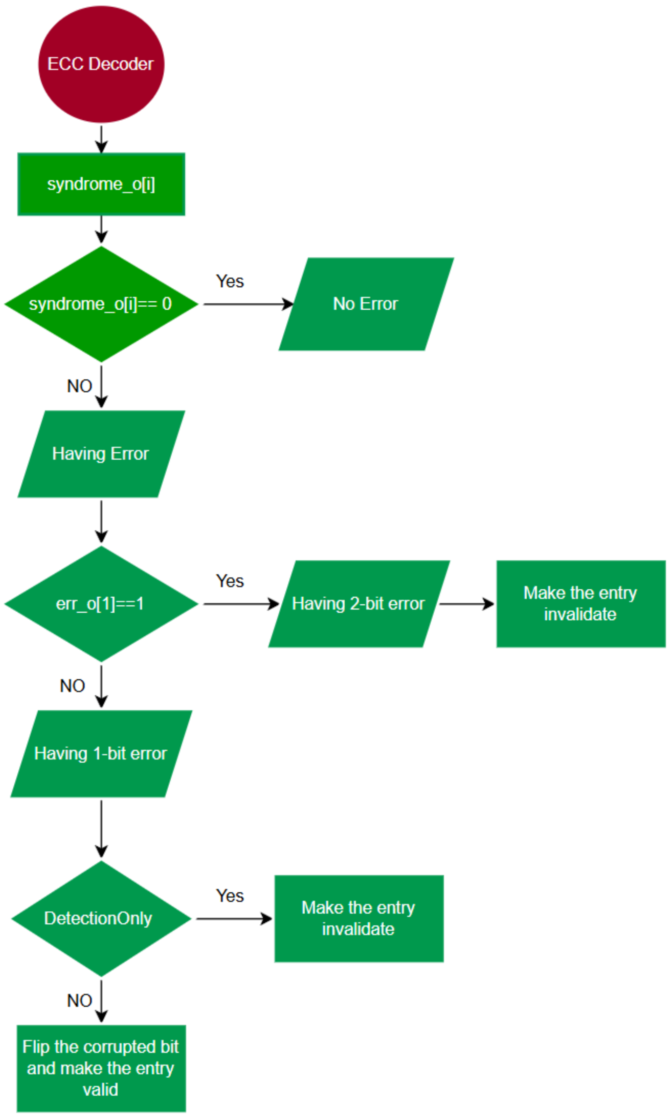


Figure 4.2: Error Detection and Correction Process Flowchart

4.3 Full parallel correction using separate encoder per each TLB entry

In this method, the TLB architecture is modified to enable the correction of entries affected by single-bit errors. This is accomplished by utilizing the outputs of the ECC decoder, which provides corrected data, to directly update the TLB. Specifically, the tag or content encoders, typically used to process the PTW results received from the MMU, are instead driven by the decoder outputs whenever a single-bit error is detected in either the tag or content fields. This re-initiate the encoding and decoding cycle, ensuring that the corrected, error-free data is accurately stored back into the TLB.

However, this correction mechanism is only activated when the MMU does not intend to overwrite the affected entry with new data. Upon re-feeding the encoder with the decoder's corrected output, the system detects errors during the first cycle, allowing the tags_n or content_n register, responsible for holding the encoder's output, to be updated with corrected data. In the subsequent cycle, the tags_q or content_q register, which governs the actual operational state of the TLB entry, is also updated. This two-cycle process ensures that future address translations utilize the corrected values, thereby maintaining system reliability and correctness.

The approach employed in this section is inspired by the "normal replacement" logic within the baseline TLB code, wherein the outputs of the tag and content encoders are assigned to corresponding TLB registers. To enable correction, this same assignment mechanism is extended to operate when the decoder’s corrected outputs are routed back through the encoders. This allows the corrected data to be properly encoded and stored within the TLB, maintaining consistency with the system's existing update pathway. One limitation of this approach arises when multiple TLB entries experience 1-bit errors simultaneously. Since the encoder can process only one input per cycle, each subsequent corrupted entry overwrites the output of the previous one, resulting in only the final entry being corrected and updated in the TLB. To address this issue, the design can be extended by allocating a dedicated encoder for each TLB entry, as illustrated in Figure 4.3. For instance, in a TLB with 16 entries, 16 separate encoders would be employed. This architecture enables all corrupted entries with 1-bit errors to be corrected and updated concurrently, thereby enhancing both the efficiency and reliability of the correction process.

As illustrated in Figure 4.3, the first multiplexer determines the source of input data to the encoder. If the MMU intends to update a specific entry, the corresponding update signal (update_i.tag) is directed to the encoder. Conversely, if no update is required, the corrected output from the decoder (tags_dec) is utilized instead. Subsequently, a second multiplexer decides whether the TLB should be updated with the new encoded data or retain the existing values by maintaining tags_n = tags_q. This logic and architectural framework are similarly applied to the content of the entries. In this case, the multiplexer selects between the MMU-provided data (update_i.content) and the corrected output from the content decoder (tlb_content_dec) as the encoder input, ensuring consistency and flexibility in handling both tag and content corrections. The correction mechanism for content is the same as Figure 4.3 with having separate encoders for each entry's content.

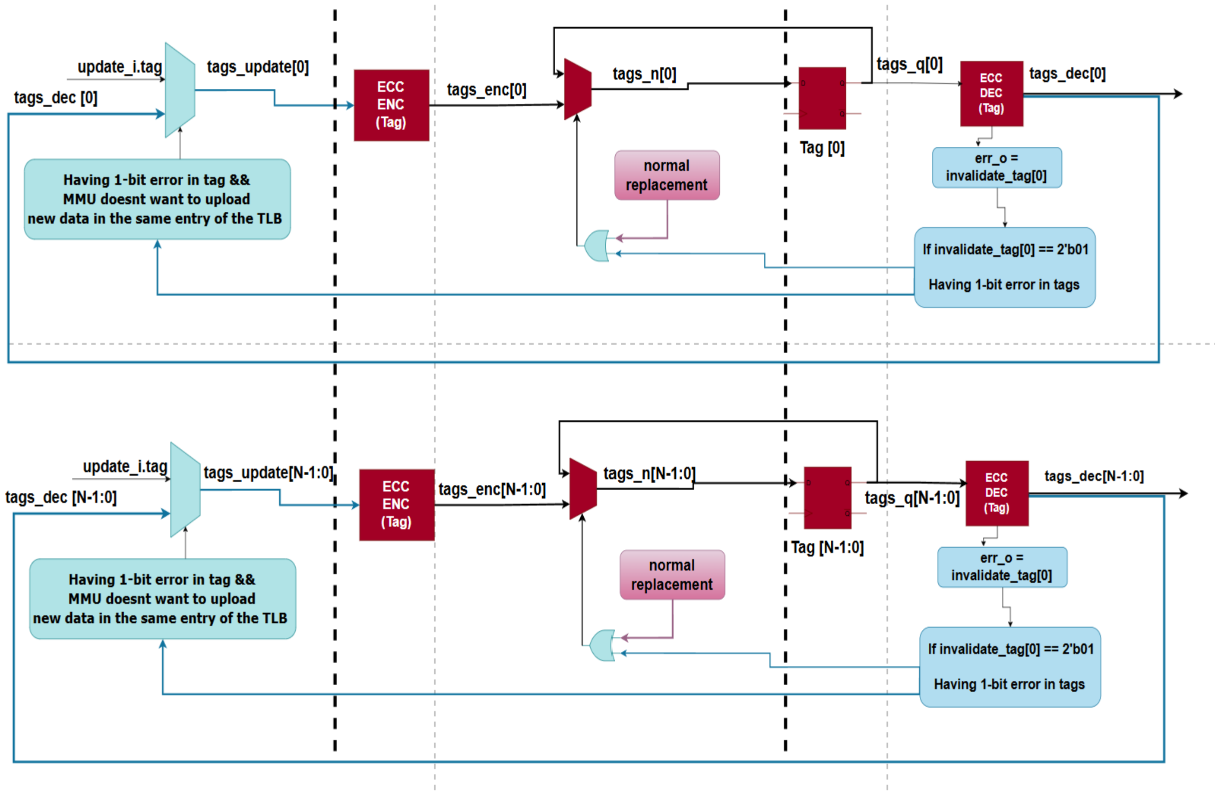

update_i.tag
tags_update[0]
tags_dec [0]
ECC ENC (Tag)
tags_enc[0]
tags_n[0]
tags_q[0]
ECC DEC (Tag)
tags_dec[0]
Tag [0]
Having 1-bit error in tag && MMU doesnt want to upload new data in the same entry of the TLB
normal replacement
err_o = invalidate_tag[0]
If invalidate_tag[0] == 2'b01
Having 1-bit error in tags
update_i.tag
tags_update[N-1:0]
tags_dec [N-1:0]
ECC ENC (Tag)
tags_enc[N-1:0]
tags_n[N-1:0]
tags_q[N-1:0]
ECC DEC (Tag)
tags_dec[N-1:0]
Tag [N-1:0]
Having 1-bit error in tag && MMU doesnt want to upload new data in the same entry of the TLB
normal replacement
err_o = invalidate_tag[0]
If invalidate_tag[0] == 2'b01
Having 1-bit error in tags

Figure 4.3: Correction mechanism by using separate encoders per each entry

However, this approach introduces a significant area overhead due to the inclusion of individual encoders, one for each TLB entry, instead of a single shared encoder. This increase in hardware resources may be considered impractical, particularly given that the likelihood of multiple entries experiencing simultaneous single-bit errors in real-world scenarios is relatively low. Consequently, the trade-off between improved correction capability and hardware efficiency must be carefully evaluated.

To address this limitation, an alternative approach is proposed that utilizes a finite state machine (FSM) in place of multiple encoders. In this method, corrupted entries are corrected sequentially, one per cycle, rather than concurrently. Although this sequential correction introduces a delay, requiring the system to wait for the FSM to complete its cycle before all errors are resolved, it offers a favorable trade-off between hardware area and performance.

4.4 Counter-based solution for correction

In this approach, a simple finite state machine (FSM) is realized using a counter mechanism, as depicted in Figure 4.4. The FSM operates in two modes: IDLE and CORRECTING. Once a 1-bit error is detected in any of the entries (indicated by one or more bits of the invalidate_tag, invalidate_pte, or invalidate_gpte signals being set to 1) the FSM transitions from the IDLE state to the CORRECTING state. The counter then begins iterating from the lowest-index corrupted entry. It sequentially updates the correction index to target the next corrupted entry and applies the correction mechanism accordingly. As a result of this process, the corresponding error flag for each corrected entry is cleared (changing from 1 back to 0) thereby signaling that the entry has been successfully repaired.

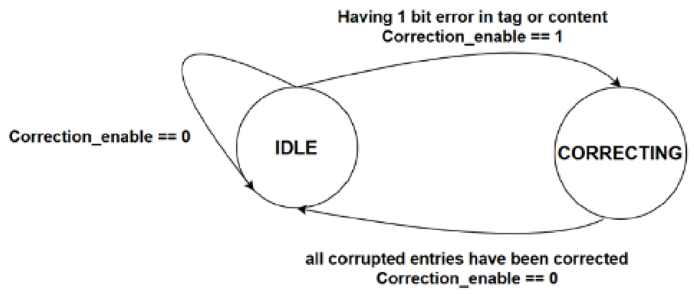
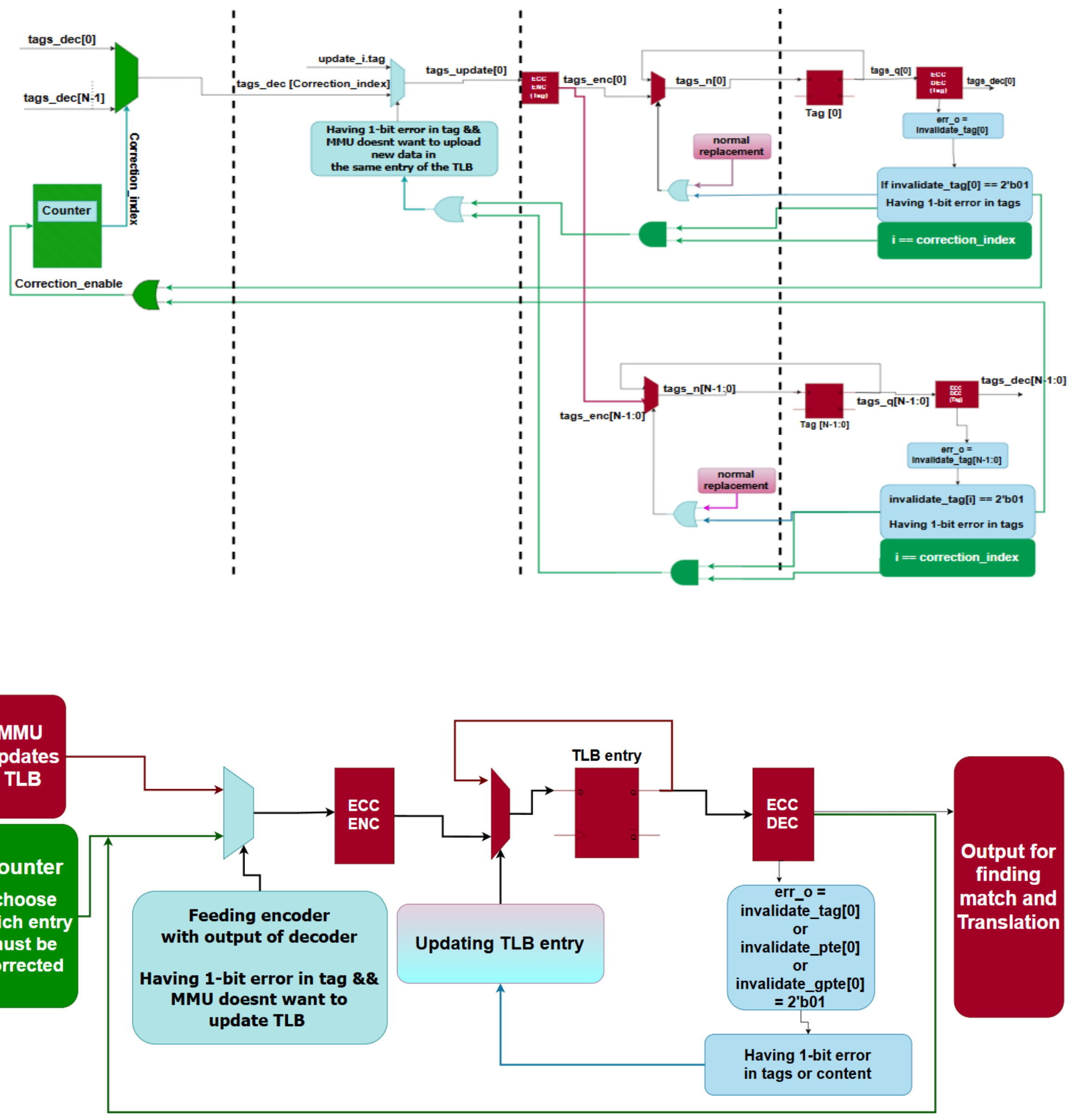


Figure 4.4: Counter-based correction FSM

The correction mechanism, as discussed before, includes injection of decoder output (corrected data) in encoder again and assign output of encoder to flip flop of the corrupted entry in TLB to update the entry (Figure 4.5) and proceed with this process till last corrupted entry.



tags_dec[0]
tags_dec[N-1]
Correction_index
Counter
Correction_enable
update_i.tag
tags_dec [Correction_index]
tags_update[0]
Having 1-bit error in tag && MMU doesnt want to upload new data in the same entry of the TLB
tags_enc[0]
tags_n[0]
Tag [0]
tags_q[0]
tags_dec[0]
err_o = invalidate_tag[0]
normal replacement
If invalidate_tag[0] == 2'b01
Having 1-bit error in tags
i == correction_index
tags_n[N-1:0]
tags_enc[N-1:0]
Tag [N-1:0]
tags_q[N-1:0]
tags_dec[N-1:0]
err_o = invalidate_tag[N-1:0]
normal replacement
invalidate_tag[i] == 2'b01
Having 1-bit error in tags
i == correction_index
MMU updates TLB
Counter choose which entry must be corrected
Feeding encoder with output of decoder
Having 1-bit error in tag && MMU doesnt want to update TLB
ECC ENC
Updating TLB entry
TLB entry
ECC DEC
err_o = invalidate_tag[0] or invalidate_pte[0] or invalidate_gpte[0] = 2'b01
Having 1-bit error in tags or content
Output for finding match and Translation

Figure 4.5: Counter-based correction mechanism for TLB extended with ECC

Thus, only one entry is updated per cycle. It is important to note that during the counting cycles, if a newly corrupted entry, positioned before the next previously detected erroneous entry, is identified, the counter will correct this new entry upon reaching it. Once corrected, the counter resumes progressing toward the originally detected corrupted entry, ensuring all errors are addressed in sequence.

One notable limitation of this design is its handling of newly occurring single-bit errors in entries that the counter has already passed. In such cases, the counter does not revisit these entries during the ongoing correction cycle. However, the finite state machine (FSM) is prevented from transitioning back to the IDLE state due to the persistent error flag. As a result, the counter continues to increase up to the index of last entry, subsequently wrapping around to revisit earlier entries, thereby eventually correcting the missed error. While it is technically feasible to introduce additional hardware to immediately address such late-occurring errors, the likelihood of this scenario is exceedingly rare in practical applications. Thus, the complexity and resource cost of implementing such enhancements are not considered justified.

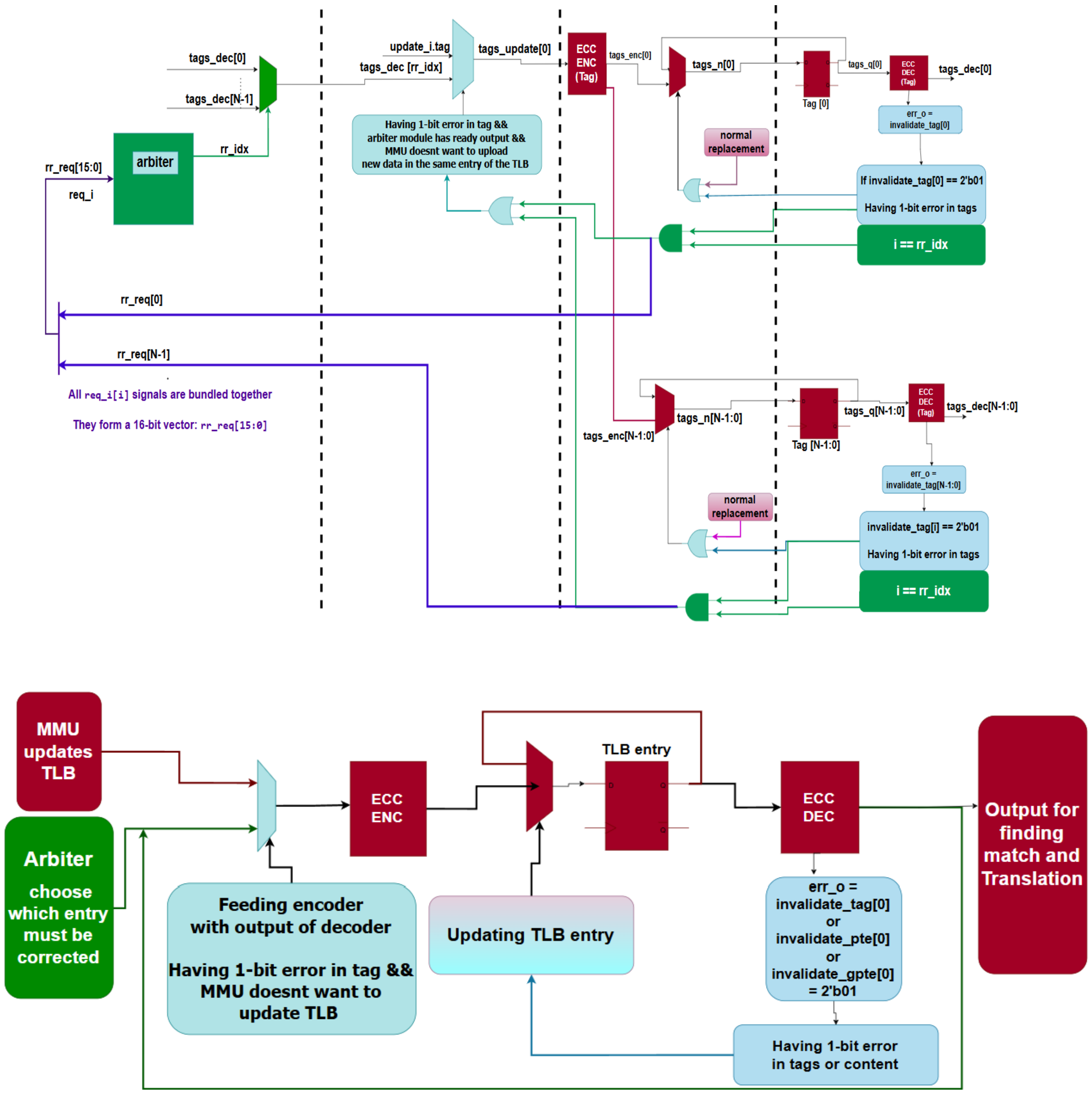
In conclusion, this approach successfully corrects all corrupted entries while reducing the architectural overhead compared to the previous method. However, it still pushes latency due to the need for the counter to iterate from the beginning up to the index of the next corrupted entry. For instance, if single-bit errors exist in both entry 0 and entry 12, the counter must remain active for 11 cycles before correcting the latter. During this period, if corrupted entry is required for address translation, the ECC unit must perform correction to support MMU access, thereby consuming additional energy. To address this inefficiency, an alternative solution is proposed: replacing the counter with an arbiter, which can directly identify and access the next corrupted entry without the need for sequential counting.

4.5 Arbiter-based solution for correction

By replacing the counter with a round-robin arbiter, the correction mechanism is optimized to eliminate the sequential counting overhead. In this configuration, a request vector is generated in which each bit signifies an entry with a detected single-bit error.

This vector is then fed into the round-robin arbiter, which efficiently cycles through the active requests and sequentially grants correction access to one entry at a time. This approach enables more targeted correction, reducing unnecessary traversal and improving energy efficiency.

Unlike the counter-based approach, the arbiter does not inherently prioritize special index to start the correction and follow the order. Instead, it grants the next correction request starting from the entry immediately following the last one it served.



tags_dec[0]
tags_dec[N-1]
arbiter
rr_idx
rr_req[15:0]
req_i
update_i.tag
tags_dec [rr_idx]
tags_update[0]
Having 1-bit error in tag && arbiter module has ready output && MMU doesnt want to upload new data in the same entry of the TLB
ECC ENC (Tag)
tags_enc[0]
tags_n[0]
Tag [0]
tags_q[0]
ECC DEC (Tag)
tags_dec[0]
err_o = invalidate_tag[0]
normal replacement
If invalidate_tag[0] == 2'b01
Having 1-bit error in tags
i == rr_idx
rr_req[0]
rr_req[N-1]
All req_i[i] signals are bundled together
They form a 16-bit vector: rr_req[15:0]
tags_enc[N-1:0]
tags_n[N-1:0]
Tag [N-1:0]
tags_q[N-1:0]
ECC DEC (Tag)
tags_dec[N-1:0]
err_o = invalidate_tag[N-1:0]
normal replacement
invalidate_tag[i] == 2'b01
Having 1-bit error in tags
i == rr_idx
MMU updates TLB
Arbiter choose which entry must be corrected
ECC ENC
TLB entry
ECC DEC
Output for finding match and Translation
Feeding encoder with output of decoder
Having 1-bit error in tag && MMU doesnt want to update TLB
Updating TLB entry
err_o = invalidate_tag[0] or invalidate_pte[0] or invalidate_gpte[0] = 2'b01
Having 1-bit error in tags or content

Figure 4.6: Arbiter-based correction mechanism for TLB with ECC

This round-robin scheduling ensures equitable treatment of all entries and eliminates the risk of starvation, as each request is guaranteed to be serviced in a cyclic and fair manner.

Upon detection of a single-bit error, the arbiter module is activated, and the corresponding entry is immediately corrected using the same correction mechanism previously described as shown in Figure 4.6. In scenarios where multiple entries contain errors, the arbiter sequentially processes each entry, applying corrections over successive cycles to ensure complete recovery without introducing significant overhead. There is no need to FSM here.

4.6 Performance Enhancement

Although the enhancements thus far have enabled the TLB integrated with ECC to correct single-bit errors effectively, the ECC process and associated correction mechanisms may introduce latency during MMU access to TLB entries for address translation.

The integration of ECCs to enhance system reliability inevitably introduces trade-offs, including increased circuit delay due to data encoding and decoding processes. Additionally, there is a notable rise in silicon area and power consumption, attributed to the extra interconnects, storage elements, and the overhead introduced by the encoder and decoder circuitry. [52] The substantial delay introduced by encoding and decoding operations can potentially render the register a performance bottleneck. This delay may necessitate a longer clock cycle, thereby reducing the system's operating frequency and overall performance efficiency [52].

Therefore, it is needed to modify the code such that decrease the sign of decoder or error signals on critical path to increase the frequency.

To address this issue, it is essential to restrict reliance on decoder output during periods when the MMU accesses the TLB. Consequently, all correction mechanisms must be halted during this interval, as they fundamentally depend on the corrected data provided by the decoder. Therefore, whenever the MMU is active (lu_access_i = 1), any TLB entry containing a single-bit error must be invalidated, regardless of whether it is the specific entry required by the MMU or not. Selectively invalidating only, the needed entry is not a feasible approach. As a result, if the MMU requests data from a corrupted entry, it must initiate a PTW to retrieve the correct information.

To ensure proper invalidation, it is critical that any TLB entry identified as corrupted becomes invalidated in the same cycle that the MMU initiates a translation request and remains invalid in subsequent cycles. This constraint is necessary to prevent inconsistencies: if the corrupted entry were to be corrected and revalidated in the following cycles, it could reappear in the TLB while the MMU is already executing a PTW based on the original invalidation. Such behavior may result in both the newly corrected entry and the one being uploaded by the PTW coexisting in the TLB. This overlap can lead to a functional conflict, where the TLB produces multiple hits for the same virtual address, thereby violating the expected invariants of the TLB's operation.

To meet the objective while also preventing the formation of combinational loops, the existing update and normal replacement logic in the TLB design (previously used to refresh entries with new or corrected data) has been repurposed to explicitly invalidate corrupted entries by setting their validity bits to zero. Unlike the correction mechanism, where validity is conditionally masked, this approach requires a direct update to the valid bit of the entry by changing the valid_update[i] signal for the affected entry. Furthermore, an auxiliary flag, already_invalidated[i], is introduced to track whether a specific entry was invalidated due to a detected single-bit error occurring concurrently with an MMU translation request. This ensures that the invalidation is persistent and correctly managed during MMU interactions.

Additionally, the conditions governing the initiation and execution of the correction process have been revised. Specifically, if the MMU issues a translation request and a corresponding entry in the TLB has been previously invalidated due to a detected single-bit error during that request, the correction mechanism is inhibited. This design choice ensures that such entries remain invalidated and are not inadvertently corrected, thereby avoiding any conflict with ongoing PTWs. The invalidated entry will persist in this state until it is either explicitly overwritten with new data from the MMU or removed through a flush operation.

Furthermore, up to this point, the translation process and match detection within the TLB have relied on the decoder outputs, regardless of whether the data was originally correct or corrected by the ECC decoder. However, to implement the ECC bypass strategy, it becomes necessary to revise the process so that, in the case of entries that are both valid and free from error, the MMU directly accesses the tag and content information from the TLB, completely bypassing the decoder outputs. This modification ensures that ECC-related latency is avoided when it is not required, thereby optimizing performance.

Because of this enhancement, the registers responsible for producing the output of the TLB system, previously driven by the decoder outputs, must now be connected to the raw, uncorrected data. This change extends to the validity signal as well. As discussed earlier in this chapter, the validity bit of tags was previously assigned to the decoder-corrected validity bit (valid_dec). However, in the updated architecture, this signal is reassigned to the raw validity bit (valid_q), under the assumption that all entries affected by single-bit errors have already been invalidated. Furthermore, the matching logic used for address translation is now required to rely solely on raw data, excluding any decoder-corrected values.

Therefore, the previous design for counter-based correction solution will be changed to Figure 4.7.

Also, all the changes for arbiter-based correction solutions with performance enhancement is the same as the counter-based solution as Figure 4.8.

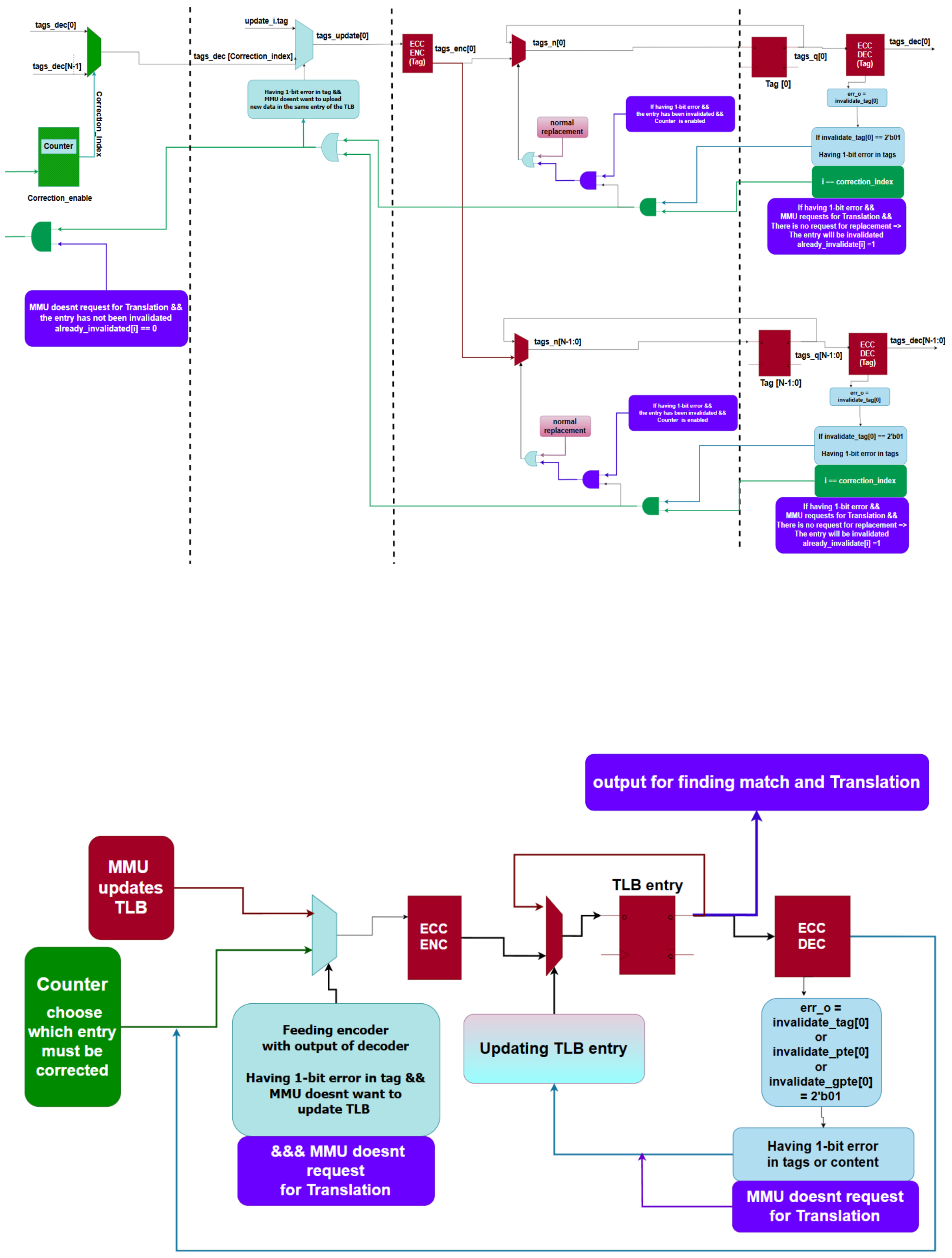

tags_dec[0]
tags_dec[N-1]
Correction_index
Counter
Correction_enable
MMU doesnt request for Translation && the entry has not been invalidated already_invalidated[i] == 0
update_i.tag
tags_update[0]
tags_dec [Correction_index]
Having 1-bit error in tag && MMU doesnt want to upload new data in the same entry of the TLB
ECC ENC (Tag)
tags_enc[0]
tags_n[0]
normal replacement
If having 1-bit error && the entry has been invalidated && Counter is enabled
Tag [0]
tags_q[0]
ECC DEC (Tag)
tags_dec[0]
err_o = invalidate_tag[0]
If invalidate_tag[0] == 2'b01
Having 1-bit error in tags
i == correction_index
If having 1-bit error && MMU requests for Translation && There is no request for replacement => The entry will be invalidated already_invalidate[i] =1
tags_n[N-1:0]
Tag [N-1:0]
tags_q[N-1:0]
tags_dec[N-1:0]
MMU updates TLB
Counter choose which entry must be corrected
Feeding encoder with output of decoder
Having 1-bit error in tag && MMU doesnt want to update TLB
&&& MMU doesnt request for Translation
ECC ENC
TLB entry
Updating TLB entry
ECC DEC
output for finding match and Translation
err_o = invalidate_tag[0] or invalidate_pte[0] or invalidate_gpte[0] = 2'b01
Having 1-bit error in tags or content
MMU doesnt request for Translation

Figure 4.7: Counter-based correction mechanism with Performance Enhancement for TLB extended with ECC

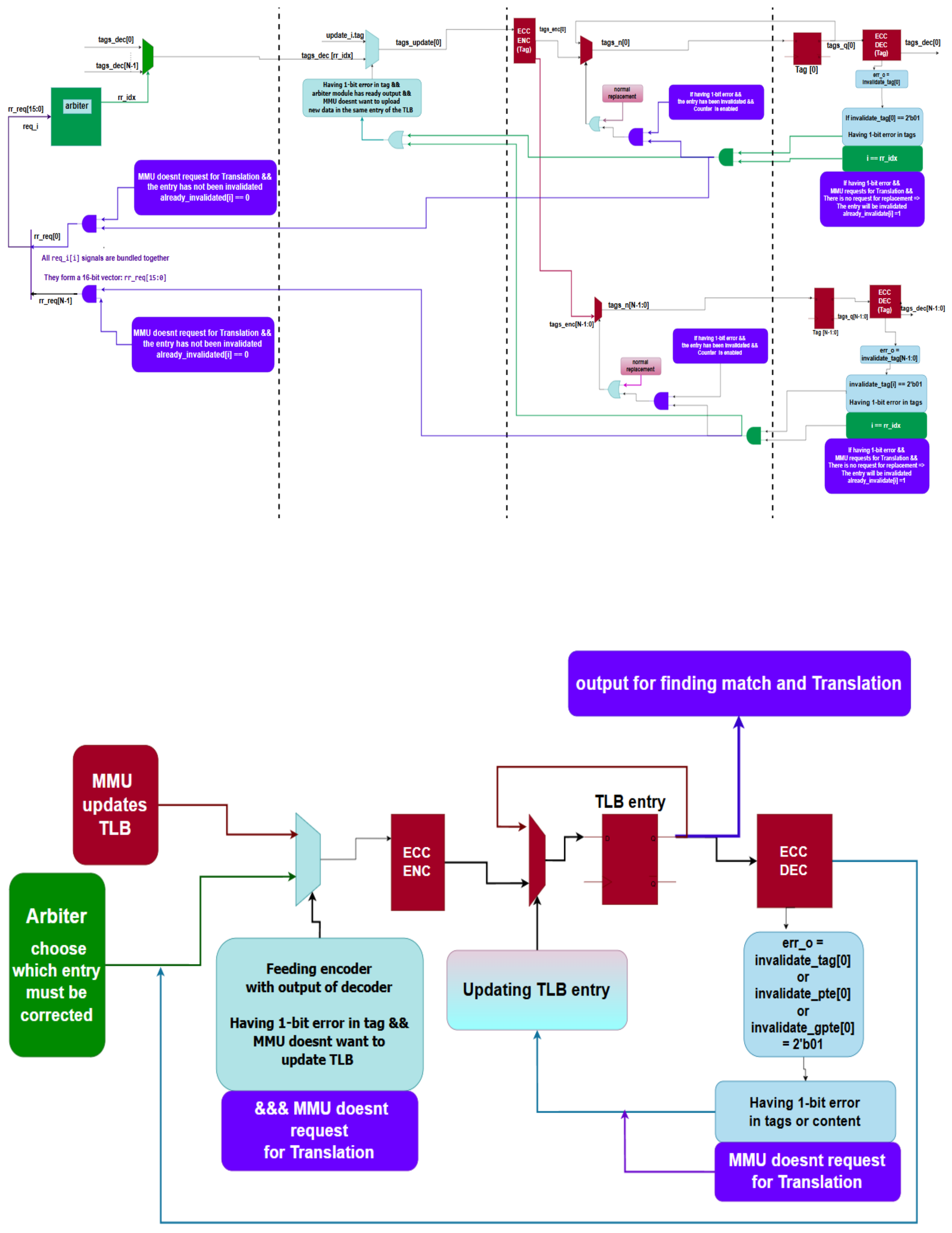

tags_dec[0]
tags_dec[N-1]
rr_idx
arbiter
rr_req[15:0]
req_i
MMU doesnt request for Translation && the entry has not been invalidated already_invalidated[i] == 0
rr_req[0]
All req_i[i] signals are bundled together
They form a 16-bit vector: rr_req[15:0]
rr_req[N-1]
MMU doesnt request for Translation && the entry has not been invalidated already_invalidated[i] == 0
update_i.tag
tags_dec [rr_idx]
tags_update[0]
Having 1-bit error in tag && arbiter module has ready output && MMU doesnt want to upload new data in the same entry of the TLB
ECC ENC (Tag)
tags_enc[0]
tags_n[0]
normal replacement
If having 1-bit error && the entry has been invalidated && Counter is enabled
Tag [0]
tags_q[0]
ECC DEC (Tag)
tags_dec[0]
err_o = invalidate_tag[0]
If invalidate_tag[0] == 2'b01
Having 1-bit error in tags
i == rr_idx
If having 1-bit error && MMU requests for Translation && There is no request for replacement => The entry will be invalidated already_invalidate[i] =1
tags_n[N-1:0]
tags_enc[N-1:0]
If having 1-bit error && the entry has been invalidated && Counter is enabled
normal replacement
Tag [N-1:0]
tags_q[N-1:0]
ECC DEC (Tag)
tags_dec[N-1:0]
err_o = invalidate_tag[N-1:0]
invalidate_tag[i] == 2'b01
Having 1-bit error in tags
i == rr_idx
If having 1-bit error && MMU requests for Translation && There is no request for replacement => The entry will be invalidated already_invalidate[i] =1
output for finding match and Translation
MMU updates TLB
TLB entry
ECC ENC
ECC DEC
Arbiter choose which entry must be corrected
Feeding encoder with output of decoder
Having 1-bit error in tag && MMU doesnt want to update TLB
&&& MMU doesnt request for Translation
Updating TLB entry
err_o = invalidate_tag[0] or invalidate_pte[0] or invalidate_gpte[0] = 2'b01
Having 1-bit error in tags or content
MMU doesnt request for Translation

Figure 4.8: Arbiter-based correction mechanism with Performance Enhancement for TLB extended with EC

5. Results

5.1 Fault Coverage

To validate the functionality and fault resilience of the TLB error detection and correction logic, first, RV64UI for different instruction test (like “and”, “add” or “xor”) from the official RISC-V tests have been executed in virtual environments. This test was chosen for its frequent memory accesses which exercise TLB lookups.

For observing the performance of the work, simulation has been run in Questasim, where it was possible to inject 1-bit or 2-bit errors manually on tags, content and valid bits on negative edges of the clock pulse on relevant waves to evaluate the error correction and entry invalidation capabilities.; Figure 5.1 shows a single-bit flip happening in the TLB tag Flip Flop (i.e. the tags_q bus). In the same clock cycle, the bit corresponding to the corrupted tag bitfield positions gets asserted in the invalidation bus associated to the faulty tag. After a clock cycle, the TLB correction logic restores the correct content of the TLB tag, restoring the invalidation and hit/miss states.

These tests have been done under both Detection Only and Correction modes across instruction and data TLBs (ITLB and DTLB).

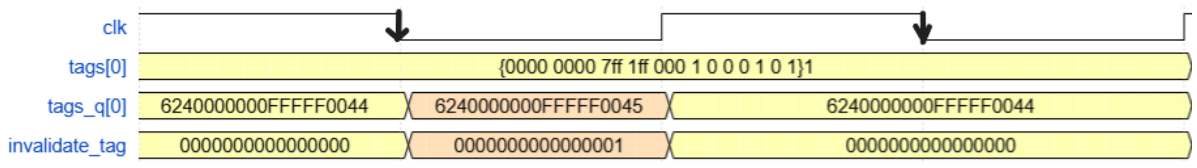


Figure 5.1: correction of 1-bit error in tag bits

If 2 bits are flipped in tag or content, the detection flag shows 2 for that entry and never changes because that entry will be invalidated as can be seen in Figure 5.2.

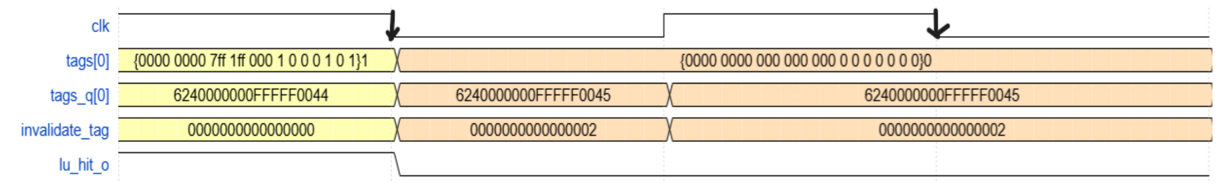


Figure 5.2, invalidate entry [0] with 2-bit error

Correction of the two entries, corrupted simultaneously, has been shown in Figure 5.3 and 5.4 based on counter-based and arbiter-based solutions, respectively.

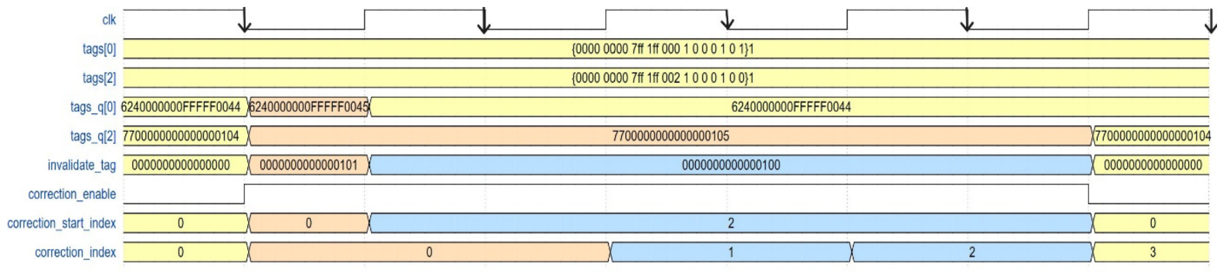


Figure 5.3: counter-based correction of 1-bit error in entry [0] and [2] simultaneously

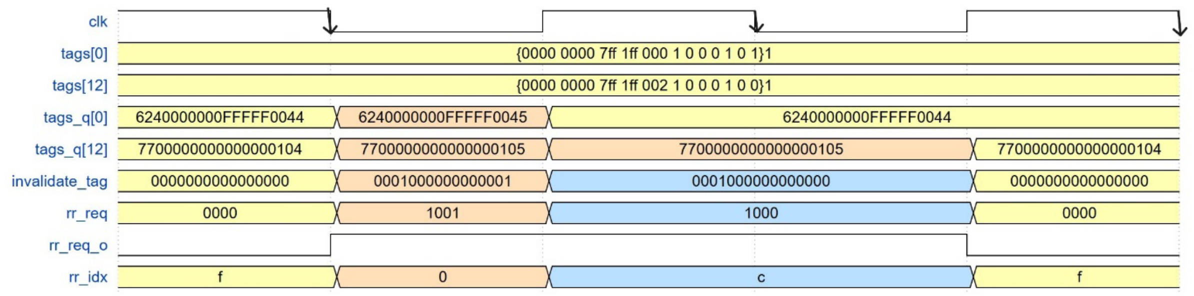


Figure 5.4: arbiter-based correction of 1-bit error in entry [0] and [12] simultaneously

On the contrary, in performance enhancement solution, when MMU is asking for translation, there will be no correction in tags, contents or valid bits, no matter if it is 1 bit of an error or more and if the invalidated entry was the one needed for translation, MMU will face a miss as Figure 5.5

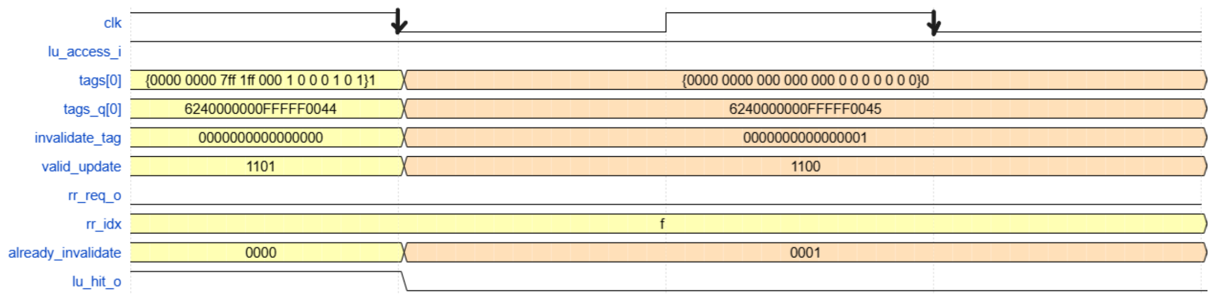


Figure 5.5: invalidation of 1-bit error in entry [0]
in arbiter-based performance enhancement

5.2 Frequency Results

We conducted a Power-Performance-Area (PPA) analysis of the proposed TLBs implementations using the GlobalFoundries 22nm FDSOI (GF22) technology. The synthesis and PPA assessments were performed under the slow-slow (SS) process corner at 0.72 V and –40 °C, which represents a worst-case scenario commonly used for maximum frequency testing. This ensures that the design can meet its timing constraints even under the most pessimistic conditions.

The baseline TLB design integrated with ECC but without any correction of TLB entries, had the ECC decoder in the critical path during MMU translation, and achieved a maximum frequency of 823 MHz. After applying architectural modifications to add correction mechanism as well as decouple the decoder output from the timing-critical path, the following results were observed as chart 5.1.

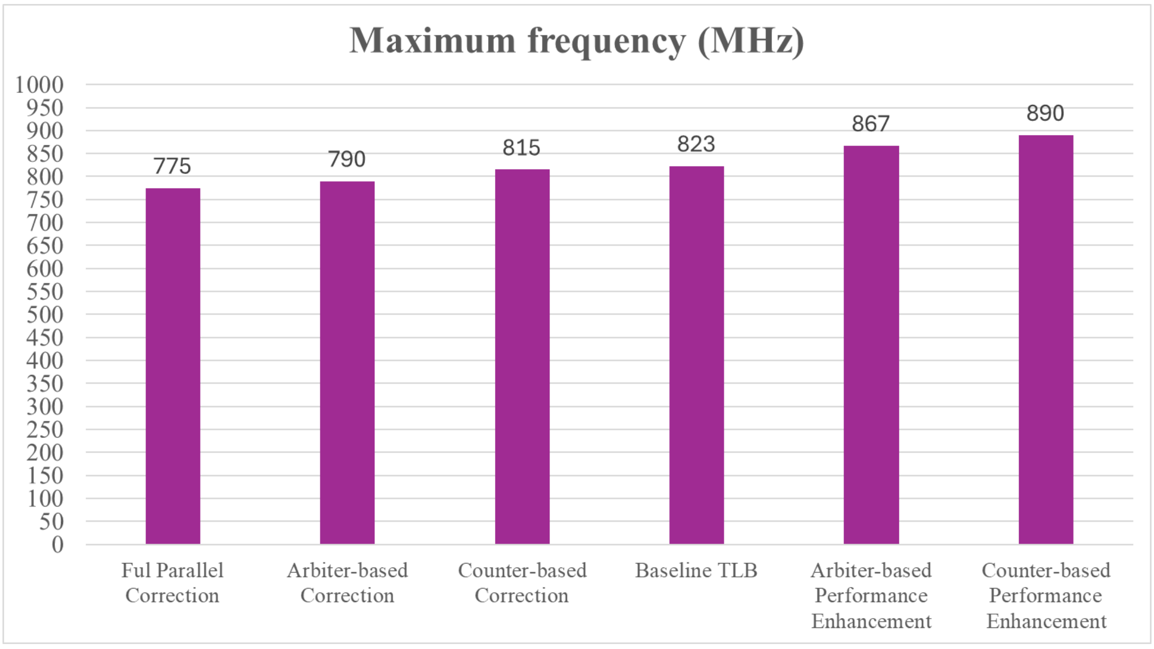


Chart 5.1, Comparison of frequency of different correction solutions based on Baseline TLB design

These results demonstrate an ~5.35% frequency gain (867Mhz) with the arbiter and an ~8.14% gain (890Mhz) with the counter method over the baseline TLB. The improved frequency confirms that eliminating decoder dependency during active MMU translation helps reduce logic depth and critical path delay. ECC decoding typically involves syndrome generation, bitwise XOR operations, and comparator trees, which introduce non-negligible combinational delay. This can significantly reduce the system's maximum achievable frequency, especially when multiple ECC bits are processed in parallel or complex codes like Hsiao are used. Such designs can result in persistent soft error correction being performed dynamically on every access, increasing both timing pressure and power consumption due to the reuse of slow ECC decoder logic in the pipeline's data path [70].

Furthermore, it is interesting that, although using arbiter instead of counter was supposed to remove the latency to count between simultaneous corrupted entries and be faster, it had less advancement in frequency compared to counter one. Despite both methods successfully removing ECC correction from the critical path during MMU translation, the arbiter-based solution showed a smaller improvement in maximum frequency. This difference stems from the intrinsic complexity of round-robin arbitration logic. Unlike the counter approach, which performs sequential corrections using minimal combinational logic, the arbiter introduces additional control structures for fairness, masking, and priority rotation. These structures typically involve tree-based grant computation and selection logic, which add to the overall logic depth and routing complexity of the circuit. Consequently, the arbiter module becomes a new contributor to the critical path, partially offsetting the gains achieved by bypassing the ECC decoder. This behavior aligns with observations in literature, where round-robin arbiters are recognized for their robustness but also for their timing cost due to increased gate count and combinational delay [71].

5.3 Area Results

All synthesis and PPA evaluations were also performed at the maximum frequency achievable by the baseline TLB design (823Mhz). The results are summarized in Table 5.1, where both the total CVA6 area of the design and the specific area occupied by the TLB subsystem (including both ITLB and DTLB modules) have been reported. Additionally, the resulted area for maximum frequency achieved by performance enhancement solutions have been illustrated in Table 5.2.

Correction Mechanism	Total CVA6 Area	TLB Area	dtlb Area	itlb Area	Total CVA6 Area Difference %	TLB Area Difference %
	(mm^2) (At 823Mhz)					
Baseline TLB	0.330	0.069	0.029	0.040	0	0
Arbiter-based Performance Enhancement	0.329	0.070	0.033	0.036	-0.30	0.86
Counter-based Performance Enhancement	0.333	0.069	0.033	0.036	0.19	0.19

Table 5.1, Comparison of Area for Performance Enhancement solutions with Baseline TLB design at Maximum Frequency of Baseline TLB

Correction Mechanism	Maximum Frequency (MHz)	Total CVA6 Area	TLB Area	dtlb Area	itlb Area	Total CVA6 Area Difference %	TLB Area Difference %
	(mm^2)						
Baseline TLB	**823**	0.33	0.069	0.029	0.04	0	0
Arbiter-based Performance Enhancement	**867**	0.334	0.073	0.035	0.038	1.25	4.35
Counter-based Performance Enhancement	**890**	0.331	0.067	0.031	0.036	0.41	-2.37

Table 5.2, Comparison of Area for Performance Enhancement solutions with Baseline TLB design at Maximum Frequency of each solution

The results show that the area overhead of each solution depends on the frequency constraint applied during implementation. At 823 MHz (baseline TLB frequency), the counter-based method exhibits a slight increase in both CVA6 and TLB area, while the arbiter-based approach shows a slight reduction in CVA6 area but increment in TLB area, even more than counter-based one. These results reflect the trade-off between centralized correction control and distributed arbitration logic as Charts 5.2. and 5.3

The counter-based solution introduces an FSM and a scanning counter to sequentially identify and correct entries with ECC errors. This increases the total logic and leads to an overall area increase compared to the baseline. However, since correction is serialized, it requires fewer additions inside each TLB entry, resulting in a more modest increase in TLB-specific area.

In contrast, the arbiter-based design employs a round-robin arbiter to allow parallel correction across entries. This reduces the need for centralized FSM control, leading to a decrease in the CVA6 area. However, each TLB entry must now incorporate logic to participate in arbitration (e.g., request generation, acknowledgment), which increases the local area of the TLB modules.

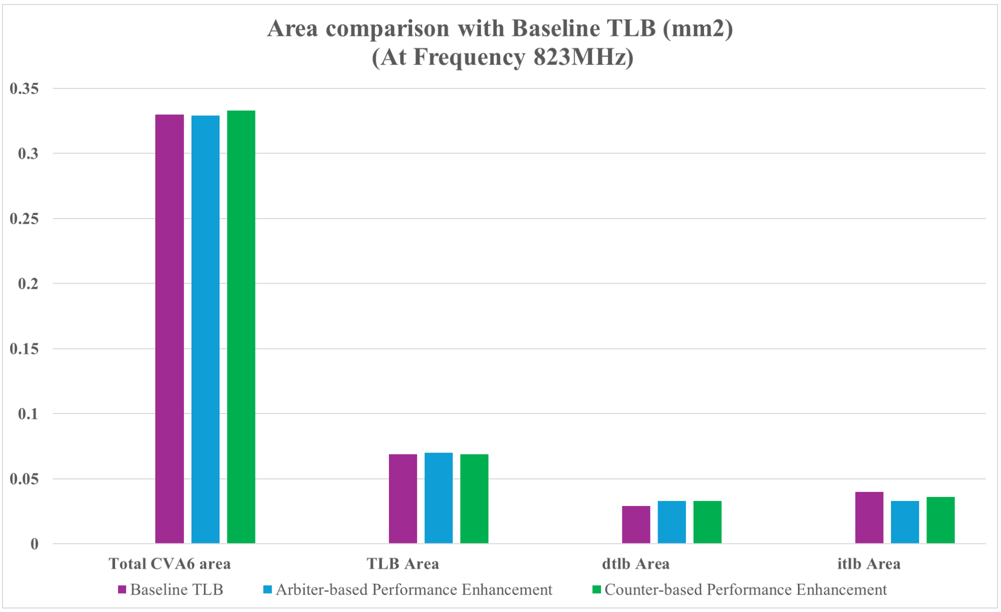


Chart 5.2: Comparison of Area among performance enhancement solutions and baseline TLB

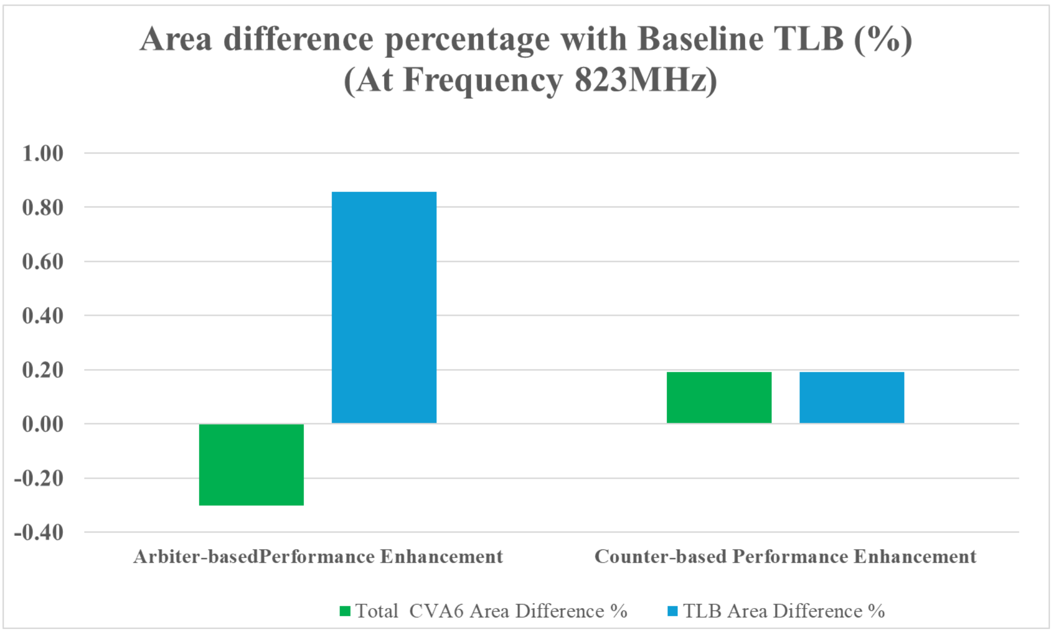


Chart 5.3: Comparison of Area difference percentage of Performance Enhancement solutions with baseline TLB

When each solution is synthesized at its own maximum achievable frequency, the counter-based solution becomes more area-efficient within the TLB subsystem (2.37% decrease), likely due to better logic sharing or retiming optimizations by the synthesis tool; However, the overall CVA6 area increases slightly, as more aggressive optimization is required across the entire processor to meet timing constraints, including the addition of pipeline stages, buffering, and high-speed cells outside the TLB subsystem.

Conversely, the arbiter-based method shows CVA6 area increase (~1.25%) and even more increment in TLB area (4.35%) which reflects the higher resource cost of meeting timing with distributed arbitration logic. The trade-off between frequency and area can be seen in chart 5.4.

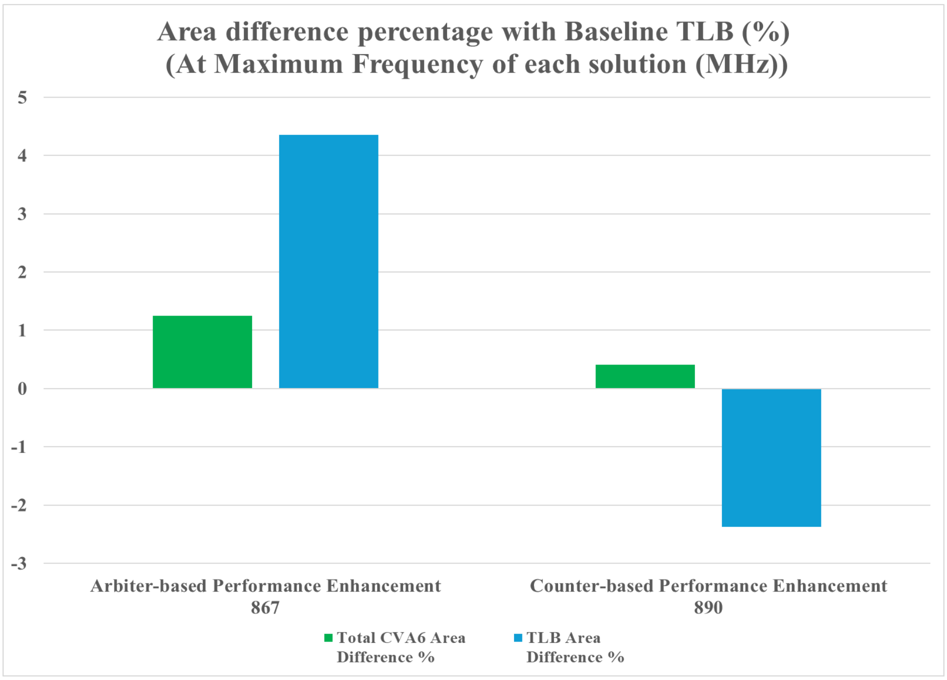


Chart 5.4: Comparison of Area difference percentage of Performance Enhancement solutions with baseline TLB

5.4 Efficiency

Despite the overall CVA6 area increase observed in the counter-based solution, the increment remains below 1% compared to the baseline TLB. This marginal cost is acceptable when weighed against the substantial frequency gain achieved, with an improvement of over 8%. However, this gain comes with a trade-off: when corrupted entries are in use, they are invalidated and trigger a page table walk (PTW), introducing a runtime penalty due to the additional cycles required to restore valid TLB entries. Nevertheless, in scenarios where maximizing clock frequency is a key priority, the counter-based approach remains attractive, offering a favorable balance between area, performance, and fault handling overhead.

6. Conclusion

This thesis presented a set of architectural and logic-level enhancements aimed at improving the reliability and timing performance of Translation Lookaside Buffers integrated with ECC support. The motivation originated from the need to protect TLB structures, vulnerable to soft errors due to their frequent access and compact size, without compromising the performance of the memory management pipeline, particularly in modern embedded and SoC-based RISC-V designs.

Starting from a suboptimal ECC-extended TLB integrated in the CVA6 RISC-V core, the work implemented and evaluated multiple correction-capable strategies. These included: a Detection-Only mode to support ECC decoding without correction, a fully parallel correction structure using combinational encoders, and two more scalable approaches, one based on a round-robin arbiter and the other on a correction counter with FSM control. Each solution was carefully integrated and synthesized using GlobalFoundries' 22nm FDSOI (GF22) technology to evaluate Power-Performance-Area (PPA) metrics.

Beyond fault correction, the thesis introduced a critical timing optimization: during MMU translation requests, ECC decoder outputs were bypassed entirely to eliminate their influence on the critical path. Instead of performing on-the-fly error correction, corrupted entries were invalidated immediately upon MMU access and repopulated via the PTW. This design ensured that matching and valid signals during TLB lookup were derived directly from raw data registers, avoiding latency penalties from ECC logic.

PPA evaluations showed promising results. The counter-based correction solution achieved the highest frequency gain of approximately 8.14%, increasing from 823 MHz (baseline TLB maximum frequency) to 890 MHz. The arbiter-based method, while slightly more complex in terms of logic, reached 867 MHz, representing a 5.35% improvement. The frequency uplift confirms that eliminating decoder dependencies during translation reduces logic depth and critical path delays. Area analysis further revealed that while both solutions introduced overhead to the total CVA6 area, their impact was minimal; however, Arbiter-based Performance Enhancement had considerable increment in TLB area in contrast with the Counter-based Performance Enhancement solution with 2.37% decrease in TLB area.

From a system integration perspective, all proposed designs were verified under realistic synthesis conditions (GF22, slow-slow (SS) process corner at 0.72 V and –40 °C) ensuring feasibility for high-frequency applications. The proposed ECC-extended TLB architecture not only enhanced resilience against soft errors but also reduced the performance penalty typically associated with ECC logic. Compared to traditional parity-based or detection-only designs, the Counter-based Performance Enhancement Solution offers a scalable, and timing-efficient framework suitable for fault-tolerant RISC-V processors.

Overall, the thesis demonstrates that combining correction mechanisms with targeted pipeline decoupling is an effective strategy for enhancing the reliability of the TLB integrated with ECC in high-performance systems without degrading operational frequency.

Bibliography

[1] S. Di Mascio, A. Menicucci, E. Gill, G. Furano, and C. Monteleone,“Open-source IP cores for space: A processor-level perspective on soft errors in the RISC-V era,” Computer Science Review, vol. 39, Feb. 2021, art. no. 100349. [Online]. Available: https: //www.sciencedirect.com/science/article/pii/S1574013720304494.

[2] A. Walsemann, M. Karagounis, A. Stanitzki, and D. Tutsch, “STRV— a radiation hard RISC-V microprocessor for high-energy physics applications,” Journal of Instrumentation, vol. 18, Feb. 2023, art. no. C02032. [Online]. Available: https://iopscience.iop.org/article/10.1088/ 1748-0221/18/02/C02032.

[3] N. Sridevi, K. Jamal and K. Mannem, "Implementation of Error Correction Techniques in Memory Applications," 2021 5th International Conference on Computing Methodologies and Communication (ICCMC), Erode, India, 2021, pp. 586-595, doi: 10.1109/ICCMC51019.2021.9418432.

[4] M. P. Stefani, Dynamic Fault Tolerant Mechanism for Memory Controllers, Ph.D. dissertation, Dept. of Computer Science, Pontifical Catholic Univ. of Rio Grande do Sul, Porto Alegre, Brazil, 2023.

[5] V. Kiani and P. Reviriego, "Improving Instruction TLB Reliability with Efficient Multi-bit Soft Error Protection," Microelectron. Reliab., vol. 93, pp. 29-38, Feb. 2019, doi: https://doi.org/10.1016/j.microrel.2018.12.011.

[6] A. Sánchez-Macián, L. A. Aranda, P. Reviriego, V. Kiani and J. A. Maestro, "Enhancing Instruction TLB Resilience to Soft Errors," in IEEE Transactions on Computers, vol. 68, no. 2, pp. 214-224, 1 Feb. 2019, doi: 10.1109/TC.2018.2874467.

[7] T. W. Griffith Jr and L. E. Thatcher, “TLB parity error recovery,” U.S. Patent No. 6,901,540, May 31, 2005.

[8] S. M. Lang, "Processor fault tolerance through translation lookaside buffer refresh," US- 8429135, Apr. 23, 2013 [Online]. Available:

https://scienceon.kisti.re.kr/srch/selectPORSrchPatent.do?cn=USP20130484291 35

[9] A. Sanchez-Macian, P. Reviriego, and J. A. Maestro, “Combined modular key and data error protection for content-addressable memories,” IEEE Trans. Comput., vol. 66, no. 6, pp. 1085–1090, Jun. 1, 2017, doi: 10.1109/TC.2016.2633998.

[10] I. Koren and C. M. Krishna, *Fault-tolerant systems*. Amsterdam Boston: Elsevier/Morgan Kaufmann, 2007.

[11] A. Aponte-Moreno, F. Restrepo-Calle, and C. Pedraza, "A Low-cost Fault Tolerance Method for ARM and RISC-V Microprocessor-based Systems using Temporal Redundancy and Approximate Computing through Simplified Iterations," *J. Integr. Circuits Syst.*, vol. 16, no. 3, pp. 1-14, 2021, doi: 10.29292/jics. v16i3.539.

[12] M. Nicolaidis, "Design for soft error mitigation," *IEEE Trans. Device Mater.* Reliab., vol. 5, no. 3, pp. 405-418, Sep. 2005, doi: 10.1109/TDMR.2005.855790.

[13] P. Shivakumar, M. Kistler, S. W. Keckler, D. Burger, and L. Alvisi, "Modeling the effect of technology trends on the soft error rate of combinational logic," in Proceedings International Conference on Dependable Systems and Networks, Washington, DC, USA: IEEE Comput. Soc, 2002, pp. 389-398. doi: 10.1109/DSN.2002.1028924.

[14] Alexander Aponte Moreno, Felipe Restrepo-Calle, and Cesar Pedraza, "Using Approximate Computing and Selective Hardening for the Reduction of Overheads in the Design of Radiation- Induced Fault-Tolerant Systems," *design Electronics*, vol. 8, no. 12, pp. 1-18, Dec. 2019, doi: 10.3390/electronics8121539.

[15] A. Makihara *et al.*, "Analysis of single-ion multiple-bit upset in high-density DRAMs," *IEEE Trans. Nucl. Sci.*, vol. 47, no. 6, pp. 2400-2404, Dec. 2000, doi: 10.1109/23.903783.

[16] T. Li, J. A. Ambrose, R. Ragel, and S. Parameswaran, "Processor Design for Soft Errors: Challenges and State of the Art," *ACM Comput Surv*, vol. 49, no. 3, p. 57:1-57:44, Nov. 2016, doi: 10.1145/2996357.

[17] S.M. Sze and Kwok K. Ng, *Physics of Semiconductor Devices*, Third Edition. United States of America: A JOHN WILEY & SONS, JNC., PUBLICATION, 2007.

[18] P. Georgiou, X. Kavousianos, R. Cantoro, and M. S. Reorda, "Fault-Independent Test- Generation for Software-Based Self-Testing," *IEEE Trans. Device Mater. Reliab.*, vol. 19, no. 2, pp. 341-349, Jun. 2019, doi: 10.1109/TDMR.2019.2911022.

[19] M. Rogenmoser, Y. Tortorella, D. Rossi, F. Conti, and L. Benini, "Hybrid Modular Redundancy: Exploring Modular Redundancy Approaches in RISC-V Multi-core Computing Clusters for Reliable Processing in Space," *ACM Trans Cyber-Phys Syst*, vol. 9, no. 1, p. 8:1-8:29, Jan. 2025, doi: 10.1145/3635161.

[20] John L. Hennessy and David A. Patterson. 2017. Computer Architecture, Sixth Edition: A Quantitative Approach (6th. ed.). Morgan Kaufmann Publishers Inc., San Francisco, CA, USA.

[21] Q. Huang and J. Jiang, "An overview of radiation effects on electronic devices under severe accident conditions in NPPs, rad-hardened design techniques and simulation tools," *Prog. Nucl. Energy*, vol. 114, pp. 105-120, Jul. 2019, doi: 10.1016/j.pnucene.2019.02.008.

[22] M. Tahoori, "Lecture 3: Faults, Errors, Failures," presented at the Reliable Computing I, University of the State of Baden-Wuerttemberg and National Laboratory of the Helmholtz Association, 2017. [Online]. Available: https://cdnc.itec.kit.edu/downloads/WS-17-18-Reliable-Computing lecture3_new.pdf

[23] IEC, *IEC 61508-1*. IEC, 2010.

[24] *Road vehicles — Functional safety*, ISO 26262-11, 2018.

[25] A. Munir, "Safety Assessment and Design of Dependable Cybercars: For today and the future, "IEEE Consum. Electron. Mag., vol. 6, no. 2, pp. 69-77, Apr. 2017, doi: 10.1109/MCE.2016.2640738.

[26] D. A. Gajewski, "Challenges and Peculiarities in Developing New Standards for SiC," in *IEEE International Reliability Physics Symposium (IRPS)*, 2020, pp. 1-5. doi: doi.org/10.1109/IRPS45951.2020.9128319.

[27] Nicolaidis, M. (eds) Soft Errors in Modern Electronic Systems. Frontiers in Electronic Testing, vol 41. Springer, Boston, MA. https://doi.org/10.1007/978-1-4419-6993-4_1

[28] ECSS. Techniques for Radiation Effects Mitigation in ASICs and FPGAs Handbook (1 September 2016) European Cooperation for Space Standardization; ESA Requirements and Standards Division: Noordwijk, The Netherlands,2016.

[29] Martínez-Álvarez, A.; Cuenca-Asensi, S.; Restrepo-Calle, F. Soft Error Mitigation in Soft-Core Processors. In FPGAs and Parallel Architectures for Aerospace Applications; Kastensmidt, F., Rech, P., Eds.; Springer International Publishing: Cham, Switzerland, 2016; Chapter 16, pp. 239–258

[30] K. Lilja et al., "Single-Event Performance and Layout Optimization of Flip-Flops in a 28-nm Bulk Technology," *IEEE Trans. Nucl. Sci.*, vol. 60, no. 4, pp. 2782-2788, Aug. 2013, doi: 10.1109/TNS.2013.2273437.

[31] T. D. Loveless et al., "Neutron- and Proton-Induced Single Event Upsets for D- and DICE- Flip/Flop Designs at a 40 nm Technology Node," IEEE Trans. Nucl. Sci., vol. 58, no. 3, pp. 1008-1014, Jun. 2011, doi: 10.1109/TNS.2011.2123918.

[32] Ming Zhang, Subhasish Mitra, T.M. Mak, and Norbert Seifert, "Sequential Element Design with Built-In Soft Error Resilience," *IEEE Trans. Very Large Scale Integr. VLSI Syst.*, vol. 14, no. 12, pp. 1368-1378, Jan. 2007, doi: 10.1109/TVLSI.2006.887832.

[33] Marcello Barbirotta, Abdallah Cheikh, Antonio Mastrandrea, Francesco Menichelli, and Mauro Olivieri, "Design and Evaluation of Buffered Triple Modular Redundancy in Interleaved-Multi- Threading Processors," *IEEE Access*, vol. 10, pp. 126074-126088, Dec. 2022, doi: 10.1109/ACCESS.2022.3225975.

[34] S. Alcaide, L. Kosmidis, C. Hernandez, and J. Abella, "Software-only based Diverse Redundancy for ASIL-D Automotive Applications on Embedded HPC Platforms," in *2020 IEEE International Symposium on Defect and Fault Tolerance in VLSI and Nanotechnology Systems (DFT)*, Frascati, Italy: IEEE, 2020, pp. 1-4. doi: 10.1109/DFT50435.2020.9250750.

[35] M. Rogenmoser et al., "Trikarenos: Design and Experimental Characterization of a Fault- Tolerant 28nm RISC-V-based SoC," 2025. doi: https://doi.org/10.48550/arXiv.2407.05938.

[36] P. Bernardi, R. Cantoro, S. D. Luca, E. Sanchez, and A. Sansonetti, "Development Flow for On- Line Core Self-Test of Automotive Microcontrollers," *IEEE Trans. Comput.*, vol. 65, no. 03, pp. 744-754, Mar. 2016, doi: 10.1109/TC.2015.2498546.

[37] Mihalis Psarakis, Dimitris Gizopoulos, Ernesto Sanchez, and Matteo Sonza Reorda, "Microprocessor Software-Based Self-Testing," *IEEE Des. Test Comput.*, vol. 27, no. 3, pp. 4- 19, Jan. 2010, doi: 10.1109/MDT.2010.5.

[38] V. Vargas, P. Ramos, J.-F. M~haut, and R. Velazco, "NMR-MPar: A Fault-Tolerance Approach for Multi-Core and Many-Core Processors," *Appl. Sci.*, vol. 8, no. 3, Art. no. 3, Mar. 2018, doi: 10.3390/app8030465.

[39] Christos Gkiokas and Martin Schoeberl, "A Fault-Tolerant Time-Predictable Processor," in *NORCHIP and International Symposium of System-on-Chip (SoC)*, Helsinki, Finland: IEEE Nordic Circuits and Systems Conference, Oct. 2019, pp. 1-6. doi: 10.1109/NORCHIP.2019.8906947.

[40] X. Iturbe, B. Venu, and E. Ozer, "Soft error vulnerability assessment of the real-

time safety- related ARM Cortex-R5 CPU," in *2016 IEEE International Symposium on Defect and Fault Tolerance in VLSI and Nanotechnology Systems (DFT)*, Storrs, CT, USA, Oct. 2016. doi: 10.1109/DFT.2016.7684076.

[41] Y. Zhang and K. Chakrabarty, "Fault recovery based on checkpointing for hard real-time embedded systems," in *Proceedings 18th IEEE Symposium on Defect and Fault Tolerance in VLSI Systems*, Boston, MA, USA: IEEE Xplore, Dec. 2003. doi: 10.1109/DFTVS.2003.1250127.

[42] X. Iturbe, Balaji Venu, E. Ozer, J.-L. Poupat, G. Gimenez, and Hans-Ulrich, "The Arm Triple Core Lock-Step (TCLS) Processor," *ACM Trans. Comput. Syst. TOCS*, vol. 36, no. 3, pp. 1-30, Jun. 2019, doi: 10.1145/3323917.

[43] M. S. Sadi, MD. P. Molla, A. Asraf, and Md. L. Ali, "Enhancing the Capability of Single Error Correction and Double Error Detection Method," in *2nd International Conference on Information and Communication Technology (ICICT)*, Dhaka, Bangladesh, 2024, pp. 249-253. doi: 10.1109/ICICT64387.2024.10839685.

[44] F. S. Alghareb, R. A. Ashraf, and R. F. DeMara, "Designing and Evaluating Redundancy-based Soft Error Masking on a Continuum of Energy versus Robustness," *IEEE Trans. Sustain. Comput.*, vol. 3, no. 3, pp. 139-152, Sep. 2018, doi: 10.1109/TSUSC.2017.2764857.

[45] Infineon Technologies AG, “32-bit AURIX™ TriCore™ Microcontroller Family –Product Overview.” [Online]. Available: https://www.infineon.com/cms/en/product/microcontroller/32-bit-tricore-microcontroller/. [Accessed: Jul. 2, 2025].

[46] AN5342, "How to use error correction code (ECC) management for internal memories protection on STM32 MCUs." STMicroelectronics, Feb. 2024. [Online]. Available: https://www.st.com/resource/en/application_note/an5342--how-to-use-error-correction-code-ecc-management-for-internal-memories-protection-on-stm32-mcus-stmicroelectronics.pdf

[47] L.-J. Saiz-Adalid, P. Gil, J. Gracia-Moran, D. Gil-Tomas, and J.-C. Baraza-Calvo, "Ultrafast Single Error Correction Codes for Protecting Processor Registers," in *2015 11th European Dependable Computing Conference (EDCC)*, Paris, France: IEEE Xplore, Sep. 2015. doi: 10.1109/EDCC.2015.30.

[48] R. Alom, N. Shakib, and M. A. Rahaman, "Enhanced Hamming Codes: Reducing Redundant Bit for Efficient Error Detection and Correction," in *2023 5th International Conference on Sustainable Technologies for Industry 5.0 (STI)*, Dhaka, Bangladesh: IEEE, Dec. 2023. doi: 10.1109/STI59863.2023.10465152.

[49] N. A. Koca, C. H. Chang, A. T. Do, and V. P. Nambiar, "Exploring Error Correction Circuits on RISC-V based Systems for Space Applications," in *2024 IEEE International Symposium on Circuits and Systems (ISCAS)*, Singapore, Singapore: IEEE, May 2024, pp. 1-5. doi: 10.1109/ISCAS58744.2024.10558401.

[50] L. J. Saiz, P. Gil-Vicente, J. C. Ruiz, J. Gracia, D. Gil, and J.-C. Baraza-Calvo, "Ultrafast Error Correction Codes for Double Error Detection/Correction," presented at the 2016 12th European Dependable Computing Conference (EDCC), Sep. 2016. doi: 10.1109/EDCC.2016.28.

[51] M. Neri, "Design of a fault tolerant instruction decode stage in RISC-V core against soft and hard errors," Research, Politecnico di Torino, Italy, 2021. [Online]. Available: https://webthesis.biblio.polito.it/17871/1/tesi.pdf

[52] L.-J. Saiz-Adalid, J. Gracia-Moran, D. Gil-Tomas, J.-C. Baraza-Calvo, and P.-J. Gil-Vicente, "Ultrafast Codes for Multiple Adjacent Error Correction and Double Error Detection," *IEEE Access*, vol. 7, pp. 151131-151143, 2019, doi: 10.1109/ACCESS.2019.2947315.

[53] Alexander Dörflinger, Yejun Guan, Sören Michalik, Sönke Michalik, Jamin Naghmouchi, and Harald Michalik, "ECC Memory for Fault Tolerant RISC-V Processors," presented at the Lecture Notes in Computer Science, Springer, Cham, Jul. 2020. doi: 10.1007/978-3-030-52794- 5_4.

[54] T. Satyanarayana, V. A. Qureshi and G. Divya, "Design and implementation of

error detection and correction system for semiconductor memory applications," 7th International Conference on Computing in Engineering & Technology (ICCET 2022), Online Conference, 2022, pp. 325-330, doi: 10.1049/icp.2022.0641.

[55] F. Conti, "Big Data Analytics for Automotive Manufacturing Applications - Module 2." Statistics and Architectures for Big Data Processing- Module 2, Nov. 17, 2023. [Online]. Available: https://www.unibo.it/en/study/course-units-transferable-skills-moocs/course-unit- catalogue/course-unit/2023/472948

[56] Sá, Bruno, et al. "CVA6 RISC-V virtualization: Architecture, microarchitecture, and design space exploration." IEEE Transactions on Very Large Scale Integration (VLSI) Systems 31.11 (2023): 1713-1726

[57] A. Efthymiou, "An error tolerant CAM with nand match-line organization," in *GLSVLSI '13: Proceedings oj the 23rd ACM international conference on Great lakes symposium on VLSI*, New York, NY, USA: Association for Computing Machinery, May 2013, pp. 257-262. doi:10.1145/2483028.2483105.

[58] K. Pagiamtzis, N. Azizi, and F. N. Najm, "A Soft-Error Tolerant Content-Addressable Memory (CAM) Using an Error-Correcting-Match Scheme," in *IEEE Custom Integrated Circuits Conference 2006*, San Jose, CA, USA: IEEE, Sep. 2006, pp. 301-304. doi:10.1109/CICC.2006.320887.

[59] S. Baeg, S. Wen, and R. Wong, "Minimizing Soft Errors in TCAM Devices: A Probabilistic Approach to Determining Scrubbing Intervals," *IEEE Trans. Circuits Syst. Regul. Pap.*, vol. 57, no. 4, pp. 814-822, Jun. 2009, doi: 10.1109/TCSI.2009.2025856.

[60] D. Luong, M. Goshima, and S. Sakai, "Mitigating soft errors in highly associative cache with CAM-based tag," in Proceedings oj the 2005 International Conjerence on Computer Design (ICCD'05), IEEE Xplore, Nov. 2005, pp. 1-6. doi: 10.1109/ICCD.2005.76.

[61] S. Pontarelli, M. Ottavi, and A. Salsano, "Error Detection and Correction in Content Addressable Memories," in *2010 IEEE 25th International Symposium on Defect and Fault Tolerance in VLSI Systems*, Kyoto, Japan: IEEE, 2010, pp. 420-428. doi: 10.1109/DFT.2010.56.

[62] S. Pontarelli and M. Ottavi, "Error Detection and Correction in Content Addressable Memories by Using Bloom Filters," *IEEE Trans. Comput.*, vol. 62, no. 6, pp. 1111-1126, 2013, doi:10.1109/TC.2012.56.

[63] J. A. Maestro, P. Reviriego, S. Baeg, S. Wen, and R. Wong, "Soft error tolerant Content Addressable Memories (CAMs) using error detection codes and duplication," *Microprocess. Microcyst.*, vol. 37, no. 8 (D), pp. 1103-1107, 2013, doi: 10.1016/j.micpro.2013.10.003.

[64] R. Tedeschi *et al.*, "CVA6S+: A Superscalar RISC-V Core with High-Throughput Memory Architecture," Paris: reprint arXiv:2505.03762, 2025. doi: 10.48550/arXiv.2505.03762.

[65] Y. Tortorella, "Dual-Core Lockstep (DCLS) implementation of an application processor for fault tolerance", PowerPoint presentation, Università di Bologna, Italy.

[66] A. Ottaviano, T. Benz, P. Scheffler and L. Benini, "Cheshire: A Lightweight, Linux-Capable RISC-V Host Platform for Domain-Specific Accelerator Plug-In," in IEEE Transactions on Circuits and Systems II: Express Briefs, vol. 70, no. 10, pp. 3777-3781, Oct. 2023, doi: 10.1109/TCSII.2023.3289186

[67] L. Valente et al., "A Heterogeneous RISC-V Based SoC for Secure Nano-UAV Navigation," in IEEE Transactions on Circuits and Systems I: Regular Papers, vol. 71, no. 5, pp. 2266-2279, May 2024, doi: 10.1109/TCSI.2024.3359044.

[68] F. Zaruba and L. Benini, "The Cost of Application-Class Processing: Energy and Performance Analysis of a Linux-Ready 1.7-GHz 64-Bit RISC-V Core in 22-nm FDSOI Technology," in *IEEE Transactions on Very Large-Scale Integration*

(VLSI) Systems, vol. 27, no. 11, pp. 2629-2640, Nov. 2019, doi: 10.1109/TVLSI.2019.2926114.

[69] Z. Fu *et al.*, "Ramping Up Open-Source RISC-V Cores: Assessing the Energy Efficiency of Superscalar, Out-of-Order Execution," in *22nd ACM International Conference on Computing Frontiers (CF '25)*, Cagliari, Italy, 2025. doi: 10.1145/3719276.3725186.

[70] J. Juffinger, Rowhammer Exploits are still possible, master’s thesis, Graz Univ. of Technol., Inst. for Appl. Inf. Process. and Commun., Graz, Austria, Sep. 2021.

[71] J. Luo, W. Wu, Q. Xing, M. Xue, F. Yu, and Z. Ma, "A low-latency fair-arbiter architecture for network-on-chip switches," *Appl. Sci.*, vol. 12, no. 23, Art. no. 12458, Dec. 2022. doi: 10.3390/app122312458.