POLITECNICO DI TORINO

Department of Electronics and Telecommunications

Master's Degree in Electronic Engineering

Enhancing Security and Performance of Post-Quantum Cryptographic Signatures on Ramps

Supervisor:
Prof. Guido MASERA
Ph.D. Alessandra Dolmeta
Valeria Piscopo

Candidate:
Giuseppe Cutrera

Academic Year 2026

Abstract

The increasing threat posed by quantum computing to classical public-key cryptography has accelerated the development of Post-Quantum Cryptographic (PQC) schemes. Within this context, the National Institute of Standards and Technology (NIST) has run multiple standardization rounds for post-quantum schemes. One notable candidate from recent rounds is the Linear Equivalence Signature Scheme (LESS), a code-based signature construction whose security relies on the hardness of the Linear Code Equivalence problem.

LESS derives signatures by applying the Fiat–Shamir transform to a zero-knowledge identification protocol, whose main computations manipulate large generator matrices over finite fields and apply monomial isometries. As a result, the scheme relies heavily on linear algebra operations, with the reduction of the generator matrix to the Reduced Row Echelon Form (RREF) representing a major bottleneck. This thesis presents the design and implementation of a hardware accelerator targeting the RREF computation within the LESS PQC algorithm, featuring pivot-element reuse. The proposed RREF engine is integrated as a loosely coupled external hardware accelerator within the RISC-V–based X-HEEP microcontroller, which serves as the target embedded platform for system-level validation. The architecture leverages direct memory access (DMA) for high-throughput data movement and an interrupt-driven control scheme to minimize processor idle time and synchronization overhead. The accelerator operates on GF(127) arithmetic and is optimized for on-chip memory, resource constraints, and scalable matrices.

A key design objective was to explore the trade-off between area and computational latency. The RREF algorithm and microarchitecture were therefore carefully tailored to achieve a balanced operating point suitable for resource-constrained FPGA deployments. This trade-off-oriented design enables meaningful comparison with state-of-the-art LESS hardware implementations in terms of performance efficiency and resource utilization. The design was synthesized and implemented on a Zynq UltraScale+ ZCU104 FPGA board, where resource utilization, timing performance, and memory footprint were thoroughly evaluated. A memory layout optimized for parallel access was employed, using a word-parallel organization to pack multiple elements per access, to balance BRAM usage and computational throughput.

The accelerator communicates with the host processor through a memory-mapped interface, while bulk data transfers are handled via DMA and execution completion is signaled through interrupts. This integration approach enables efficient overlap between computation and data movement, reduces processor idle time, and preserves system modularity without requiring modifications to the processor pipeline.

Experimental evaluations show over 20× speedup for the single RREF matrix-reduction function call across all LESS security levels (up to 21.16×), while maintaining low hardware overhead. On the ZCU104 platform, the design requires at most 9.2% LUTs, 6.9% FFs, and 12.8% BRAM and 61.5% DSPs of the complete system, demonstrating an efficient performance–area trade-off for embedded PQC applications.

Index Terms: Post-Quantum Cryptography, LESS, RREF, Hardware Acceleration.

Contents

List of Tables

List of Figures

Acronyms

PQC	Post-Quantum Cryptography
NIST	National Institute of Standards and Technology
ISA	Instruction Set Architecture
DMA	Direct Memory Access
LESS	Linear Equivalence Signature Scheme
LEP	Linear Equivalence Problem
RREF	Reduced Row Echelon Form
RV-PLIC	RISC-V Platform Level Interrupt Controller
X-HEEP	eXtendable Heterogeneous Energy-Efficient Platform
CV-X-HEEP	X-HEEP using the CoreV eXtension Interface (XIF)
SoC	System-On-Chip
MMIO	Memory-Mapped I/O
FPGA	Field Programmable Gate Array
KAT	Known Answer Test

1 Introduction

1.1 Background on Cryptography

Cryptography has existed since ancient times and has long served as the cornerstone of secure communications. The key idea behind cryptography is to build and analyze protocols that prevent third parties from reading private messages. Prior to the modern age, cryptography was effectively synonymous with encryption, intended as a conversion from readable information (**plaintext**) to an unintelligible nonsense text (**ciphertext**) which can only be read by reversing the process (**decryption**). Modern cryptography has deeply evolved from the past especially since the introduction of the first computers, which allowed for more complex algorithms heavily based on mathematical theory. This shift coincided with the explosive growth of the digital era where interconnected networks, e-commerce, and data proliferation skyrocketed demand for robust cryptographic protections across communications, storage, and transactions.[1]

1.2 The advent of Quantum Computing

Until recent times, the security of modern cryptographic systems has relied on the practical intractability of specific mathematical problems when solved using classical computers. The most common Public-key cryptographic schemes (such as RSA [2], Diffie–Hellman key exchange [3], and Elliptic Curve Cryptography (ECC) [4, 5]) derive their security from problems including *integer factorization* and the *discrete logarithm problem*, which are believed to be computationally infeasible to solve within a reasonable time frame using conventional computing resources. For decades, this assumption has been the foundation of global digital security, supporting secure web communication, digital signatures, and authentication systems.
However this assumption has now been challenged by the advent of quantum computing. Unlike classical computers, which process information in binary form, quantum computers exploit quantum-mechanical phenomena such as superposition and entanglement to perform computations in fundamentally different ways. In particular, quantum algorithms have been shown to provide exponential or polynomial speedups for certain classes of problems. Most notably, **Shor's algorithm** [6] demonstrates that a sufficiently powerful quantum computer could efficiently solve both integer factorization and discrete logarithm problems, thereby breaking the security of widely deployed public-key cryptosystems such as RSA and ECC (as shown in Table 1.1). Although large-scale, fault-tolerant quantum computers capable of executing such

Table 1.1: Impact of quantum computing on classical cryptographic primitives.

Primitive	**Quantum Impact**	**Mitigation**
RSA / ECC / DH (Asymmetric or public-key)	**Broken** (Shor's algorithm)	Migrate to **PQC**
AES and hash functions (Symmetric)	**Weakened** (Grover's algorithm)	Increase key/output size (AES-256 or SHA-512)

attacks are not yet available, significant progress in quantum hardware and algorithms has intensified concerns regarding long-term cryptographic security. Data encrypted today using vulnerable public-key schemes may be recorded and stored by adversaries, to be decrypted in the future once quantum capabilities mature – a threat commonly referred to as the "*harvest now, decrypt later*" attack [7, 8].
Symmetric cryptographic primitives (AES [9, 10], SHA [11]), including block ciphers and hash functions, are comparatively less affected by quantum attacks. While *Grover's algorithm* [12] provides a quadratic speedup, its impact can be mitigated by appropriately increasing security parameters, making symmetric cryptography a viable option even in a post-quantum context [13, 14].
In contrast, public-key cryptographic schemes (**Asymmetric**) are significantly more vulnerable, as quantum algorithms like Shor's can efficiently solve the underlying mathematical problems on which their security is based, with no equivalent mitigation through parameter scaling [6]. This vulnerability is further amplified by the growing accessibility of quantum computing resources, which are increasingly available to the public through cloud-based platforms rather than being confined to specialized research or government facilities.
These developments highlight the urgent need to transition public-key infrastructures toward Post-Quantum Cryptography, ensuring long-term security against both classical and quantum threats, while maintaining efficient and practical implementations.[15]

1.3 Post-Quantum Signatures

Digital signatures are cryptographic primitives that enable a party to authenticate the origin and integrity of a message in a public-key setting. A digital signature scheme typically consists of three algorithms:

- **Key Generation** (`KeyGen`): generates a pair of keys consisting of a public verification key and a private signing key.
- **Signing** (`Sign`): using the private key, the signer produces a signature on a given message.
- **Verification** (`Verify`): using the public key, anyone can verify whether the signature is valid for the given message.

Digital signatures provide three fundamental security properties: *authentication*, since the signature can only be produced by the holder of the private key; *integrity*, since any modification of the signed message invalidates the signature; and *non-repudiation*, since the signer cannot deny having produced a valid signature. In response to this emerging threat of Quantum Computers, the U.S. National Institute of Standards and Technology (NIST) initiated the Post-Quantum Cryptography (PQC) Standardization Process in 2016, with the goal of evaluating and standardizing quantum-resistant public-key algorithms. The process consists of multiple competitive rounds, during which candidate schemes are assessed with respect to security, performance, implementation efficiency, and resistance to side-channel attacks.
The first phase of the NIST PQC standardization process aimed at identifying secure replacements for classical public-key cryptographic primitives vulnerable to quantum attacks. For digital signatures, the process resulted in the standardization of three schemes: the lattice-based constructions **CRYSTALS-Dilithium** [16] and **Falcon** [17], and the hash-based scheme **SPHINCS+** [18]. These algorithms represent the first generation of standardized post-quantum signature schemes and provide practical solutions for secure digital signatures in the presence of quantum adversaries.
In parallel with the main PQC standardization effort, NIST launched a separate initiative known as the *Additional Digital Signature Schemes* project. The goal of this process is to evaluate alternative signature constructions based on different cryptographic assumptions and design principles, broadening the range of potential post-quantum signature solutions.
The candidate algorithms submitted to this initiative cover a wide range of cryptographic paradigms. In particular, the second round of evaluation includes schemes belonging to several different families:

- **Code-based signatures**, including **CROSS** and **LESS**, which rely on hard problems from coding theory;
- **Isogeny-based signatures**, such as **SQIsign**, which exploit the difficulty of computing isogenies between elliptic curves;
- **Lattice-based constructions**, such as **HAWK**, which explore alternative lattice-based designs;
- **Multivariate signatures**, including **UOV**, **QR-UOV**, **SNOVA**, and **MAYO**, based on the difficulty of solving systems of multivariate polynomial equations;
- **MPC-in-the-Head signatures**, such as **Mirath**, **MQOM**, **PERK**, **RYDE**, and **SDitH**, which derive signatures from interactive zero-knowledge protocols;
- **Symmetric-based signatures**, such as **FAEST**, which combine symmetric cryptographic primitives with zero-knowledge proof techniques.

These additional candidates allow researchers to explore a broader set of cryptographic constructions and evaluate their security, performance, and implementation characteristics.

However, many post-quantum signature schemes involve computationally intensive operations on large algebraic structures, which can make efficient implementations challenging, especially on embedded platforms. From an implementation perspective, improving the efficiency of these algorithms can therefore benefit from the use of *hardware accelerators*. A hardware accelerator is a specialized computing module designed to execute specific computational tasks more efficiently than a general-purpose processor. By offloading the most computationally intensive operations to dedicated hardware logic, it is possible to reduce execution time, improve energy efficiency, and increase overall system performance. In order to investigate the practical implementation characteristics of these signatures, several candidate algorithms were analyzed. Because the objective of this work is the design of an hardware accelerator targeting embedded platforms, particular attention was given to identifying algorithms whose computational workload is concentrated in a limited number of dominant operations. Algorithms exhibiting this structure are especially well suited for hardware acceleration, as optimizing a small set of computational kernels can significantly reduce the overall execution time of the scheme allowing us to focus on just one or two functions for a single accelerator.
The profiling analysis was carried out using the **ASIP Designer** framework, which provides cycle-accurate simulation and profiling capabilities for software execution on configurable processor architectures. The results of this analysis are presented in chapter 2 and motivated the selection of the algorithm targeted by the hardware accelerator developed in this thesis.

1.4 RISC-V ISA and Zynq UltraScale+ FPGA

The hardware accelerator developed in this work is designed to operate within a modern embedded system environment based on the **RISC-V** Instruction Set Architecture (ISA) and later implemented on a **Xilinx Zynq UltraScale+ FPGA** platform.
RISC-V [19] is an open and modular instruction set architecture originally developed at the University of California, Berkeley. Unlike proprietary architectures, RISC-V is freely available and allows both academic and industrial researchers to design custom processors and extensions without licensing restrictions. The architecture follows a reduced instruction set computing (RISC) philosophy, emphasizing a small set of simple instructions that can be efficiently implemented in hardware.
One of the key advantages of RISC-V is its extensibility. The ISA is organized into a minimal base instruction set complemented by optional extensions that enable additional functionality such as integer multiplication, floating-point arithmetic, vector processing, and custom instruction extensions. This modular structure makes RISC-V particularly suitable for research and hardware prototyping, where custom accelerators can be integrated alongside the processor.
In this thesis, the hardware accelerator is integrated within a RISC-V-based embedded platform derived from the **X-HEEP** microcontroller system [20]. X-HEEP provides a lightweight system-on-chip environment including a RISC-V processor core, memory subsystems, and peripheral interfaces. The accelerator is connected as a loosely coupled hardware peripheral, allowing the processor to offload computa-

tionally intensive operations while maintaining a clear separation between software control and hardware execution.
The complete system is implemented and evaluated on a **Xilinx Zynq UltraScale+ ZCU104 FPGA development board**. Field Programmable Gate Arrays (FPGAs) provide a flexible hardware platform that allows digital circuits to be synthesized, configured, and tested without the need for fabrication of custom integrated circuits. The Zynq UltraScale+ architecture combines programmable logic with high-performance processing capabilities, making it well suited for rapid prototyping of hardware accelerators and embedded systems.
Using this platform enables the proposed architecture to be evaluated under realistic hardware conditions, including synthesis results, timing analysis, resource utilization, and execution performance. These measurements provide insight into the practical feasibility of deploying the proposed accelerator within resource-constrained embedded systems.

1.5 Thesis Structure

The structure of this thesis is organized as follows:

- **Chapter 2** introduces the *Linear Equivalence Signature Scheme (LESS)* and presents the cryptographic background required to understand its construction. The chapter first describes the core principles of the algorithm, including linear codes, monomial transformations, and the signature protocol. It then analyzes the *Reduced Row Echelon Form (RREF)* operation used within LESS and discusses the pivot reuse optimization. Finally, the reference software implementation of the RREF routine is presented.

- **Chapter 3** presents the hardware architecture of the proposed RREF accelerator. The design choices, internal modules, and control mechanisms are described in detail, together with the strategies adopted to efficiently perform matrix reduction over the finite field GF(127).

- **Chapter 4** describes the integration of the accelerator within the X-HEEP embedded platform. The chapter explains the system integration process, including the register interface, wrapper design, system-level connectivity, the FuseSoC integration flow, and the development of the software driver used to control the accelerator.

- **Chapter 5** reports the experimental results obtained from the hardware implementation. The evaluation includes cycle-level performance measurements, FPGA resource utilization, timing analysis, power consumption, and comparisons with other implementations.

- **Chapter 6** summarizes the main contributions of the work and discusses potential directions for future research.

- **Appendix A** provides additional material including code snippets and implementation details that support the hardware and software components described throughout the thesis.

2 LESS Algorithm/signature

Before designing a hardware accelerator, several post–quantum digital signature schemes were analyzed using the profiling capabilities of **ASIP Designer**. The goal of this analysis was to identify algorithms whose execution time is dominated by a small number of computational kernels, making them suitable candidates for architectural acceleration.
The profiling process consisted of executing the reference implementations of several candidate signature schemes and measuring the distribution of clock cycles across the different functions of the signing procedure. The execution inputs were derived from the official *Known Answer Tests* (**KATs**) provided with the reference implementations. KATs are deterministic test vectors that specify fixed inputs and the corresponding expected outputs of the cryptographic algorithm, allowing implementations to be verified against known correct results and ensuring reproducible execution during profiling.
Among the analyzed algorithms, the **Linear Equivalence Signature Scheme (LESS)** algorithm exhibited a particularly favorable structure from a hardware acceleration perspective.
The profiling results for the LESS signing method are shown in Table 2.1. The `generator_RREF_pivot_reuse` function accounts for approximately 87.73% of the total execution cycles. This indicates that the performance of the entire signing procedure is largely determined by a single computational kernel.

Table 2.1: Function-level profiling of the LESS signing method obtained using ASIP Designer.

Function	**Cycles [%]**	**Role**
`generator_RREF_pivot_reuse`	87.73%	RREF matrix reduction kernel
`compute_canonical_form_type4`	2.52%	Canonical form computation
`compute_canonical_form_type5_popcnt`	2.08%	Canonical form computation
`KeccakF1600_StatePermute`	1.50%	Hash permutation
Others	6.17%	Remaining functions (sorting, memory operations, hashing helpers)

Since the vast majority of the execution time is spent inside a single function, optimizing this kernel can lead to significant overall performance improvements. This characteristic makes LESS an ideal candidate for hardware acceleration, as improving the performance of the matrix reduction routine directly impacts the total latency of

the signing process.
For this reason, LESS was selected as the target algorithm for the design of the accelerator presented in this work.
Although LESS has already been the subject of a dedicated hardware implementation, most notably in the work of Beckwith et al. [21], the design objective of the present work is substantially different. The implementation proposed by Beckwith et al. [21] targets the acceleration of the LESS signature scheme as a whole, resulting in a custom hardware architecture tailored to the complete execution of the protocol.
In contrast, the goal of this thesis is not to build a standalone hardware implementation of LESS, but rather to design a lightweight accelerator that can be integrated into a RISC-V based system and that targets only the computational kernel dominating the execution time. Focusing the hardware design on this single function enables a more efficient use of hardware resources compared to implementing the entire signature scheme in dedicated hardware.
This choice is particularly important in the context of **X-HEEP** [20], an open-source embedded system-on-chip platform based on a RISC-V microcontroller architecture designed for research and prototyping of hardware extensions and accelerators (that will be explained in more detail in Chapter 4). Within such a platform, the accelerator must coexist with a general-purpose embedded system and must therefore satisfy stricter integration and resource constraints than a fully custom standalone design.
Consequently, the contribution of this work is not to replace existing full-hardware implementations of LESS, but to explore a more modular and resource-conscious acceleration strategy, in which only the main computational bottleneck is offloaded to hardware while the rest of the algorithm remains in software.
To better understand the computational structure of the LESS signature scheme and the role of the RREF operation within it, the following section provides an overview of the algorithm.

2.1 Overview of the Algorithm

The LESS algorithm is a code–based post-quantum digital signature scheme whose security relies on the hardness of the *Linear Equivalence Problem* (LEP) [22]. The scheme was first introduced by Biasse et al. [23] and later extended and refined in subsequent works by Barenghi et al. [24] and Persichetti [22, 25]. The most recent version of the scheme was submitted to the NIST post-quantum cryptography standardization process [26].
Code-based cryptography relies on the difficulty of particular problems in coding theory, which are believed to remain hard even for quantum computers. Unlike more traditional code-based signatures, LESS is built using a different framework based on **code equivalence** and **sigma protocols**.

2.1.1 Linear Codes and Generator Matrices

A fundamental concept in coding theory is the **linear code**. An $[n, k]$ linear code over a finite field $\mathbb{F}_q$ is a k-dimensional subspace of $\mathbb{F}_q^n$. In practice, such a code is

represented by a **generator matrix**

$$G \in \mathbb{F}_q^{k \times n}$$

whose rows span the vector space defining the code. A message $m \in \mathbb{F}_q^k$ can be encoded as a codeword

$$c = mG$$

where $c \in \mathbb{F}_q^n$ is the resulting codeword.
The same code can also be described using a *parity-check matrix* $H \in \mathbb{F}_q^{(n-k) \times n}$ which satisfies

$$Hc^T = 0$$

for every valid codeword c. Generator and parity-check matrices therefore provide two equivalent representations of the same code.
For many applications it is useful to convert the generator matrix to a unique representation. One common representation is the **systematic form**

$$G = (I_k \mid M)$$

where I_k is the $k \times k$ identity matrix and $M \in \mathbb{F}_q^{k \times (n-k)}$. When a generator matrix is in systematic form, the first k coordinates of every codeword correspond directly to the message, making the code systematic in those positions.
In practice, converting a generator matrix into systematic form requires performing Gaussian elimination on the matrix. This process produces a canonical representation known as the *Reduced Row Echelon Form* (RREF). The RREF representation is unique for a given matrix and therefore provides a convenient way to compare codes.

2.1.2 Permutation and Monomial Transformations

The LESS signature scheme is based on the idea that two linear codes can represent the same structure even if their generator matrices look different. In particular, one generator matrix can be transformed into another by applying a sequence of algebraic transformations that do not change the underlying code.
These transformations operate in two different ways. First, it is possible to perform **row operations** on the generator matrix. Such operations simply change the basis used to represent the code, without modifying the set of codewords generated by the matrix. In matrix form, these operations correspond to multiplying the generator matrix on the left by an invertible matrix

$$S \in GL_k(q),$$

where $GL_k(q)$ denotes the set of all invertible $k \times k$ matrices over the finite field $\mathbb{F}_q$.
Second, it is possible to modify the **coordinates of the codewords**. This means reordering the positions of the symbols in the codewords or scaling individual coordinates by non–zero elements of the field. These transformations correspond to multiplying the generator matrix on the right by a special class of matrices known as *monomial matrices*.

To understand this, it is useful to first consider **permutation matrices**. A permutation matrix contains exactly one entry equal to 1 in each row and each column. Multiplying a matrix by such a matrix simply reorders its columns, which corresponds to permuting the coordinates of every codeword.
A **monomial matrix** generalizes this concept. It has the same structure as a permutation matrix, but the non–zero entries can be any non–zero element of $\mathbb{F}_q$. As a result, multiplying a generator matrix by a monomial matrix both permutes the columns and scales each column by a non–zero field element.
These transformations preserve the Hamming weight of the vectors and therefore do not change the essential properties of the code. For this reason, they are referred to as **isometries** of the code.
Using this notation, the Linear Equivalence Problem (LEP) can be formulated as follows. Given two generator matrices

$$G, G' \in \mathbb{F}_q^{k \times n},$$

the problem is to determine whether there exist matrices

$$S \in GL_k(q), \qquad P \in M_n$$

such that

$$G' = SGP.$$

Here M_n denotes the set of $n \times n$ monomial matrices over $\mathbb{F}_q$. If such matrices exist, the two codes are said to be **linearly equivalent**.
In the LESS signature scheme, the secret key essentially corresponds to such a transformation. The public key is obtained by applying a secret monomial transformation to a known generator matrix and then converting the result into RREF. Recovering the hidden transformation between the two matrices is believed to be computationally hard, and this hardness forms the basis of the security of LESS.

2.1.3 LESS Signature Construction

The LESS signature scheme is constructed by first defining a **zero-knowledge identification protocol** based on the LEP and then transforming this interactive protocol into a non–interactive digital signature scheme using the Fiat–Shamir transformation [27]. The underlying interactive protocol belongs to the class of **Sigma protocols** [28], which are widely used in cryptography to prove knowledge of a secret without revealing the secret itself.
A Sigma protocol is a three–move interactive protocol consisting of a *commitment*, a *challenge*, and a *response*. It involves two parties:

- the **prover**, who possesses a secret value,
- the **verifier**, who wishes to confirm that the prover knows the secret.

Sigma protocols satisfy three fundamental security properties:

- **Completeness**: an honest prover who knows the secret can always convince the verifier.

- **Soundness**: a dishonest prover cannot convince the verifier without knowing the secret, except with negligible probability.
- **Zero-knowledge**: the interaction does not reveal any information about the secret itself.

In the case of LESS, the secret corresponds to a monomial transformation that maps one generator matrix to another equivalent matrix. The goal of the protocol is therefore to allow the prover to demonstrate knowledge of this transformation without revealing it.
Let G_0 be a public generator matrix. The prover's secret key is a monomial matrix

$$Q \in M_n$$

while the public key is derived as

$$G_1 = \mathrm{RREF}(G_0 Q).$$

Since Q is secret, an external observer only sees two generator matrices G_0 and G_1 that generate equivalent codes, without knowing the transformation linking them.

Commitment phase. To start the protocol, the prover samples a random monomial transformation $\tilde{Q}$ and computes

$$\tilde{G} = \mathrm{RREF}(G_0 \tilde{Q}).$$

The matrix $\tilde{G}$ acts as a temporary commitment. Instead of sending the matrix directly, its hash is transmitted to the verifier in order to bind the prover to this value.

Challenge phase. The verifier then generates a random challenge bit $c \in \{0, 1\}$ and sends it to the prover. This challenge determines which information the prover must reveal in the next step.

Response phase. The prover answers the challenge in one of two possible ways. If $c = 0$, the prover reveals the random transformation $\tilde{Q}$. The verifier checks that the commitment corresponds to the matrix

$$\tilde{G} = \mathrm{RREF}(G_0 \tilde{Q}).$$

If $c = 1$, the prover instead reveals

$$Q^{-1}\tilde{Q}.$$

The verifier can then verify the commitment using the public matrix G_1:

$$G_1 Q^{-1} \tilde{Q} = (G_0 Q) Q^{-1} \tilde{Q} = G_0 \tilde{Q}.$$

In both cases, the verifier can confirm that the commitment is valid, but learns nothing about the secret transformation Q.

Security intuition. A dishonest prover who does not know the secret Q cannot prepare a commitment that allows answering both possible challenges correctly. Therefore, such a prover can only guess the challenge in advance. Since the challenge is a single bit, the probability of cheating successfully in one round is $1/2$.
To reduce this probability to a negligible value, the protocol is repeated multiple times. Each additional round halves the probability that a dishonest prover can successfully impersonate the legitimate signer.

Fiat–Shamir transformation. The protocol described above is interactive, meaning that it requires communication between the prover and the verifier. To convert it into a practical digital signature scheme, this interaction must be eliminated.
The Fiat–Shamir transformation [27] achieves this by replacing the verifier's random challenge with the output of a cryptographic hash function. Instead of waiting for the verifier to send the challenge, the signer computes

$$c = H(\text{commitments}, m),$$

where H is a cryptographic hash function and m is the message being signed. The resulting hash value deterministically generates the sequence of challenges that would normally be provided by the verifier.
In this way, the signer can simulate the entire interaction locally. The signature consists of the commitments together with the responses corresponding to the generated challenges. Any verifier can recompute the hash and verify that the responses are consistent with the commitments.
This transformation removes the need for interaction while preserving the security guarantees of the identification protocol, resulting in a non-interactive digital signature scheme.

LESS Signature Scheme

The complete LESS signature scheme is composed of three main algorithms: **key generation**, **signing**, and **verification**.

Key Generation During the key generation phase, a public and private key pair is generated. A public generator matrix G_0 is first defined. The signer then samples several secret monomial transformations $Q_i \in M_n$. For each transformation, the corresponding public matrix is computed as

$$G_i = \text{RREF}(G_0 Q_i).$$

The matrices G_i form the public key (pk), while the corresponding monomial transformations Q_i (or their seeds) constitute the private key (sk). The use of the RREF representation ensures that each generator matrix has a unique canonical form, allowing different equivalent matrices to be compared consistently.

Signing To generate a signature on a message m, the signer simulates several rounds of the Sigma protocol. For each round, a random monomial transformation $\tilde{Q}_i$ is sampled and a commitment matrix is computed as

$$\tilde{G}_i = \mathrm{RREF}(G_0 \tilde{Q}_i).$$

These commitment matrices are then encoded and hashed together with the message being signed. The resulting hash value is used to derive the sequence of challenges according to the Fiat–Shamir transformation [27].
Depending on the value of each challenge, the signer reveals either the random transformation $\tilde{Q}_i$ or a transformation derived from the secret key. The collection of responses, together with the challenge seed, forms the final digital signature.

Verification During verification, the verifier reconstructs the challenges by re-hashing the commitments and the message. Using the responses contained in the signature, the verifier recomputes the expected commitment matrices by performing the corresponding transformations and converting the resulting matrices into RREF form.
The verifier then hashes the reconstructed commitments and checks that the resulting challenge matches the one included in the signature. If the values match, the verifier is convinced that the signer must know the secret monomial transformations linking the public matrices.

Since each of these phases repeatedly requires converting matrices of the form $G_0 Q$ into RREF, the matrix reduction routine becomes one of the dominant computational components of the LESS implementation. For this reason, optimizing the RREF operation is essential for improving the overall performance of the LESS implementation and constitutes the primary focus of the accelerator design presented in this work.

2.1.4 LESS Parameters

The LESS scheme defines several parameter sets corresponding to the NIST security levels. These parameters include the code dimensions n and k, the finite field size q, and the protocol parameters t, ω, and s [26]. The parameter t defines the number of protocol repetitions, ω defines the number of non-zero challenges, and s defines the number of pairs of generator matrices included in the public key.
The different parameter sets aim at optimizing different metrics of the signature scheme. As reported in Table 2.2, the balanced parameter sets, denoted by the suffix b, aim to minimize the combined size of the public key and the signature. The small parameter sets, denoted by s, are designed to minimize the signature size, at the cost of larger public keys. For security level 1, an additional intermediate configuration is provided, denoted by i, which offers a compromise between the two optimization goals. Also pk denotes the size of the public key, while sig denotes the size of the resulting digital signature for the corresponding parameter set.

Table 2.2: Parameter sets for LESS and resulting data sizes.[21]

NIST Cat.	Parameter Set	Code Params			Prot. Params			pk (KiB)	sig (KiB)
		n	k	q	t	ω	s		
1	LESS-1b	252	126	127	247	30	2	13.7	8.4
	LESS-1i				244	20	4	41.1	6.1
	LESS-1s				198	17	8	95.9	5.2
3	LESS-3b	400	200	127	759	33	2	34.5	18.4
	LESS-3s				895	26	3	68.9	14.1
5	LESS-5b	548	274	127	1352	40	2	64.6	32.5
	LESS-5s				907	37	3	129.0	26.1

2.2 RREF Function

The Reduced Row Echelon Form (RREF) operation is a fundamental linear algebra procedure used to transform a matrix into a canonical form. In the context of the LESS signature scheme, the RREF operation is applied to generator matrices in order to obtain a unique representation of a linear code. This canonical representation makes it possible to compare matrices that represent equivalent codes, since two generator matrices representing the same code will produce the same RREF representation.
Given a matrix $G \in \mathbb{F}_q^{k \times n}$, the RREF algorithm systematically performs elementary row operations until the matrix satisfies the following properties:

- the first non–zero element of each non–zero row (the *pivot*) is equal to 1,
- each pivot is the only non–zero element in its column,
- pivots appear in strictly increasing column order when moving from the first row to the last.

When these conditions are satisfied, the matrix is said to be in *row reduced echelon form*. Since this representation is unique, it can be used to identify when two matrices correspond to the same linear code.

2.2.1 RREF Algorithm

The standard algorithm for converting a $k \times n$ matrix to RREF is based on Gaussian elimination. The typical complexity of this operation is $O(nk^2)$, since up to k pivot rows must be processed, and each reduction step involves operations across the $k \times n$ matrix.
The reduction procedure consists of four main steps that are repeated for each row of the matrix:

1. **Pivot search.** The algorithm first searches for a pivot element. The pivot is the leftmost non–zero element that can be used to eliminate other elements in its column. The search is performed within the portion of the matrix that has not yet been reduced.

2. **Row swap.** If the pivot is not already located in the current row, the row containing the pivot is swapped with the row currently being processed.

3. **Pivot normalization.** The pivot row is then rescaled so that the pivot element becomes equal to 1. This is done by multiplying the entire row by the multiplicative inverse of the pivot element in the finite field $\mathbb{F}_q$.

4. **Row reduction.** Finally, the pivot row is used to eliminate all other elements in the same column. This is achieved by subtracting suitable multiples of the pivot row from the other rows of the matrix.

After completing these steps, the algorithm moves to the next row and repeats the procedure until all k rows have been processed. An example of the RREF transformation process is illustrated in Figure 2.1, which shows how successive pivot operations progressively reduce the matrix.

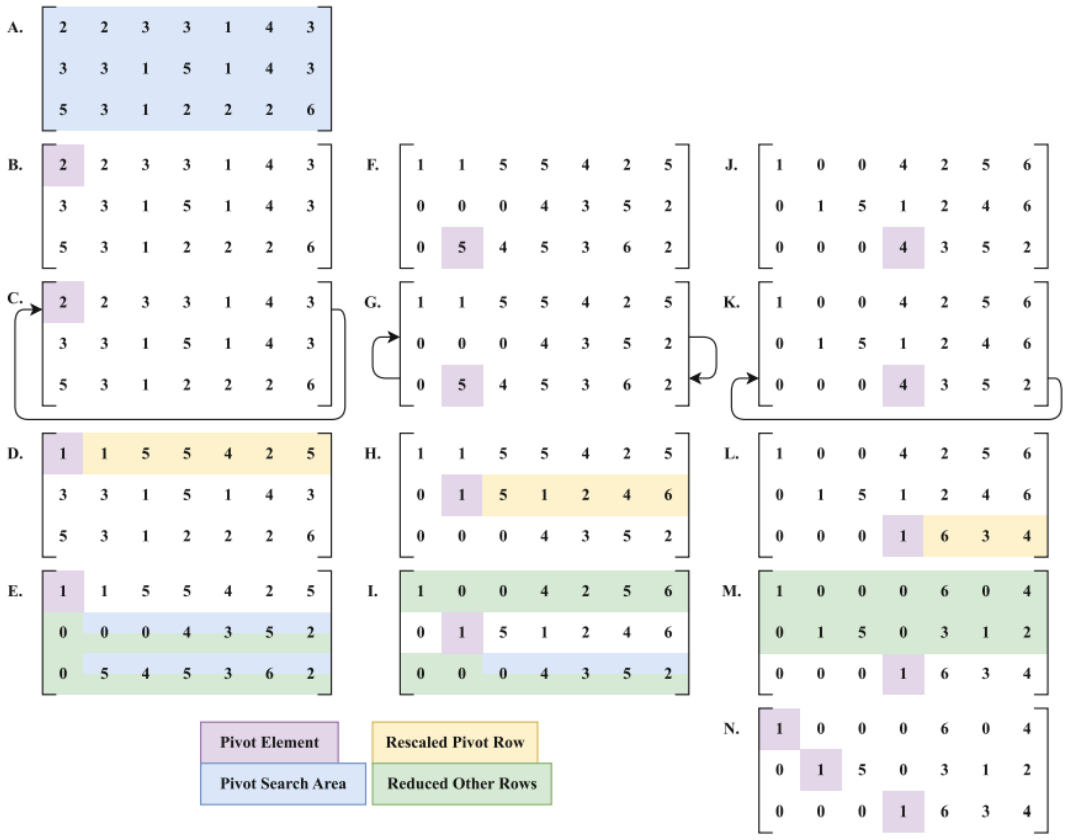

Figure 2.1: RREF Example: n = 7, k = 3, q = 7 [21]

2.2.2 Pivot Reuse Optimization

In the LESS signature scheme, the RREF operation is applied to matrices of the form

$$G_0 Q,$$

where G_0 is a generator matrix already in RREF form and Q is a monomial transformation. A monomial transformation corresponds to permuting and scaling the columns of the matrix.
Because the columns of G_0 already contain pivot positions, many of these pivots remain valid even after the column permutation and scaling performed by Q. As a result, when computing the RREF of G_0Q, some pivot columns are already correctly reduced and do not require the full elimination procedure.
The pivot reuse optimization exploits this observation by detecting columns that already contain valid pivot elements. When such columns are encountered, the algorithm can skip the corresponding row reduction step, avoiding unnecessary arithmetic operations. In practice, a significant fraction of pivots can be reused in this way, which substantially reduces the computational cost of the RREF operation.

2.2.3 Applicability in LESS

The pivot reuse optimization is particularly effective during the **signing** and **verification** phases of the LESS algorithm. In these phases, the matrices being reduced are derived from transformations of the public generator matrix G_0, which is already in RREF form. Since these matrices are obtained by applying monomial transformations to G_0, many pivot positions are preserved and can therefore be reused.
In contrast, during the **key generation** phase the generator matrices are produced from randomly generated inputs that are not guaranteed to preserve any pivot structure. Consequently, pivot reuse cannot be reliably applied, and the standard RREF algorithm must be used.
Because the signing algorithm performs this reduction many times for each signature, the RREF computation with pivot reuse becomes the dominant computational component of the LESS implementation. This makes the RREF operation a natural target for optimization in high-performance implementations of the scheme.

2.3 Software Implementation

The LESS reference implementation performs the matrix reduction operations using a software routine written in C. The function responsible for computing the reduced row echelon form with pivot reuse is shown in Appendix A.1. This routine, named `generator_RREF_pivot_reuse`, implements the RREF algorithm described in the previous section while introducing an optimization that allows previously discovered pivot positions to be reused when possible.
The function operates directly on a generator matrix

$$G \in \mathbb{F}_q^{K \times N}$$

stored in the structure `generator_mat_t`. The matrix is modified in place and converted to its reduced row echelon form.
The function receives four parameters:

- **G**: the generator matrix to be reduced to RREF.

- **is_pivot_column**: an array used to mark which columns become pivot columns during the reduction.
- **was_pivot_column**: an array indicating which columns were pivot columns in a previous RREF computation and may therefore be reused.
- **pvt_reuse_limit**: a parameter that limits the number of pivots that can be reused during the reduction process.

The function returns 1 if the reduction succeeds and 0 if the algorithm fails to find a valid pivot during the reduction.

2.3.1 Preprocessing Step

Before the main reduction begins, the algorithm performs a preprocessing step intended to improve the effectiveness of pivot reuse. If pivot reuse is enabled (i.e., `pvt_reuse_limit` is non–zero), the algorithm scans the columns that previously contained pivot elements. For each of these columns, the routine searches for the row containing the pivot element and swaps it with the row that will be reduced in the current iteration.
This preprocessing stage ensures that previously identified pivot elements are placed in positions that maximize the probability that they can be reused without being corrupted during the subsequent row operations.

2.3.2 Pivot Search

The main reduction loop iterates over each row of the matrix. For each row, the algorithm searches for a valid pivot element. The search starts from the current column index and scans the remaining columns until a non–zero element is found. Within each column, the algorithm scans the rows below the current row until a non–zero entry is identified.
If no pivot element is found in the remaining submatrix, the reduction cannot continue and the function returns a failure value.
Once a pivot element is identified, the corresponding column is marked as a pivot column in the `is_pivot_column` array.

2.3.3 Row Swap

If the pivot element is located in a row different from the row currently being reduced, the algorithm performs a row swap. This operation moves the pivot element into the correct row position so that the reduction process can proceed correctly.
If a row swap occurs, the corresponding entry in `was_pivot_column` is cleared, since the pivot structure may no longer be reusable after the row operations that follow.

2.3.4 Pivot Reuse

After locating the pivot, the algorithm checks whether the pivot can be reused from a previous reduction. This condition is verified using three criteria:

- the column was previously a pivot column,
- the number of reused pivots has not exceeded the `pvt_reuse_limit`,
- the pivot column lies within the leading K columns of the matrix.

If these conditions are satisfied, the algorithm skips the pivot row normalization and row elimination steps.
By skipping these operations, the algorithm avoids a large number of finite-field arithmetic operations, significantly reducing the overall computational cost of the RREF computation.

2.3.5 Pivot Row Normalization

If pivot reuse cannot be applied, the algorithm proceeds with the standard RREF procedure. The pivot row is first normalized so that the pivot element becomes equal to 1.
This is achieved by computing the multiplicative inverse of the pivot element in the finite field $\mathbb{F}_q$ using the function `fq_inv`. Each element of the pivot row is then multiplied by this inverse using the finite-field multiplication function `fq_mul`.

2.3.6 Row Elimination

Once the pivot row has been normalized, the algorithm eliminates the entries in the pivot column for all other rows of the matrix. For each row, a multiplier equal to the element in the pivot column is computed. The algorithm then subtracts the appropriate multiple of the pivot row from the current row.
These operations are performed using finite-field multiplication and subtraction routines, `fq_mul` and `fq_sub`, respectively.
This process ensures that the pivot column contains zeros in all rows except the pivot row, satisfying one of the defining properties of the reduced row echelon form.

3 Hardware Implementation

This chapter presents the hardware implementation of the RREF accelerator over GF(127), derived from the LESS algorithm for efficient cryptographic matrix processing. The parameterized architecture supports arbitrary generator matrix dimensions and covers LESS security Categories 1, 3, and 5, enabling applicability across multiple security levels.

Hardware accelerators can be broadly classified as **tightly coupled** or **loosely coupled**. In a tightly coupled architecture, the accelerator is deeply integrated within the processor pipeline or memory hierarchy and interacts with the host at fine-grained instruction level, often sharing registers or cache structures. Conversely, a loosely coupled accelerator operates as an independent compute engine, communicating with the host through explicit command, status, and data-transfer interfaces, typically handling coarse-grained computational tasks autonomously.

The proposed RREF engine adopts a loosely coupled accelerator paradigm. This choice is motivated by the computational structure of the RREF procedure, which consists of long, deterministic, and data-intensive matrix transformations with limited need for fine-grained interaction with the host processor. A tightly coupled integration would introduce unnecessary synchronization overhead and tighter constraints on the processor pipeline and memory hierarchy, without providing significant performance benefits for this workload.

By contrast, a loosely coupled approach allows the accelerator to execute complete row-reduction tasks autonomously, improving modularity, scalability across different matrix dimensions, and portability across FPGA-based systems. This design decision therefore prioritizes architectural simplicity, area efficiency, and clean system-level integration over fine-grained host–accelerator interaction.

Adapting LESS's core principles, the design reuses pivot elements from previous cycles to minimize redundant computations while striking a deliberate tradeoff between area efficiency and latency. Particular emphasis is placed on parallel computation, efficient memory usage, and deterministic behavior, which makes the accelerator well-suited for high-throughput cryptographic and coding-theory applications on FPGA platforms targeted at resource-limited environments.

The design employs a hierarchical **SystemVerilog** structure with parameterization for matrix dimensions and field parameters, enabling tuning for diverse workloads. Design trade-offs between area and latency are validated through **Vivado** synthesis flows and **ModelSim** verification, starting with software-based simulations, then targeting the Xilinx UltraScale+ ZCU104 FPGA for real timing and cycle counts.

This chapter details the top-level architecture, internal modules, and the key architectural decisions that shape the final implementation.

3.1 Overview of the accelerator

From a system integration perspective, the proposed design follows the **loosely coupled** paradigm introduced above and is implemented as an autonomous hardware compute engine. At the interface level, the RREF accelerator exposes explicit control and status registers together with streaming data channels for matrix input and output. The host processor initiates the computation by configuring the matrix dimensions and security parameters, then transfers the input data block to the accelerator memory space. Once triggered, the accelerator executes the complete row-reduction procedure without further host intervention. Internal control logic sequences the pivot selection, row scaling, and elimination phases deterministically, while the host remains decoupled from cycle-level synchronization. This execution model enables a clear separation between software orchestration and hardware computation. The processor is responsible only for task configuration and data movement, whereas the accelerator autonomously manages all computational phases, exploiting internal parallelism and local memory resources to sustain deterministic high-throughput operation.
The RREF accelerator follows the structure of the reference software implementation of LESS[29], mapping each algorithmic phase onto a finite state machine (FSM)-based control unit while exploiting hardware parallelism in pivot search and elimination stages.
As illustrated in Figure 3.1, the main internal modules are:

- **CU (Control Unit)**: The FSM-based core of the accelerator acts as its brain, activating and utilizing the other internal modules precisely when required and resetting them afterward.
- **G_MEM (Generator Matrix Memory)**: BRAM-based storage for the input matrix G (preferred implementation over full register one after initial prototypes exceeded LUT/register limits even for smallest category sizes)
- **Arithmetic Unit**: Parallel GF(127) operators for multiplication, addition/ subtraction, and multiplicative inversion.
- **WAS/IS Pivot Registers**: Dedicated storage for pivot position tracking and search state across cycles
- **Temporary Registers**: Pipeline buffering for intermediate row operations

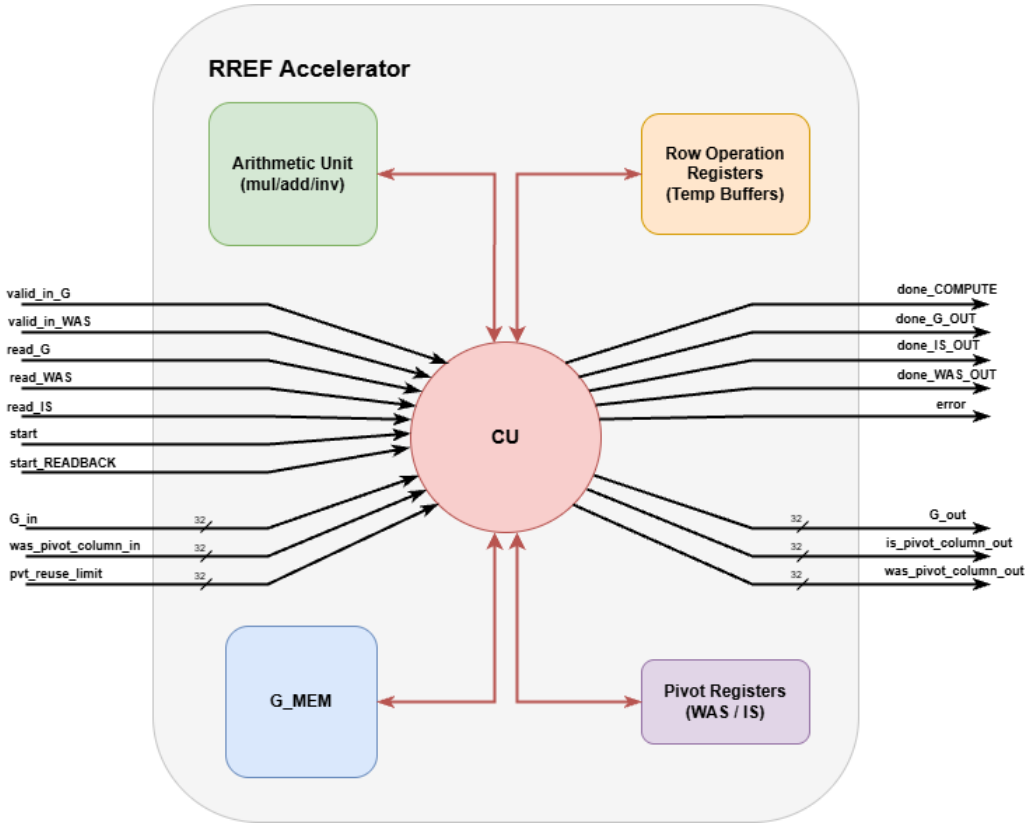

Figure 3.1: Top-level architecture of the RREF accelerator over GF(127), showing the I/O connections to the outside and an overview of the inside modules. Clock and Reset signals have been omitted.

Figure 3.1 also highlights the external interface of the accelerator. The upper control signals on the left side consist of single-bit handshake and control lines, including *start*, download-to-accelerator *valid*, and upload-from-accelerator *read* signals used to coordinate data flow and computation phases.
The lower input ports form a 32-bit streaming interface used for DMA-style data transfers. These inputs carry the generator matrix data (G), the previously detected pivot map (*was_pivot*), and the pivot reuse limit parameter (*pvt_reuse_limit*), enabling efficient configuration and high-throughput data movement between the accelerator and the surrounding system.
On the right side, the upper output signals consist of four single-bit *done* status flags, each indicating the completion of a major processing phase within the accelerator, plus one error bit signal. The lower output ports implement a 32-bit streaming DMA interface used to return processed data, including the updated matrix values (G_{out}), the current pivot map (*is_pivot*), and the updated pivot history (*was_pivot*).

3.2 Internal Modules

3.2.1 Control Unit

The Control Unit (CU) is in charge of the complete execution of the RREF algorithm by sequencing memory accesses, arithmetic operations, and pivot tracking through a centralized Finite State Machine (FSM). Rather than implementing a deeply pipelined dataflow architecture, the design adopts a *macro–micro control hierarchy*: high-level algorithmic phases are handled by the main FSM states, while fine-grained BRAM read–modify–write operations are managed through an internal micro-phase counter (`mem_phase`).

Execution Model

The CU operates on the matrix stored in the on-chip BRAM using a word-parallel access pattern (32-bit words, four 8-bit field elements per cycle). Each matrix manipulation such as row swapping, scaling, or elimination, is decomposed into a sequence of deterministic micro-operations:

- Issue BRAM read
- Wait one cycle for data availability (synchronous memory latency)
- Perform finite-field arithmetic
- Write the updated word back to memory

These sub-steps are scheduled using the `mem_phase` signal, which allows multi-cycle memory transactions without increasing FSM state complexity (described in more detail in later sections).

FSM Structure

The FSM directly reflects the algorithmic structure of Gaussian elimination with pivot reuse. As shown in Figure 3.2, to improve readability of the diagram, states are grouped into color-coded regions that correspond to the main algorithmic phases. The same grouping is reflected in the structured description below, so that the textual explanation can be directly mapped to the visual representation.
The FSM interacts with the datapath through explicit control signals, including memory read/write enables, row-swap triggers, scaling activation, reduction enable signals, and pivot tracking updates. Handshake signals such as `start_*`, `valid_*`, `read_*` and `done_*` coordinate execution with the host interface, while internal flags such as `is_pivot` and `was_pivot` guide pivot selection and reuse decisions.

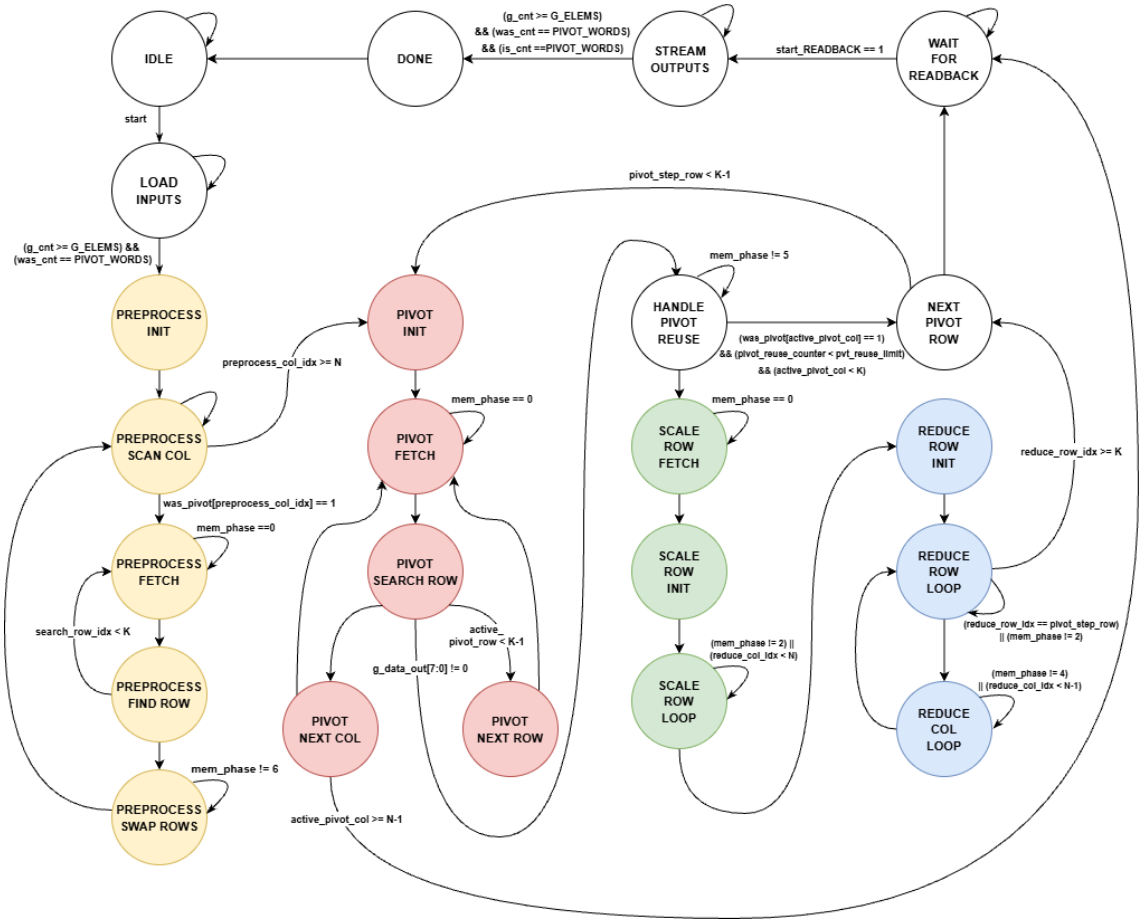

Figure 3.2: Finite State Machine (FSM) of the RREF accelerator control unit. The diagram shows state transitions across the main algorithmic phases: Preprocessing using historical pivots (yellow), Pivot discovery (red), Row normalization (green), and Column elimination (blue). Uncolored states correspond to general control operations. The default reset state of the accelerator is `IDLE`.

The states can be grouped into six functional regions:

1. **Input Handling (Uncolored — `IDLE`, `LOAD_INPUTS`)**

The accelerator resets into `IDLE`, where all counters and pivot maps are cleared. Upon assertion of the `start` signal, the FSM transitions to `LOAD_INPUTS` as shown in Figure 3.3.

During this phase, matrix elements are streamed into on-chip word-parallel memory through controlled byte-enable signals (`g_wstrb`). **Simultaneously**, the historical pivot map (`was_pivot`) is loaded and unpacked from 32-bit words into the internal bit vector representation. The input counters (`g_cnt`, `was_cnt`) ensure correct termination of the loading phase before transitioning to preprocessing.

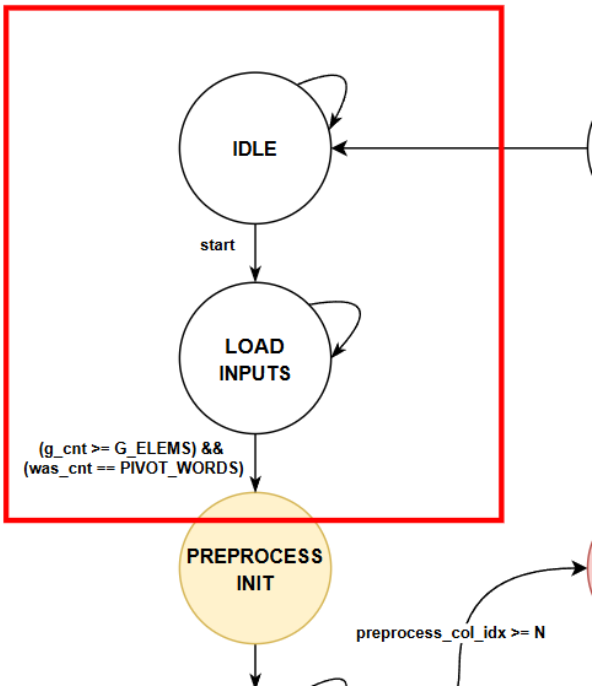

Figure 3.3: Zoomed view of the Input Handling region of the FSM (Figure 3.2). The transition from `IDLE` to `LOAD_INPUTS` occurs upon assertion of the `start` signal. The loading phase (download-to-accelerator) completes once both the matrix counter (`g_cnt`) and the historical pivot counter (`was_cnt`) reach their programmed limits.

2. **Preprocessing Using Historical Pivots (Yellow — `PREPROCESS_*`)**

   The preprocessing phase, as highlighted in Figure 3.4, begins immediately after the completion of `LOAD_INPUTS`. Once both `g_cnt` and `was_cnt` reach their programmed limits, the FSM transitions to `PREPROCESS_INIT`.

   In `PREPROCESS_INIT`, the internal column traversal register `preprocess_col_idx` is initialized to $K - 1$, where K denotes the number of matrix rows.

   This choice exploits a structural property of a $K \times N$ generator matrix in RREF form: at most K pivot positions can exist, and each pivot occupies a distinct row. Consequently, no more than K pivot columns are structurally meaningful.

Restricting the preprocessing scan to the range $[0, K-1]$ avoids unnecessary inspection of columns that cannot correspond to valid pivot restorations, while also providing a static and deterministic upper bound on the traversal length.

Traversing columns from high to low preserves the relative ordering of previously established pivots and prevents interference with lower-index pivot positions. No row search is performed in this state; it strictly prepares the column traversal logic.

The FSM then enters `PREPROCESS_SCAN_COL`, where the `was_pivot` vector is examined at the current `preprocess_col_idx`. Two cases are possible:

- If `was_pivot[preprocess_col_idx]` is deasserted, the column does not correspond to a historical pivot. The FSM decrements `preprocess_col_idx` and remains in `PREPROCESS_SCAN_COL`.
- If the bit is asserted, the column previously contained a pivot and must be structurally restored. In this case, the row search registers are initialized: `search_row_idx` and `active_pivot_row` are cleared (set to 0), and the FSM transitions to `PREPROCESS_FETCH`.

In `PREPROCESS_FETCH`, a memory read transaction is initiated to retrieve the matrix word corresponding to the element at column `preprocess_col_idx` and row `search_row_idx` (so it starts to search from the top of the column). Memory accesses follow the standard multi-cycle protocol controlled by `mem_phase`. Once the requested word is available, the FSM transitions to `PREPROCESS_FIND_ROW`.

In `PREPROCESS_FIND_ROW`, the element corresponding to `preprocess_col_idx` is extracted from the fetched 32-bit word and tested for a non-zero value. If the element is non-zero, the register `active_pivot_row` is updated with the current `search_row_idx`. Regardless of the comparison result, `search_row_idx` is incremented and the FSM returns to `PREPROCESS_FETCH` to continue scanning subsequent rows.

Importantly, the row search completes its full traversal until the upper row bound (K) before any structural decision is taken. Even if a non-zero element is found early, the scan proceeds deterministically until the termination condition on `search_row_idx` is reached ($< K$).

After the row scan completes, the FSM enters `PREPROCESS_SWAP_ROWS`, with `active_pivot_row` set to the last non-zero element found in the column (from top to bottom), provided that a candidate row was detected.

If `active_pivot_row` corresponds to the canonical pivot row position associated with the current column index, no swap is required and control returns to `PREPROCESS_SCAN_COL`. Otherwise, a row exchange is performed.

The swap operation is implemented as a multi-cycle read–buffer–write sequence controlled by `mem_phase`. Entire rows are exchanged word by word. This phase is accelerated because each memory transaction handles 4 elements at a time (1 word). The swap continues until the internal column counter reaches

the matrix width. Upon completion of the swap sequence, control returns to `PREPROCESS_SCAN_COL`, and `preprocess_col_idx` is decremented to continue the descending traversal.

The preprocessing phase terminates when the column traversal register exits the valid range (`preprocess_col_idx` < 0 which in unsigned terms can be checked by the out-of-bounds range $[N : +\infty)$ because of roll-over when decreasing under 0), indicating that all historical pivot columns have been examined. The FSM then transitions to the main pivot discovery stage (`PIVOT_INIT`).

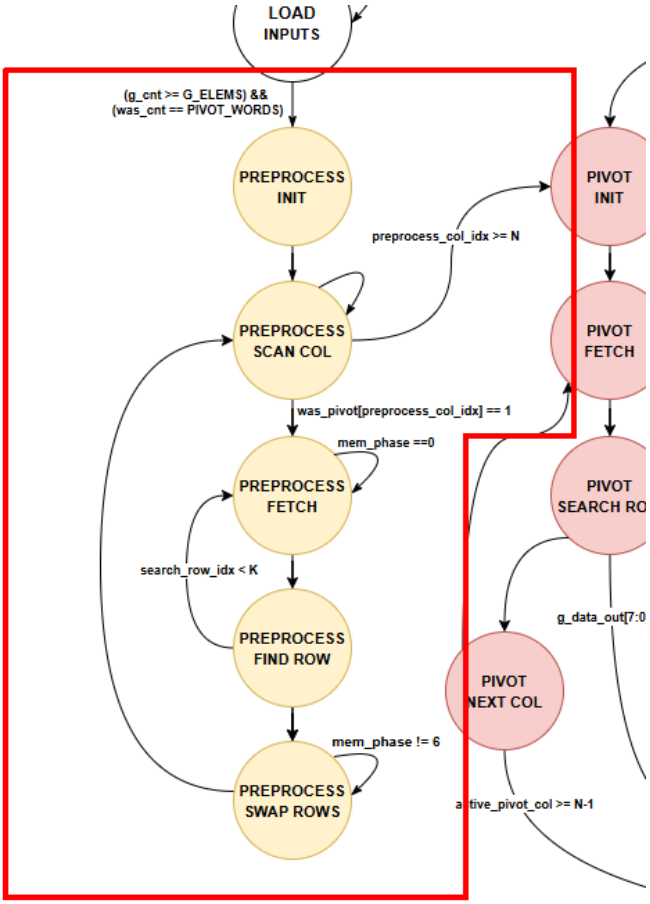

Figure 3.4: Zoomed view of the Preprocessing region of the FSM (Figure 3.2). After input loading, the FSM scans the `was_pivot` vector starting from the highest column index. For each marked column, potential row reordering is performed. The preprocessing phase terminates when the column counter (`preprocess_col_idx`) completes its full traversal of the matrix width.

3. **Pivot Discovery (Red — `PIVOT_*`, `HANDLE_PIVOT_REUSE`)**

   The pivot discovery phase, shown in Figure 3.5, begins immediately after preprocessing completes. When `preprocess_col_idx` exits its valid range, the FSM transitions to `PIVOT_INIT`.

   In `PIVOT_INIT`, the control registers `active_pivot_row` and `active_pivot_`

col are both initialized to pivot_step_row. This establishes the diagonal starting position for the current pivot search step.

The pivot search scans columns in an ascending order: starting from index 0 and advancing toward N-1. For each candidate column (active_pivot_col), the FSM searches downwards along the rows, starting from pivot_step_row and incrementing active_pivot_row. This guarantees that previously reduced rows (with index smaller than pivot_step_row) are not modified during subsequent elimination steps.

The FSM then enters PIVOT_FETCH, a read request is issued to the matrix memory at address IDX(active_pivot_row, active_pivot_col). In the following cycle, once the data is available, mem_phase is cleared and the FSM transitions to PIVOT_SEARCH_ROW.

In the PIVOT_SEARCH_ROW state, the CU checks the fetched element for a non-zero value.

There are two possible outcomes:

- If the element is zero, the FSM transitions to PIVOT_NEXT_ROW, where active_pivot_row is incremented. If additional rows remain (active_pivot_row $< K - 1$), control returns to PIVOT_FETCH. Otherwise, the column contains no valid pivot candidate, and the FSM transitions to PIVOT_NEXT_COL.
- If the element is non-zero, a valid pivot has been found and the FSM transitions to the next state HANDLE_PIVOT_REUSE.

In PIVOT_NEXT_COL, the column index active_pivot_col is incremented, provided that it has not reached $N-1$. The row index is reset to pivot_step_row, and the FSM returns to PIVOT_FETCH. If the column index reaches $N-1$ without finding a valid pivot, an error flag is asserted, the search terminates and it transitions to WAIT_FOR_READBACK.

As previously mentioned, when a pivot candidate is detected, the FSM moves to HANDLE_PIVOT_REUSE. In this state, the column is marked in the is_pivot vector. If the detected pivot row active_pivot_row differs from the canonical row pivot_step_row, a row swap is performed.

The swap operation follows a deterministic multi-cycle read–buffer–write sequence controlled by mem_phase in a similar way as the PREPROCESS_SWAP_ROWS phase previously described. First, a word from the canonical row is read and buffered. Then, the corresponding word from the detected pivot row is read. Subsequently, the buffered words are written back in reversed positions. This process repeats word by word until word_idx == ROW_WORDS. The design exchanges four field elements per memory word, preserving word-parallel efficiency and maintaining a uniform memory access pattern.

If no row swap is required the FSM skips directly to the finalization step.

The swap sequence terminates when `mem_phase == 5`. At this point, memory control signals are cleared, `word_idx` is reset, and the pivot row alignment is complete.

Pivot reuse is conditionally enabled to exploit previously identified pivot positions while preserving bounded control behavior. Reuse is activated only if these three conditions align:

$$\texttt{was\_pivot[active\_pivot\_col]} = 1,$$

$$\texttt{pivot\_reuse\_counter} < \texttt{pvt\_reuse\_limit},$$

$$\texttt{active\_pivot\_col} < \texttt{K}$$

The vector `was_pivot` stores historical pivot information. If the current column was previously marked as pivot, the design can reuse this structural knowledge instead of performing a full new pivot search. The register `pivot_reuse_counter` limits the number of consecutive reuse operations through the parameter `pvt_reuse_limit`, ensuring bounded and controlled behavior.

Upon completion of `HANDLE_PIVOT_REUSE` (i.e., when `mem_phase == 5`), row alignment is finalized and `word_idx` is reset. The FSM then transitions to `SCALE_ROW_FETCH`, marking the beginning of the row-scaling phase for the aligned pivot row.

The update of `pivot_step_row` is performed only after the scaling and elimination phases complete, when control reaches `NEXT_PIVOT_ROW`.

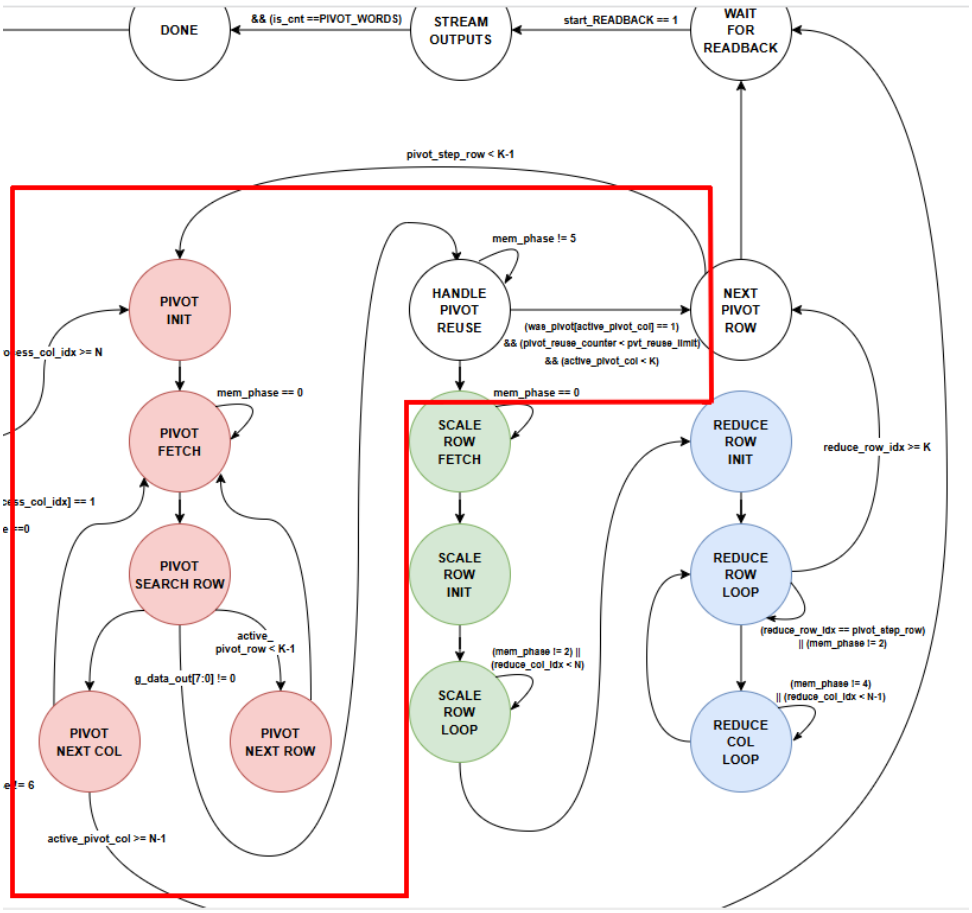

Figure 3.5: Zoomed view of the Pivot Discovery region of the FSM (Figure 3.2). Starting from the diagonal position defined by `pivot_step_row`, the FSM performs a column-wise search for a non-zero pivot candidate. If needed, row alignment (swap) and pivot reuse handling are performed.

4. **Row Normalization (Green — `SCALE_ROW_*`)**

   The row normalization phase, shown in Figure 3.6, begins immediately after a pivot has been aligned by `HANDLE_PIVOT_REUSE`. The FSM then transitions to `SCALE_ROW_FETCH` and subsequently to `SCALE_ROW_INIT`, marking the start of pivot scaling.

   In `SCALE_ROW_INIT`, the following occurs:

   - The pivot element at `IDX(pivot_step_row, active_pivot_col)` is read from memory.
   - Its multiplicative inverse in GF(127) is computed via `fq_inv()` and stored in `scaling_factor`.
   - The counter `reduce_col_idx` is initialized to `active_pivot_col` to traverse the pivot row.

   Once the scaling factor is available, the FSM transitions to `SCALE_ROW_LOOP`. In this state, the pivot row is processed sequentially, (four 8-bit elements at a time with a 32-bit memory word). Each loop iteration performs the following steps:

   (a) Issue a memory read at address `IDX(pivot_step_row,reduce_col_idx)`.

   (b) Once the word is available, multiply each 8-bit element by `scaling_factor` (thus processing 4 elements at a time).

   (c) Write the scaled word back to the same memory address, using `g_we` and `g_wstrb` to mask individual bytes (elements).

   The sequence is controlled by `mem_phase`, which advances from 0 to 3 for each read–compute–write operation:

   After processing a word, `reduce_col_idx` is incremented by 1 up to 4 depending on the remaining elements. The loop continues until all columns of the pivot row are scaled. At completion:

   - `mem_phase` is reset to 0.
   - `reduce_col_idx` is cleared.
   - The pivot element is guaranteed to be 1.

   The FSM then exits the row normalization phase and goes to `REDUCE_ROW_INIT`, starting the column elimination phase.

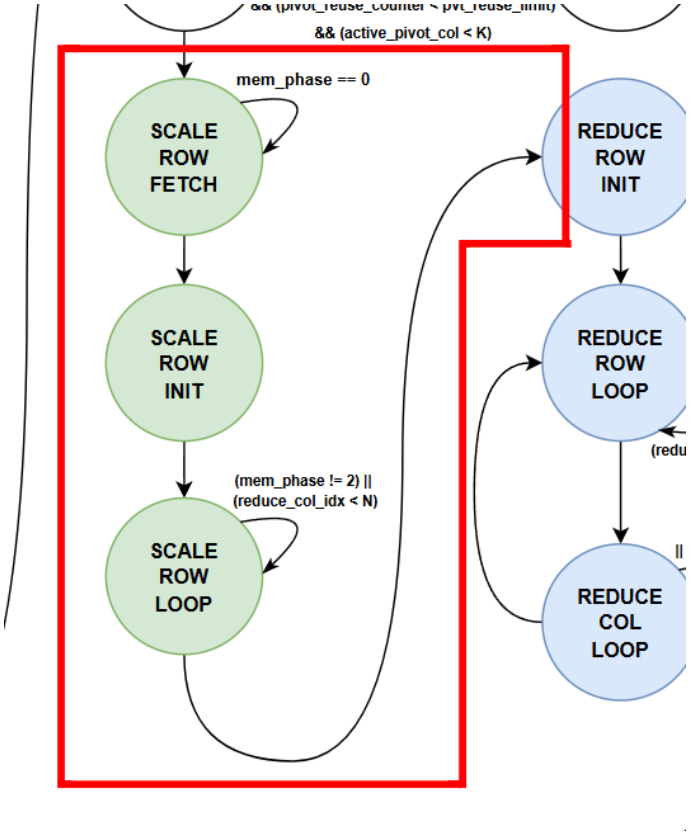

Figure 3.6: Zoomed view of the Row Normalization region of the FSM (Figure 3.2). The pivot element is first fetched (`SCALE_ROW_FETCH`), its multiplicative inverse is computed (`SCALE_ROW_INIT`), and then the entire pivot row is scaled in a word-parallel fashion (`SCALE_ROW_LOOP`) using the 32-bit datapath. Upon completion, control transitions to `REDUCE_ROW_INIT` to start the elimination phase.

5. **Column Elimination (Blue — `REDUCE_ROW_*`, `REDUCE_COL_*`, `NEXT_PIVOT_ROW`)**

   The column elimination phase begins immediately after the pivot row has been normalized in `SCALE_ROW_LOOP`. At this point the pivot element satisfies

   $$G[\texttt{pivot\_step\_row}][\texttt{active\_pivot\_col}] = 1,$$

   which allows the pivot row to be used as a reference for eliminating the pivot column entries in all other rows.

   Control enters `REDUCE_ROW_INIT`, which initializes `reduce_row_idx` to 0 and prepares iteration over all matrix rows.

   The FSM then transitions to `REDUCE_ROW_LOOP`. In this state, rows are processed sequentially to determine the elimination multiplier for each row. For every row:

   - If `reduce_row_idx == pivot_step_row`, the row corresponds to the pivot row and is skipped to avoid self-elimination.
   - Otherwise, the pivot column element `G[reduce_row_idx][active_pivot_col]` is read from memory and stored in `reduce_multiplier`. This value represents the coefficient used to cancel the pivot column entry of the current row.

   Because the pivot element has already been normalized to 1, the Gaussian elimination rule simplifies to

   $$Row_r = Row_r - G[r][\texttt{active\_pivot\_col}] \cdot Row_{\texttt{pivot}}$$

   where $r = \texttt{reduce\_row\_idx}$. The value `reduce_multiplier` therefore corresponds to the element of the current row in the pivot column that must be eliminated.

   When elimination is required, the FSM transitions to `REDUCE_COL_LOOP`. In this state the entire row is updated word by word until `reduce_col_idx` reaches the end of the matrix width.

   Each word update is executed through a deterministic multi-cycle memory micro-sequence controlled by `mem_phase`:

   (a) Read the current word of the target row `G[reduce_row_idx][reduce_col_idx]` and store it in the local buffer.

   (b) Read the corresponding word of the pivot row `G[pivot_step_row][reduce_col_idx]`.

   (c) Compute the updated word according to the elimination rule over GF(127):

   $$\texttt{row\_word} - (\texttt{reduce\_multiplier} \cdot \texttt{pivot\_row\_word})$$

   processing four field elements in parallel within the 32-bit word.

(d) Write the updated word back to memory.

After each write, `reduce_col_idx` is incremented and the process repeats until all columns of the row have been updated. Control then returns to `REDUCE_ROW_LOOP`, where `reduce_row_idx` is incremented.

If `reduce_row_idx < K`, elimination continues with the next row. Once all rows have been processed, the pivot column is fully cleared above and below the pivot position.

At this point the FSM transitions to `NEXT_PIVOT_ROW`, where `pivot_step_row` is incremented. If `pivot_step_row < K-1` ($<$ K-1 instead of $<$ K as the last row K-1 is already scaled at the end and doesn't have remaining rows below it to be reduced), control returns to `PIVOT_INIT` to begin pivot discovery for the next column. Otherwise, the matrix has reached its final reduced form and control moves to `WAIT_FOR_READBACK`, signaling completion of the RREF computation.

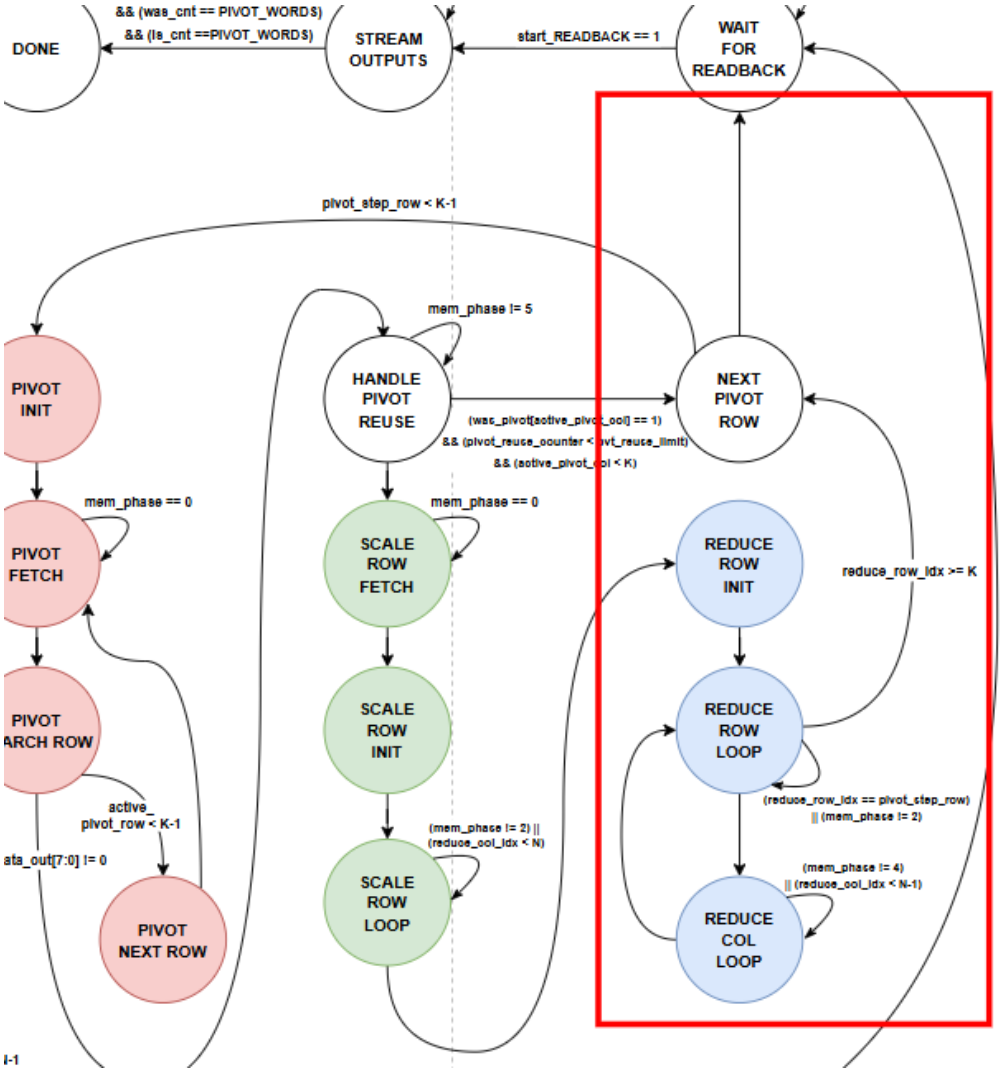

Figure 3.7: Zoomed view of the Column Elimination region of the FSM (Figure 3.2). The FSM iterates over all non-pivot rows to clear the current pivot column. Row-wise control is managed by `REDUCE_ROW_*`, while word-level elimination is performed in `REDUCE_COL_LOOP`, exploiting four-element word parallelism. After all rows have been processed, the FSM transitions to `NEXT_PIVOT_ROW`, which advances `pivot_step_row` and determines whether a new pivot discovery iteration should begin or the computation should terminate.

6. **Output Streaming (Uncolored — `WAIT_FOR_READBACK`, `STREAM_OUTPUTS`, `DONE`)**

   The output streaming phase begins once all pivot steps have been completed and the FSM exits the elimination stage. At this point the control logic transitions to `WAIT_FOR_READBACK`.

   In `WAIT_FOR_READBACK`, the accelerator asserts `done_COMPUTE` to signal that the RREF computation has finished and the resulting matrix is available in memory. The FSM remains in this state until the host asserts the `start_READBACK` signal, requesting output retrieval.

   Upon this request, the FSM transitions to `STREAM_OUTPUTS`. In this state, the accelerator sequentially reads data from the internal memory and streams the results back to the host interface when it's ready to receive them (using `read_G` for the matrix elements, and `read_WAS` / `read_IS` for the pivot tracking vectors).

   During this process:

   - matrix elements are read sequentially from memory and streamed as 32-bit words through DMA,
   - the pivot tracking vectors `was_pivot` and `is_pivot` are packed into 32-bit output words,
   - completion flags (`done_G_OUT`, `done_WAS_OUT`, and `done_IS_OUT`) indicate that each respective data stream has finished.

   Once all output data have been transmitted, the FSM ends on `DONE`. In this final state the accelerator completes the output transaction and prepares to return to `IDLE`, allowing a new computation to be started (in this state we still keep the `done_G_OUT` interrupt flag raised as it wouldn't wake the wfi() otherwise).

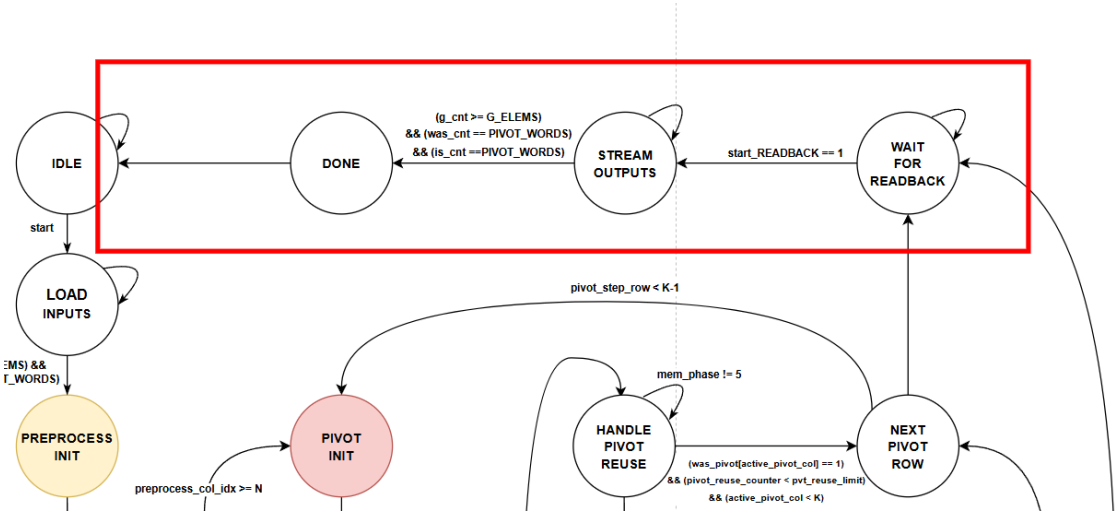

Figure 3.8: Zoomed view of the Output Streaming region of the FSM (Figure 3.2). After the RREF computation completes, the FSM enters `WAIT_FOR_READBACK` and asserts `done_COMPUTE`. Upon receiving `start_READBACK`, the FSM transitions to `STREAM_OUTPUTS`, where the reduced matrix and pivot maps (`was_pivot`, `is_pivot`) are sequentially read from memory and streamed to the host. Once all outputs have been transmitted, the FSM enters `DONE` before returning to `IDLE`.

Memory-Centric Operation

Because row operations dominate the workload, the CU is designed around BRAM bandwidth rather than arithmetic latency. All row manipulations (swap, scale and eliminate) are implemented as in-place memory transformations using temporary word buffers. This avoids the need for full-row register storage and allows the accelerator to support large matrices within FPGA memory constraints.

CU Operation and Timing Overview

The Control Unit (CU) coordinates the complete RREF computation by orchestrating input streaming, pivot reprocessing, pivot discovery, row scaling, column elimination, and output streaming. Figure 3.9 provides an example of a normal execution of the RREF accelerator, showing all major phases from `IDLE` to the `DONE` states. After the preprocessing macro-phase, the pivot-discovery, row-scaling, and column-elimination phases repeat iteratively for each pivot. Most of the execution time is occupied by input/output streaming and, in particular, by the row elimination phase, as iterating over every row for each pivot in a large matrix dominates the computation time.

Figure 3.9: Example timing diagram of a normal execution of the RREF computation on the accelerator, illustrating all major phases from input loading to output streaming. The timing is illustrative and does not reflect the exact lengths of internal states. Colors are consistent with previous figures.

The preprocessing phase, which includes column scans and optional row swaps before

pivot discovery, is illustrated in Figure 3.10. This phase ensures that the pivot search starts from a correctly aligned matrix state.

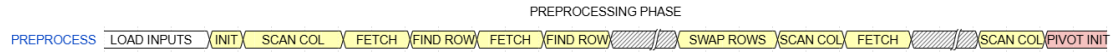

Figure 3.10: Example of a normal execution of the preprocessing phase, showing column scans and optional row swaps before pivot discovery. The timing is illustrative and internal state durations are not exact.

Pivot discovery, including row scanning and optional pivot reuse, is illustrated in Figure 3.11. Once a valid pivot is found, the row can be aligned and prepared for scaling.

Figure 3.11: Example timing the pivot discovery phase, highlighting row scanning and pivot selection. The timing is illustrative; internal state lengths are not exact.

The row scaling phase, shown in Figure 3.12, normalizes the pivot element and scales the entire pivot row in preparation for elimination. This phase allowed to read-modify and write four elements of the row simultaneously as each element on the same row is independent to one-another.

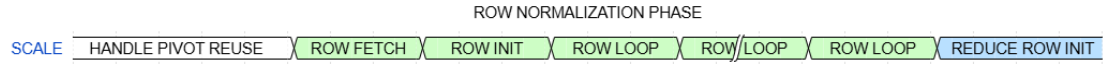

Figure 3.12: Example timing of the row scaling phase, showing the normalization of the pivot element and scaling of the entire pivot row using word-parallel operations. Timing is illustrative and internal state durations are not exact.

The row elimination phase, where pivot contributions are removed from other rows, is shown in Figure 3.13.

Figure 3.13: Example timing of the row elimination phase, showing how pivot contributions are removed from row elements. The timing is illustrative and internal state durations are not exact.

3.2.2 Pivot Registers

To reduce redundant computations across successive elimination phases, the accelerator integrates a dedicated *pivot tracking and reuse mechanism* implemented directly in hardware. Although the reuse strategy originates from the software reference implementation, its realization in the accelerator requires explicit architectural support and state management.
Pivot history is stored on-chip using two dedicated N-bit register vectors, where N corresponds to the number of matrix columns:

- `was_pivot` — pivot positions from previous executions (streamed to the accelerator at the start, and out at completion of the RREF reduction)
- `is_pivot` — pivot positions from current execution (only streamed out at completion)

Both vectors are implemented as synchronous register arrays, enabling constant-time read and write access within a single clock cycle. The vectors are cleared upon accelerator reset and updated at the end of each pivot discovery phase.
This register-based representation ensures minimal access latency and avoids additional BRAM utilization, since only one bit per column is required.

Implementation Details

Synchronous register arrays with single-cycle access:

- **Total Storage**: $2 \times N$ flip-flops (together, no BRAM)
- **Reset**: Both cleared to zero on accelerator reset
- **Bandwidth**: 32-bit parallel load/store matching streaming interface

CU Integration

The CU can reuse previously detected pivot columns if no better candidate is found. This behavior is controlled by the runtime parameter `pvt_reuse_limit`, which is applied directly in the CU logic. No additional algorithmic changes are introduced; the hardware simply enforces this behavior using the pivot vectors.

3.2.3 Generator Matrix Memory

The generator matrix G is stored in on-chip synchronous memory implemented using FPGA BRAM resources. Since the RREF algorithm is dominated by row-based read-modify-write operations, the memory subsystem is designed to provide sufficient bandwidth while minimizing LUT and register utilization.

The **G_MEM** module implements the storage for G and exposes a 32-bit word interface to the Control Unit. Internally, however, the memory is organized as four parallel 8-bit banks, matching the natural width of field elements over GF(127). This organization enables word-parallel processing of four matrix elements per cycle while preserving byte-level storage efficiency.

Figures 3.14 and 3.15 illustrate the internal architecture and the logical interleaving scheme adopted in the design. The interface follows a conventional memory signaling scheme including the write enable `g_we`, byte write strobes `g_wstrb`, byte address `g_addr`, write data `g_data_in`, and read data output `g_data_out`. The memory is dimensioned according to the generator matrix size, storing all K rows and N columns of G. The total number of stored elements is therefore:

$$G_{\text{ELEMS}} = K \times N.$$

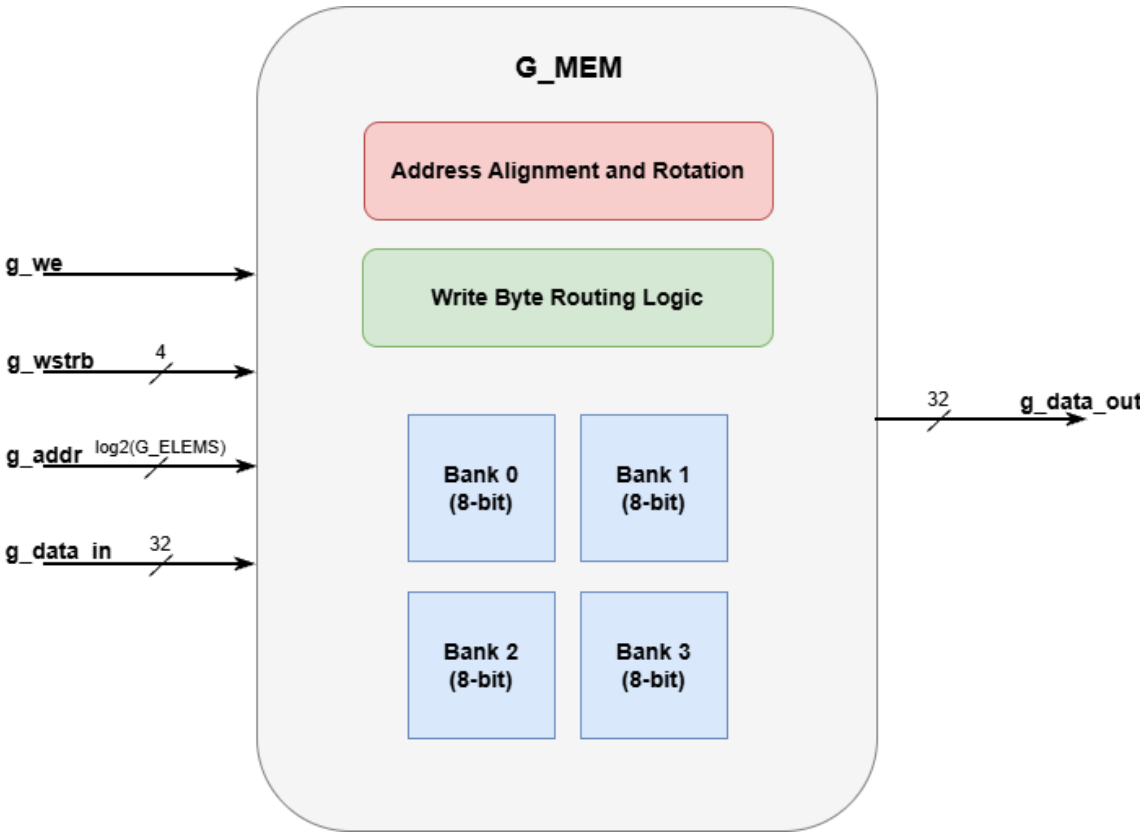

Figure 3.14: Internal organization of the G matrix synchronous memory. A 32-bit word interface is realized using four interleaved 8-bit BRAM banks. Address alignment and byte-rotation logic enable correct handling of unaligned accesses while preserving a word-parallel external interface. The memory depth is parameterized by the matrix size, with the total number of stored elements given by $G_{\mathrm{ELEMS}} = K \times N$. Clock and Reset signals have been omitted.

Design Rationale

A BRAM-based implementation was selected after early register-based prototypes exceeded LUT and flip-flop budgets even for moderate LESS parameter sets. The adopted banking scheme provides a balanced tradeoff between:

- byte-granular storage (one field element per byte),
- word-parallel access to match the 32-bit datapath,
- efficient BRAM inference during synthesis,
- support for unaligned accesses,
- deterministic single-cycle throughput.

Because Gaussian elimination is strongly memory-bound, prioritizing predictable throughput at the memory interface is essential for maintaining accelerator efficiency.

Logical Organization

Conceptually, the matrix G is stored in row-major order with one byte per field element. A logical 32-bit word therefore contains four consecutive matrix elements. The byte address corresponding to element (r, c) is:

$$\text{addr}_{\text{byte}} = r \cdot N + c.$$

This abstraction matches the software view of the matrix and simplifies Control Unit address generation. Internally, however, the data is reorganized to parallel BRAM accesses.

Banked Physical Structure

As shown in Figure 3.14, the memory is implemented using four independent BRAM banks, each storing one byte lane of the logical word. The banks are instantiated through the `g_mem_simple` module, a parameterized synchronous 8-bit memory block used as a basic building block that infers FPGA BRAM resources and exposes a simple address, write-enable, and data interface. Each instance has depth `G_WORDS`. Together they form a 32-bit word:

$$G_{\text{word}} = \{\text{bank}_3,\ \text{bank}_2,\ \text{bank}_1,\ \text{bank}_0\}.$$

This byte-interleaved organization allows the accelerator to read or write four field elements per cycle while preserving byte-level addressability. This scheme ensures that any aligned 32-bit access retrieves four consecutive field elements in a single cycle which was essential for the speedup of most sections.

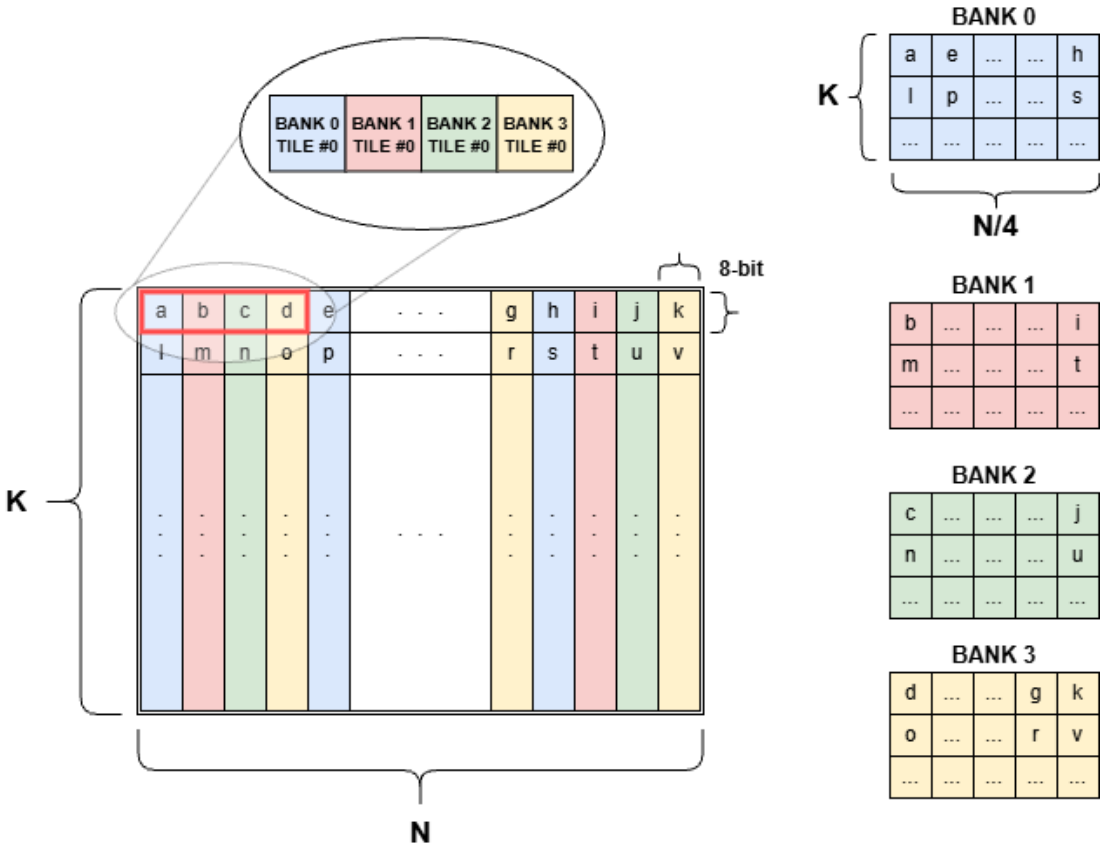

Figure 3.15: Interleaved organization of the synchronous memory storing matrix G. The external 32-bit word interface is implemented using four parallel 8-bit BRAM banks, each holding one quarter of the total memory. The memory stores exactly $G_{\text{ELEMS}} = K \times N$ field elements, distributed evenly across the banks. Each memory bank is represented with different colors: Bank 0(blue), Bank 1(red), Bank 2(green), Bank 3(yellow). Each square block represents a single 8-bit field element.

Address Decomposition

The external address `g_addr` is byte-based and is decomposed internally into:

- word index:

$$\texttt{idx} = \texttt{g\_addr} \gg 2$$

- byte offset within the word:

$$\texttt{g\_addr[1:0]}$$

The word index selects the BRAM row, while the lower bits determine byte alignment and bank rotation.

Support for Unaligned Accesses

Row operations in the RREF algorithm do not guarantee 32-bit alignment. To avoid performance penalties, G_MEM supports arbitrary byte-aligned accesses through a rotation network.
Because BRAM reads are synchronous, the lower address bits are registered:

```
addr_lo_r <= g_addr[1:0];
```

On the following cycle, the four bank outputs are rotated according to `addr_lo_r`. This effectively implements a barrel shifter that realigns the requested bytes without additional memory accesses. This functionality is implemented by the *Address Alignment and Rotation* block shown in Figure 3.14.
This mechanism allows:

- correct handling of unaligned reads,
- preservation of single-cycle throughput,
- software-transparent byte addressing.

When accesses cross a word boundary, selected bank addresses are incremented to fetch the correct byte from the next word. This ensures correctness without introducing extra cycles. The alignment logic is purely combinational and does not introduce additional memory latency.

Write Path

The write datapath mirrors the read flexibility using the byte-enable signal `g_wstrb[3:0]`. Depending on the byte offset, incoming bytes are steered to the appropriate bank and address. The routing of write bytes to the appropriate memory bank is handled by the *Write Byte Routing Logic* block depicted in Figure 3.14. This provides:

- byte-granular updates,

- correct unaligned writes,
- efficient support for row operations.

Notably, although the RREF algorithm logically performs read–modify–write updates on matrix elements, the memory architecture itself avoids internal read–modify–write sequences.

Throughput Characteristics

The G_MEM architecture sustains the following:

- one 32-bit read per cycle,
- one 32-bit write per cycle,
- full support for arbitrary byte alignment,
- no structural stalls due to memory alignment.

This bandwidth is critical for the dominant RREF operations, particularly row scaling and elimination, which repeatedly stream across matrix rows.

Timing and Latency Analysis

The memory subsystem is designed around the synchronous BRAM latency of one clock cycle. The read pipeline operates as follows:

1. **Cycle** t: address `g_addr` is presented and bank addresses are computed.
2. **Cycle** $t + 1$: BRAM outputs `q0,q1,q2` and `q3` become valid and the rotation network immediately produces the aligned word `g_data_out`.

Thus, the effective read latency is just:

$$L_{\text{read}} = 1 \text{ cycle}.$$

Importantly, once the pipeline is filled, the design achieves a sustained throughput of one word per cycle.
For writes, the datapath is purely combinational before the BRAM input registers, yielding also just:

$$L_{\text{write}} = 1 \text{ cycle}$$

from address assertion to memory update.
Because no internal read-modify-write cycles are required and no multi-cycle arbitration is present, the memory timing remains fully deterministic. This property is particularly valuable for the accelerator control unit, which relies on predictable memory latency when scheduling of the `mem_phase` micro-operations.

3.2.4 Arithmetic Unit

The arithmetic operations required by the RREF algorithm over GF(127) are implemented through a lightweight combinational Arithmetic Unit (AU). Rather than instantiating a single centralized ALU, the design adopts an *inlined functional approach*, where field operations are provided as synthesizable SystemVerilog functions inside the shared package `rref_accel_synth_pkg` and invoked directly by the Control Unit (CU).
To sustain the throughput of the memory subsystem, the architecture supports up to four field elements processed in parallel. This level of parallelism matches the maximum number of matrix elements fetched simultaneously from the scratchpad memory and represents the worst-case compute demand of the accelerator.

Parallel Arithmetic Structure

As illustrated in Figure 3.16, the AU implements the following finite-field primitives:

- Modular addition/subtraction modulo Q
- Modular field multiplication followed by modular reduction modulo Q
- Modular multiplicative inversion via LUT

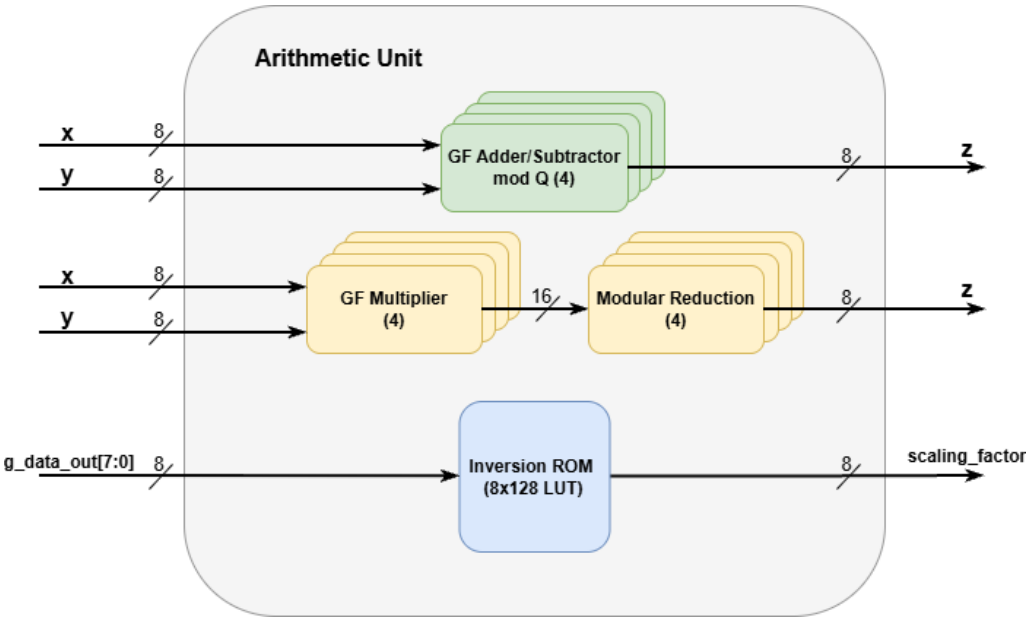

Figure 3.16: Finite-field arithmetic unit implementing operations over GF(127). The unit contains four parallel multipliers, each followed by a modular reduction stage, four adder/subtractor units for field addition and subtraction, and a lookup-table–based block for multiplicative inversion. For clarity, most arithmetic blocks are depicted with generic inputs x, y and output z, while detailed internal routing and concatenation logic are omitted. The inversion LUT interface, however, reflects the actual design signals.

It is important to note that the figure represents the *logical parallelism* of the design. Since arithmetic primitives are implemented as combinational functions, the synthesis tool may map multipliers either to LUT fabric or DSP blocks depending on resource availability, optimization settings and FanOut.
All operations target 8-bit storage elements, although only `NUM_BITS_Q` $= 7$ bits are mathematically significant. The rationale behind this representation choice and its hardware implications are discussed in the following paragraphs.

Field Representation

Each matrix element is represented as:

$$\texttt{FQ\_ELEM} \in [0, 126]$$

with arithmetic performed modulo

$$Q = 127 = 2^7 - 1.$$

The pseudo-Mersenne [30] structure of Q enables highly efficient modular reduction. Specifically, the special form $Q = 2^7 - 1$ allows any intermediate result to be reduced modulo Q using simple additions and bitwise operations, avoiding the need for costly division hardware. This property is especially advantageous in hardware implementations, where shifts and additions are significantly cheaper than normal division operations.
In the AU each `FQ_ELEM` is stored using 8 bits. However to avoid overflow during multiplication, intermediate results are computed using a wider 16-bit type, `FQ_DOUBLEPREC`(double of the FQ_ELEM size). This ensures that all products of two 8-bit field elements can be represented without loss of information before modular reduction.

Modular Reduction

Multiplication results are reduced using a folding technique based on:

$$x \bmod Q = (x_{\text{upper}} + x_{\text{lower}}) \bmod (2^7 - 1),$$

where:

$$x_{\text{upper}} = x \gg 7, \qquad x_{\text{lower}} = x \,\&\, Q = x \,\&\, 0b1111111.$$

The two terms are obtained by a right shift of 7 bits to extract the upper portion of the product and by masking the least significant 7 bits through a bitwise AND operation to obtain the lower portion.
This reduction is implemented by the function `fq_red()`, followed by a conditional subtraction to guarantee that the result is in the range $[0, Q - 1]$. This approach avoids iterative reduction and keeps the arithmetic fully combinational.

Multiplicative Inverse

The multiplicative inverse is computed through a 128-entry ROM (`fq_inv_table`). Given the small fixed modulus, the lookup approach provides constant latency and significantly lower hardware cost compared to an extended Euclidean implementation. The inverse is primarily used during the pivot normalization phase of the RREF algorithm.

Timing Characteristics

All arithmetic primitives are purely combinational, therefore:

- each arithmetic operation completes within a single clock cycle,
- no additional pipeline stages are required,
- the critical path is dominated by multiplier and reduction logic.

At the system level, the accelerator is predominantly *memory-bound*, since BRAM accesses already introduce a one-cycle latency that overlaps with arithmetic evaluation.

Integration with the Control Unit

The Arithmetic Unit operates under explicit control of the Control Unit (CU) and is activated only in the FSM states that require finite-field computation, which minimizes unnecessary switching activity. In practice, the highest arithmetic activity occurs during the `SCALE_ROW_*` and `REDUCE_ROW/COL_*` phases, where row normalization and Gaussian elimination generate sustained sequences of field multiplications and additions. Outside these phases, the accelerator behavior is largely dominated by memory movement and control flow, and only limited arithmetic activity is required. This usage pattern confirms that a lightweight combinational AU is sufficient to sustain the steady-state throughput of the overall architecture.

Pivot Normalization (`SCALE_ROW_*`) During the pivot row normalization phase, the AU is used in two steps:

1. **Pivot inversion** — When the pivot element is detected, the CU invokes `fq_inv(pivot element)` to compute

$$\texttt{scaling\_factor} = \texttt{(pivot element)}^{-1} \bmod Q.$$

2. **Row scaling** — Each element of the pivot row (`row element[i]`) is multiplied by the broadcast inverse:

$$\texttt{row\_scaled[i]} = \texttt{fq\_mul(scaling\_factor, row element[i])}.$$

In this phase, the AU primarily exercises:

- the inversion LUT (once per pivot),
- the parallel multipliers (streaming across the row).

The four-lane structure allows the CU to scale up to four pivot-row elements per memory word.

Column Elimination (`REDUCE_ROW_*`, `REDUCE_COL_*`) During elimination, the AU performs the classic Gaussian reduction shown here:

$$G[r, c] \leftarrow G[r, c] - \lambda_r \cdot G[p, c],$$

where:

- $G \in \mathrm{GF}(127)^{K \times N}$ is the generator matrix stored in `G_MEM`,
- r is the index of the row currently being reduced,
- p is the pivot-row index,
- c is the column index within the streamed word,
- $\lambda_r = G[r, p]$ is the elimination factor associated with row r.

For each element inside every streamed word, the CU performs:

```
        prod[i] = fq_mul(reduce_multiplier, pivot row element[i])
row element[i] = fq_sub(row element[i], prod[i]).
```

This phase uses:

- all multiplier lanes,
- all adder/subtractor lanes,
- the modular reduction logic.

Dataflow and Operand Sources

Across the FSM, the AU operands originate from three primary sources:

- **G_MEM read data** — streamed matrix elements,
- **temporary row buffers** — pivot and target rows,
- **scalars** — outputs of others AU operations, `reduce_multiplier`, `scaling_factor`.

Deterministic Scheduling

Because the AU is purely combinational, its latency is fully absorbed within the fixed `mem_phase` schedule:

- **Cycle** t: operands fetched from BRAM
- **Cycle** $t + 1$: AU computes and result is written back

No variable-latency operations are present, and no back-pressure from the AU to the CU is required.

3.2.5 Row Operation Registers

The *Row Operation Registers* constitute the datapath state backbone for row swapping, row normalization, and column elimination. These registers orchestrate data movement between the CU, the on-chip matrix memory (`g_mem`), and the AU.
From a control perspective, these registers are primarily active in the following CU states:

- `PREPROCESS_SWAP_ROWS`
- `HANDLE_PIVOT_REUSE`
- `SCALE_ROW_FETCH`, `SCALE_ROW_INIT`, `SCALE_ROW_LOOP`
- `REDUCE_ROW_INIT`, `REDUCE_ROW_LOOP`
- `REDUCE_COL_LOOP`

They remain idle during input streaming, pivot scanning, and output streaming. This confirms that arithmetic activity is localized to the Gaussian elimination phases, whereas the rest of the architecture is memory-driven, thus reducing switching activity.

Register Classification

The row operation registers can be grouped into four functional categories.

1) Row Buffers (Word-Level Temporary Storage)

- `row_buffer_a[4]`
- `row_buffer_b[4]`

Type: Small register arrays ($4 \times$ 8-bit field elements).
Purpose: Temporary storage of one 32-bit memory word from two different rows. Since the matrix is stored as 32-bit words (4 field elements per word), row swaps and elimination operate word-by-word. During swap operations:

- `row_buffer_a` stores the first row word,

- `row_buffer_b` stores the second row word,
- data is then written back in reversed order.

They are used in: `PREPROCESS_SWAP_ROWS` and `HANDLE_PIVOT_REUSE`.

2) Scaling and Elimination Registers

- **`scaling_factor`**
  Type: 8-bit GF element.
  Loaded in: `SCALE_ROW_INIT`.
  Purpose: Stores the multiplicative inverse of the pivot element, it remains constant throughout the entire `SCALE_ROW_LOOP` and is applied to each word of the pivot row.

- **`reduce_multiplier`**
  Type: 8-bit GF element.
  Loaded in: `REDUCE_ROW_LOOP` (at `mem_phase = 2`).
  Purpose: Stores $G[r]$[pivot_col], it's used to eliminate the pivot column entry in a non-pivot row and it remains constant during the entire `REDUCE_COL_LOOP` for the corresponding row.

- **`reduce_row_value_buffer`**
  Type: 32-bit register (stores 4 8-bit elements).
  Loaded in: `REDUCE_COL_LOOP` (at `mem_phase = 2`).
  Purpose: Temporarily stores the word read from the row currently being reduced.

  Because the design performs

  $$A_{r,c} \leftarrow A_{r,c} - \lambda \cdot A_{p,c},$$

  and memory is single-ported and synchronous, values must be buffered before the second operand arrives.

3) Index and Address Registers These registers steer memory addressing and define the iteration space:

- `pivot_step_row`
- `active_pivot_row`
- `active_pivot_col`
- `reduce_row_idx`
- `reduce_col_idx`
- `g_addr`

They:

- define which matrix row/column is being processed,
- convert 2D coordinates into linear addresses,
- are updated exclusively by the CU FSM.

They are control registers rather than arithmetic storage.

Micro-Phased Memory Execution (`mem_phase`)

A key architectural feature is the 3-bit register:

$$\texttt{mem\_phase} \in \{0, \ldots, 7\}.$$

As `g_mem` is synchronous:

- 1 cycle is required to issue an address,
- 1 cycle is required for data to become valid.

Instead of expanding the FSM with additional states, the design embeds micro-operations inside each macro-state using `mem_phase`.
This creates a two-level control hierarchy:

- **Macro-control:** FSM state (algorithm phase)
- **Micro-control:** `mem_phase` (cycle-accurate memory sequencing)

Detailed Timing per CU State

A) Row Swap Executed in `PREPROCESS_SWAP_ROWS` and `HANDLE_PIVOT_REUSE`. Up to four 8-bit elements are processed simultaneously through the mem_phases from 0 to 5, continuing in this manner until the last column is reached, as shown in Table 3.1 and Figure 3.17.

Table 3.1: Micro-operation management for row swapping used in CU states: `PREPROCESS_SWAP_ROWS` and `HANDLE_PIVOT_REUSE`.

mem_phase	Operation
0	Issue read for row A word
1	Issue read for row B word
2	Capture row A word
3	Capture row B word + issue write of A into B
4	Write word A into B + issue write of B into A
5	Write word B into A + increment `word_idx`
6	Swap complete

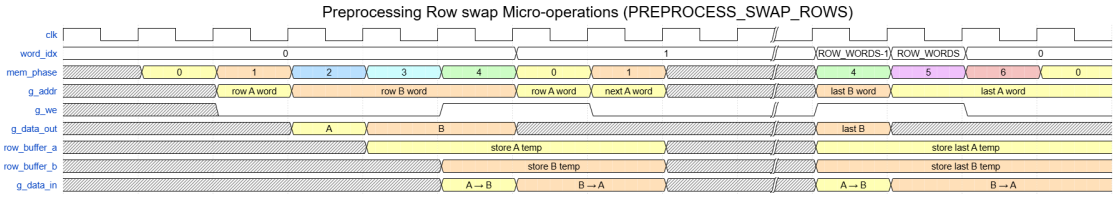

Figure 3.17: Timing diagram of the row-swap procedure executed in `PREPROCESS_SWAP_ROWS`. The control unit schedules memory accesses through the `mem_phase` micro-sequencer to account for the synchronous read latency of the generator matrix memory. Words from row A and row B are first read and stored in temporary buffers, after which the values are written back to the opposite rows, effectively completing the swap.

No arithmetic is performed; execution is entirely memory-bound.

B) Row Normalization Executed in `SCALE_ROW_LOOP`.
Up to four 8-bit elements are processed per iteration, as depicted in Table 3.2 and Figure 3.18.

Table 3.2: Micro-operation management for row normalization in `SCALE_ROW_LOOP`.

mem_phase	**Operation**
0	Issue read of pivot row word
1	Capture data (NOOP)
2	Multiply by `scaling_factor` and write back

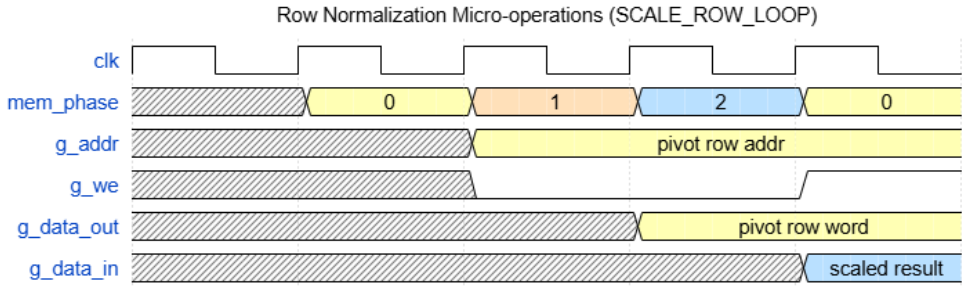

Figure 3.18: Timing diagram of the row normalization procedure executed in `SCALE_ROW_LOOP`. The control unit schedules memory accesses through the `mem_phase` micro-sequencer to account for the synchronous latency of the generator matrix memory. Each iteration reads a word from the pivot row, waits one cycle for data availability, and then multiplies the elements by the precomputed `scaling_factor` (pivot inverse) in GF(127) before writing the normalized values back to memory.

Therefore, each word requires 3 cycles to be updated. Arithmetic is confined to phase 2, whereas phases 0 and 1 are purely memory latency.

C) Row Elimination Executed in `REDUCE_ROW_LOOP` and subsequently in `REDUCE_COL_LOOP`.
The first step for row elimination consists in reading the `reduce_multiplier` value in `REDUCE_ROW_LOOP` as described in Table 3.3 and Figure 3.19.

Table 3.3: Micro-operation management for loading the elimination multiplier in `REDUCE_ROW_LOOP`.

mem_phase	Operation
0	Read pivot-column element
1	Wait (NOOP)
2	Capture `reduce_multiplier`

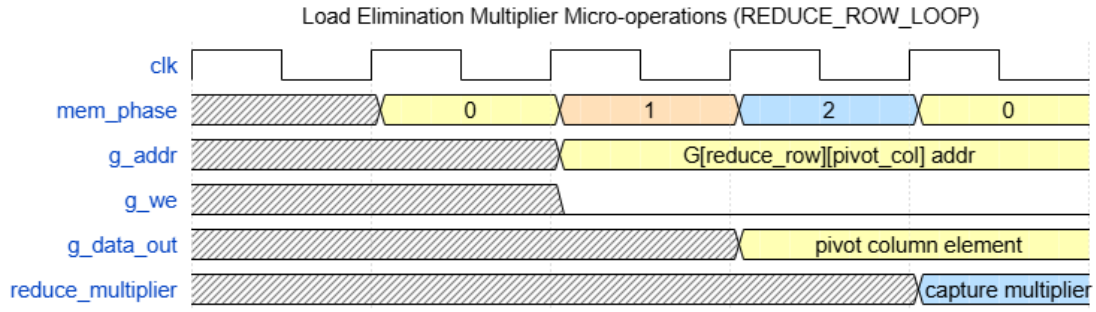

Figure 3.19: Timing diagram for loading the elimination multiplier during the `REDUCE_ROW_LOOP` phase. The control unit schedules the memory access through the `mem_phase` micro-sequencer to account for the synchronous read latency of the generator matrix memory. The element in the pivot column of the current row is first read, followed by a wait cycle, after which the value is captured and stored in the `reduce_multiplier` register. This value is later used by the arithmetic unit during the `REDUCE_COL_LOOP` phase.

In this step only a single 8-bit value is extracted from the memory word output. After obtaining the correct multiplicand value, the next phase, `REDUCE_COL_LOOP` performs the actual row reduction across all columns, as described in Table 3.4.

Table 3.4: Micro-operations for column elimination in `REDUCE_COL_LOOP`.

mem_phase	Operation
0	Issue read of current-row word
1	Issue read of reference-row word
2	Capture current-row word
3	Capture reference-row word, perform elimination, and write back
4	Advance column index and prepare next iteration

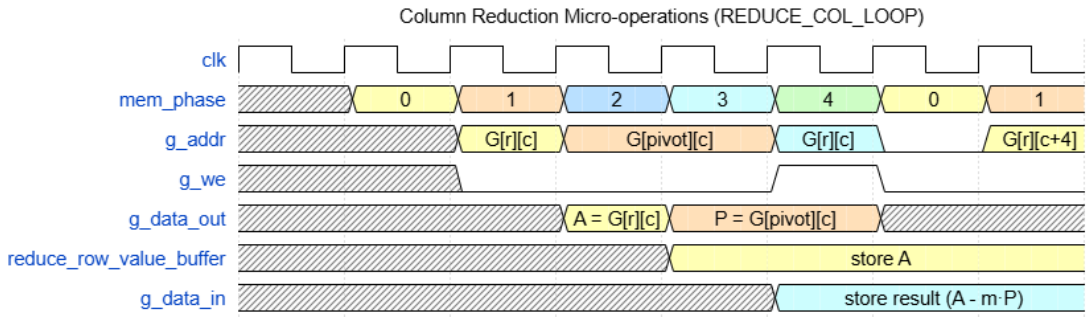

Figure 3.20: Timing diagram for the row elimination operation during the `REDUCE_COL_LOOP` phase. After the elimination multiplier has been loaded in the preceding `REDUCE_ROW_LOOP` phase, the arithmetic unit iteratively processes the elements of the row. For each column element, the pivot-row value is multiplied by the stored `reduce_multiplier`, and the result is added to the current row element to eliminate the pivot-column contribution. The control unit orchestrates the memory accesses and arithmetic operations through the `mem_phase` micro-sequencer, ensuring correct alignment with the synchronous memory read latency while streaming the row elements.

This sequence is the most memory-intensive part of the accelerator: two reads and one write per word, with the AU active only in phase 3.
These phases are executed across all columns of the target row and iterated for each non-pivot row until the elimination process completes.

4 X-HEEP Integration

This chapter describes the integration of the proposed Reduced Row Echelon Form (RREF) hardware accelerator within the eXtendable Heterogeneous Energy-Efficient Platform (X-HEEP) System-On-Chip (SoC) platform. The goal of this integration is to enable the execution of the LESS cryptographic protocol by offloading the computationally intensive RREF operation to a dedicated hardware accelerator, while the control flow remains managed by software running on an embedded processor.
Integrating the accelerator into the platform requires defining the control interface, connecting the hardware module to the system interconnect, and providing the necessary software support for its configuration and execution. The following sections introduce the X-HEEP platform and describe how the proposed accelerator is connected to the system.

4.1 X-HEEP

The hardware platform used in this work is based on X-HEEP, a configurable microcontroller-class SoC designed for embedded systems research and rapid hardware prototyping. The platform is built around a 32-bit RISC-V processor and provides a compact but flexible architecture that allows designers to experiment with different system configurations and hardware accelerators.
X-HEEP was developed with the objective of simplifying the exploration of heterogeneous embedded architectures. Its modular structure makes it possible to adapt the system configuration to the needs of a specific application by selecting different processor cores, adjusting the memory organization, or integrating custom hardware components[20].
In this project, X-HEEP is used through an internal development repository referred to as **CV-X-HEEP**. This repository is not an official version of the X-HEEP platform, but rather a project-specific environment maintained within the research group. It contains a local copy of the X-HEEP source tree together with additional configuration files, integration scripts, and peripheral definitions used to simplify the development and integration of experimental hardware modules.
The version of X-HEEP used in this work corresponds to the commit `a39bc88233d8a7c1baab05bbd703108eab5b7218` (26 October 2024), which was the revision vendored inside the CV-X-HEEP repository at the time this project started. Updating the underlying X-HEEP version would have required modifications to this external infrastructure, potentially affecting the compatibility of existing integration

components. Since the primary focus of this thesis is the design and evaluation of the RREF accelerator rather than modifications to the SoC framework itself, the vendored version of X-HEEP was kept unchanged.
Figure 4.1 presents a simplified view of the platform architecture. The system is organized around a central RISC-V processor that communicates with memory and peripherals through a shared system interconnect. Program execution is managed by the processor, while peripherals and external hardware units are accessed through memory-mapped interfaces.
The platform includes several main architectural elements. The processor subsystem provides the execution engine for the software stack and is based on cores from the CORE-V family. The memory subsystem contains on-chip memories used for program instructions and data storage. Communication between the different system components is handled by an interconnect that allows both processors and peripherals to exchange data using standardized bus interfaces. In addition to the core processing elements, the platform includes a set of system peripherals such as timers, interrupt controllers, and input/output interfaces that support typical embedded workloads.

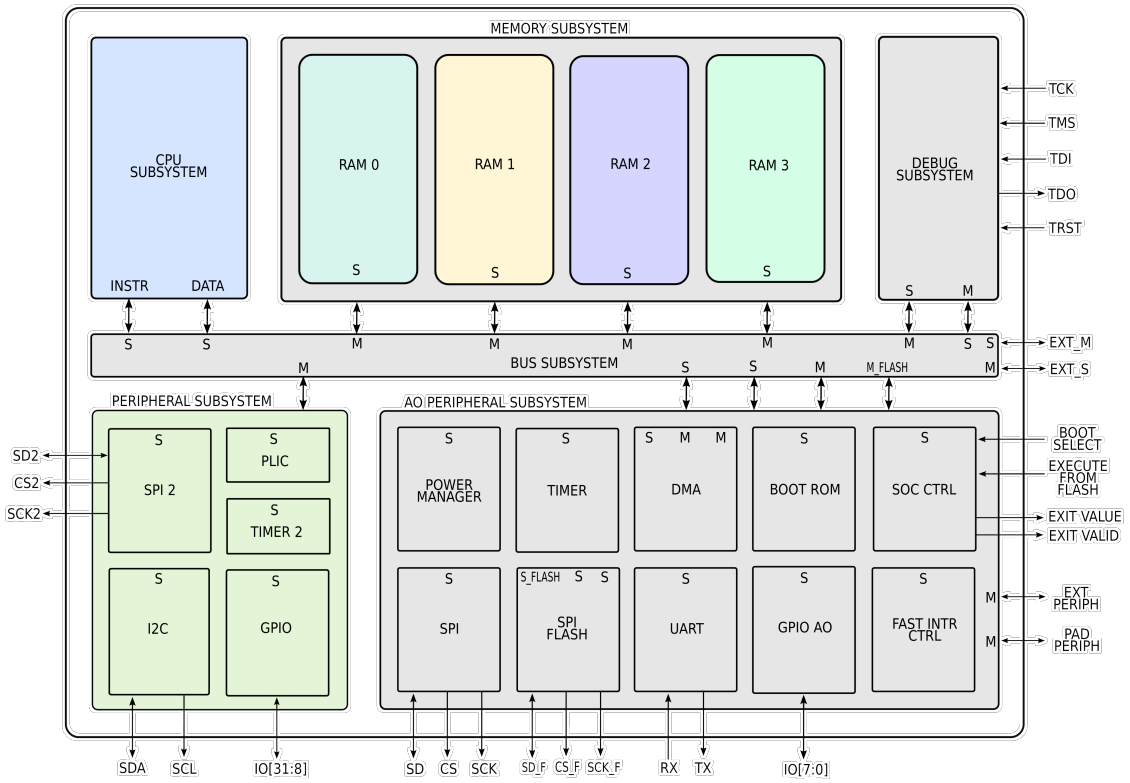

Figure 4.1: Architectural overview of the X-HEEP MCU[20].

One of the main advantages of the X-HEEP architecture is the ability to extend the system with custom hardware modules. External accelerators can be connected to the SoC through the peripheral subsystem and exposed to the processor as memory-mapped devices. This approach allows software to configure and control the accelerator using standard load and store instructions, while the hardware accelerator

itself performs the computationally intensive tasks independently in an optimized way.
In this work, the RREF accelerator is integrated using a *loosely coupled* approach. The accelerator appears as an external peripheral connected to the CV-X-HEEP subsystem. A set of control and status registers allows the processor to start and monitor accelerator operations. Large data transfers between memory and the accelerator are handled through the system DMA controller in order to reduce processor overhead and some interrupt signals to reduce polling for status registers. This organization allows the accelerator to execute independently from the processor pipeline while still remaining fully accessible through the system memory map, enabling efficient hardware/software cooperation during the execution of the LESS protocol.

4.2 RREF Integration

The integration of the RREF accelerator into the X-HEEP system follows several design steps.
Overall it can be summarized as follows:

- definition of the accelerator **register file**,
- creation of a hardware **wrapper** compatible with the system bus,
- integration into the **X-HEEP** system configuration,
- inclusion of the peripheral in the **FuseSoC** build flow,
- development of a software **driver** for processor interaction.
- functional **testing** through simulation and system-level verification.

Each of these steps is described in detail in the following subsections.

4.2.1 Step 1: Register File

The first step in the integration process consists in defining the register interface through which the processor communicates with the accelerator. In the X-HEEP platform, peripherals are exposed to the processor through **memory-mapped registers** that can be accessed using standard load and store instructions. In order to remain consistent with the platform infrastructure, all registers adopt a fixed width of 32 bits.
The register file defines the configuration and control interface of the RREF accelerator. Through these registers the processor can provide the required parameters, trigger the computation, and monitor the execution status of the hardware unit.
In the proposed design, the register interface is divided into two complementary components to efficiently manage control, status, and data streaming for the RREF accelerator:

- **`rref_accel.hjson`** – this file (Appendix A.2) defines the main register interface used for controlling the accelerator, providing input pivot vectors, and monitoring computation progress. It includes a small set of **control registers** (to start operations and enable result readback), **status registers** (to report errors and the completion of different phases), and **data/parameter registers** (to transfer pivot vectors and configure algorithm parameters). The most relevant registers of this interface are summarized in Table 4.1.
- **`rref_accel_data.hjson`** – this file (Appendix A.3) defines the interface for streaming the input G matrix. This interface, combined with the system DMA, allows the matrix to be streamed efficiently directly between memory and the accelerator, avoiding the performance bottleneck of register-based transfers as will be explained explained later.

Register	Bits	Description
`WAS_PIVOT`	31:0	Historical pivot column (32 one-bit elements).
`IS_PIVOT`	31:0	Output pivot column (32 one-bit elements) produced by the accelerator.
`CTRL`	1:0	Control register. **START** (bit 0) begins computation and input streaming; **START_READBACK** (bit 1) enables result readback streaming.
`STATUS`	4:0	Status flags: **ERROR** (0), **COMPUTE_DONE** (1), **G_OUT_DONE** (2), **WAS_OUT_DONE** (3), **IS_OUT_DONE** (4).
`PIVOT_REUSE_LIMIT`	31:0	Maximum number of pivot reuse operations allowed during the RREF computation.

Table 4.1: Register interface described in `rref_accel.hjson`.

It is worth noting that the `STATUS` register in Table 4.1 exposes five status flags. In the final implementation, only the `ERROR` flag is accessed directly by software through the register interface. The remaining signals (`COMPUTE_DONE`, `G_OUT_DONE`, `WAS_OUT_DONE`, and `IS_OUT_DONE`) are instead used to generate interrupts that notify the processor when different phases of the computation complete, as this proved to be a more efficient and faster method compared to continuous status polling. The corresponding status bits are nevertheless preserved in the register map to maintain compatibility with polling-based implementations.
The register interface is generated using the OpenTitan[31] register generation tool (`regtool`). This tool allows the register map to be described in a high-level `hjson` specification from which the corresponding SystemVerilog register file and a C header file containing the register offsets are automatically generated. The generated hardware module exposes a standardized interface connecting the register file to the accelerator logic through two structures, `reg2hw` and `hw2reg`, which represent the signals flowing from the registers to the hardware and vice versa.
During the early design phase it became evident that transferring the entire G matrix through standard register accesses would be inefficient. The matrix contains $K \times N$ elements, each encoded using 8 bits, which would require a large number of 32-bit

register transactions. Experimental measurements showed that the time spent by the CPU performing these register accesses was significantly larger than the actual computation time of the hardware accelerator.
To address this limitation, the final architecture adopts a **hybrid communication model**. Configuration parameters and control signals are exchanged through the register interface, while large data structures such as the G matrix are transferred using the system Direct Memory Access (DMA) controller. This approach allows data to be streamed directly between memory and the accelerator with minimal CPU intervention.

4.2.2 Step 2: Wrapper

After defining the register interface, the next step in the integration process consists in creating a **wrapper module** that connects the generated register files to the RREF accelerator core.
In the proposed design, this functionality is implemented in the `less_top.sv` module. The wrapper acts as the top-level hardware component of the accelerator and is responsible for connecting three main elements: the register files generated by `regtool`, the DMA-based streaming interface used for transferring the G matrix, and the RREF accelerator core (`rref_accel_synth`) as depicted in Figure 4.2.

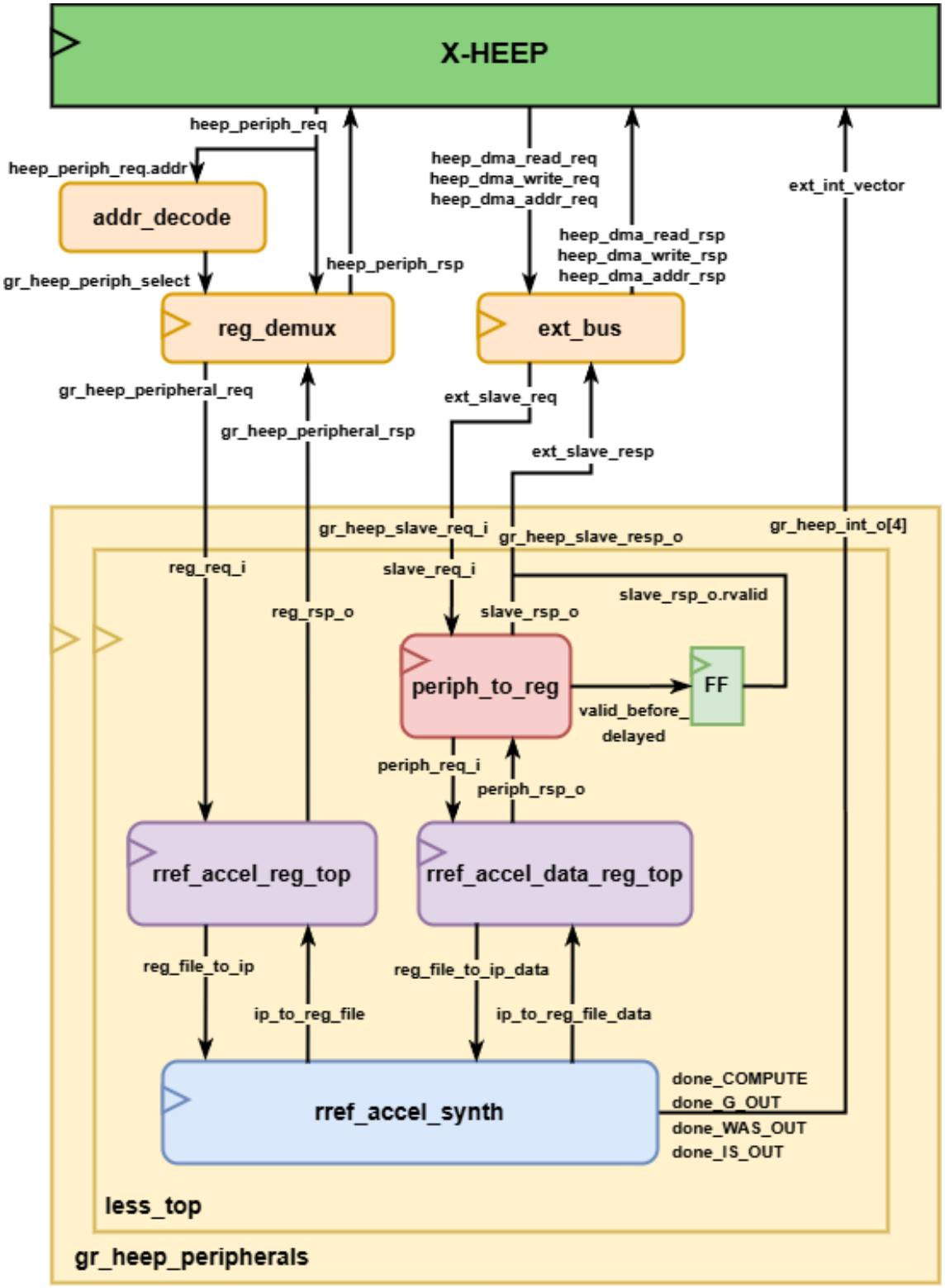

Figure 4.2: Interconnection diagram between RREF module and X-HEEP. Clock, Reset and less important signals have been omitted for clarity reasons.

The wrapper instantiates two register files generated from the previously described `hjson` specifications:

- **`rref_accel_reg_top`** – generated from `rref_accel.hjson`, this module exposes the configuration (was, is_pivot and pivot_reuse_limit), control, and status registers.
- **`rref_accel_data_reg_top`** – generated from `rref_accel_data.hjson`, this module implements the register interface used for streaming the elements of the G matrix.

The `reg2hw` and `hw2reg` structures contain several fields that encode both the register values and the access events:

- **`.q`** – the current value stored in the register (after synchronization with the clock).
- **`.d`** – the value that the hardware writes back to the register file.
- **`.qe`** – write enable signal generated when software performs a register write.
- **`.re`** – read enable signal asserted when software reads the register.
- **`.de`** – data enable signal used by hardware when updating register fields (forced to `1` for convenience).

The wrapper also integrates a `periph_to_reg` adapter that bridges the system OBI peripheral bus with the register-based interface expected by the data register file. This module converts memory transactions (`slave_req_i` and `slave_rsp_o`) originating from the DMA controller (passing through the `ext_bus`) into register read/write requests (`periph_req_i` and `periph_rsp_o`, allowing the matrix data to be written directly into the accelerator input stream.
The signal `slave_req_i.req` represents the request valid signal of the OBI slave interface. It indicates that a transaction is being issued by the DMA controller and is therefore used to drive the validity of the incoming data stream to the accelerator. Furthermore, to comply with the timing requirements of the OBI interface, the response valid signal (`slave_rsp_o.rvalid`) is delayed by one clock cycle inside the wrapper. This is implemented using a small flip-flop that captures the internal valid signal and forwards a delayed version to the bus response interface. This delay ensures that the read data and the corresponding valid signal remain properly aligned, avoiding timing mismatches between the peripheral interface and the register file.
In addition to the data path connections, the wrapper manages the control and status signals exchanged with the accelerator. Control signals such as `START` and `START_READBACK` are forwarded from the register file to the accelerator core, while status signals including `COMPUTE_DONE`, `G_OUT_DONE`, `WAS_OUT_DONE`, and `IS_OUT_DONE` are propagated back to the register interface. These signals are also exposed as interrupt outputs so that the processor can be notified when different phases of the computation complete.

Finally, the wrapper instantiates the accelerator core `rref_accel_synth`. The module receives control signals and pivot configuration parameters from the register interface, while the matrix data is streamed through the DMA-enabled interface. The results of the computation, including the updated G matrix and pivot vectors, are written back to the register file structures and made available to software through the same interface.

4.2.3 Step 3: Integration in the X-HEEP System

After the wrapper was implemented inside `less_top.sv`, it needed to be integrated into the X-HEEP platform as an external peripheral. This integration required modifications to two main modules of the system: `gr_heep_peripherals.sv` and `cv_x_heep_top.sv`. In addition, the accelerator address space had to be mapped within the system memory map by updating the platform configuration files (`gr_heep_pkg.sv` for the hardware part and `gr_heep.h` for the software).

Integration in `gr_heep_peripherals.sv` The `gr_heep_peripherals` module acts as a container for all external peripherals connected to the X-HEEP system. The RREF accelerator is instantiated inside this module through the `less_top` wrapper as we can see from Figure 4.2.
The wrapper exposes three main interfaces:

- An **OBI slave interface** connected to `gr_heep_slave_req_i` and `gr_heep_slave_resp_o`. This interface is used by the system DMA to stream the input matrix G directly to the accelerator. In the external bus architecture (`ext_bus`), the DMA acts as a bus master while the accelerator behaves purely as a slave device. The accelerator slave interface does not initiate bus transactions by itself. Instead, it only responds to requests generated by the DMA engine, which is configured by the software. The software therefore controls when the matrix G must be transferred to or from the accelerator, while the accelerator simply receives or returns the data when requested.

- A **register interface** connected to `gr_heep_peripheral_req` and `gr_heep_peripheral_rsp`. This interface exposes the control and configuration registers generated from the `hjson` specification and allows the processor to configure the accelerator through memory-mapped transactions.

- A set of **interrupt outputs** connected to `gr_heep_int_o`. These signals notify the processor when different phases of the computation are completed, such as the end of the computation or the streaming of the output matrix, was_pivot and is_pivot.

Integration in `cv_x_heep_top.sv` At the top level of the platform, the `cv_x_heep_top` module instantiates the `gr_heep_peripherals` block and connects it to the main system interconnect. In particular:

- The accelerator OBI interface is connected to the external bus through the `ext_bus` module, which routes memory transactions between the processor, DMA engine, and external peripherals.
- The register interface is connected to the peripheral configuration bus using an address decoder (`addr_decode` to correctly translate the address (`gr_heep_periph_select`)) and a register demultiplexer (`reg_demux`). This mechanism maps the accelerator register space into the global peripheral address map.
- The interrupt signals generated by the accelerator are connected to the processor through the external interrupt vector (`ext_int_vector`), allowing the CPU to react to hardware events without continuous polling.

Address Space Mapping In addition to the hardware integration, the address space of the accelerator had to be defined within the X-HEEP memory map. This mapping is specified in the platform configuration files `gr_heep_pkg.sv` and `gr_heep.h`, and passed to the system through the `addr_decode` module.
Two address regions are defined:

- An **external peripheral region**, used for the accelerator control registers and accessed by the CPU through memory-mapped transactions.
- An **external slave region**, connected to the external OBI bus and used by the DMA engine to stream the input matrix G to the accelerator.

The external slave region is described through the `ExtSlaveAddrRules` structure in `gr_heep_pkg.sv`, which defines the start and end addresses associated with the accelerator slave interface. Similarly, the register interface address range is defined through the `ExtPeriphAddrRules` structure, which maps the accelerator control registers into the external peripheral address space.
The same configuration file also defines the **number of interrupt lines** exported by the external peripherals. The RREF accelerator interrupts are therefore included in this configuration and connected to the processor through the external interrupt vector of the system.
After this step, the RREF accelerator becomes a fully functional peripheral within the X-HEEP platform, capable of interacting with the processor, the DMA engine, and the memory subsystem. To provide a visual confirmation of the hardware integration, Figures 4.3, 4.4 and 4.5 show the post-implementation hierarchy extracted from Vivado after synthesizing the complete system on the target FPGA platform. These views illustrate how the accelerator is integrated within the X-HEEP peripheral subsystem and how its internal modules are organized as explained above.

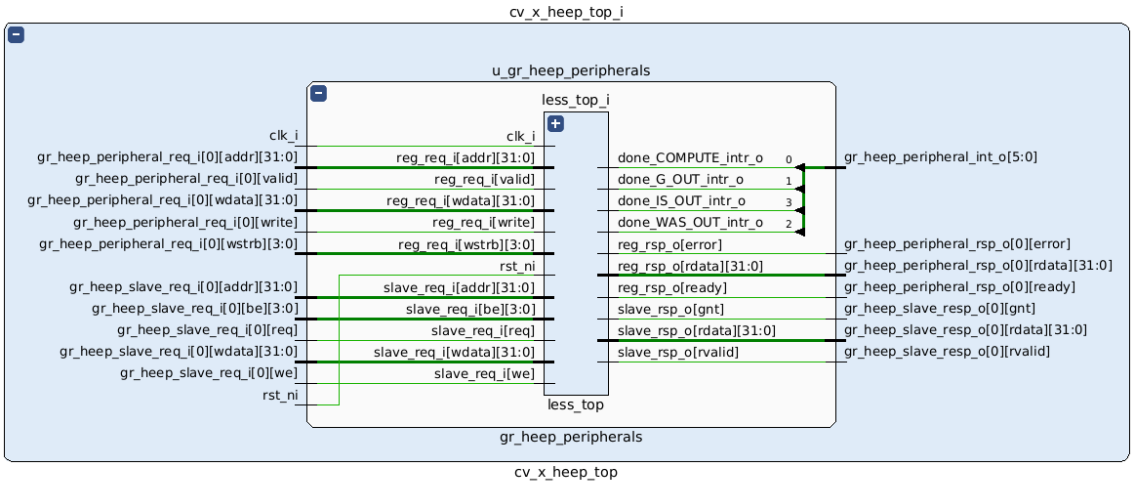

Figure 4.3: Vivado post-implementation view of the LESS RREF external accelerator `less_top` inside `gr_heep_peripherals` of the system top `cv_x_heep_top` on the Zynq UltraScale+ (xczu7ev).

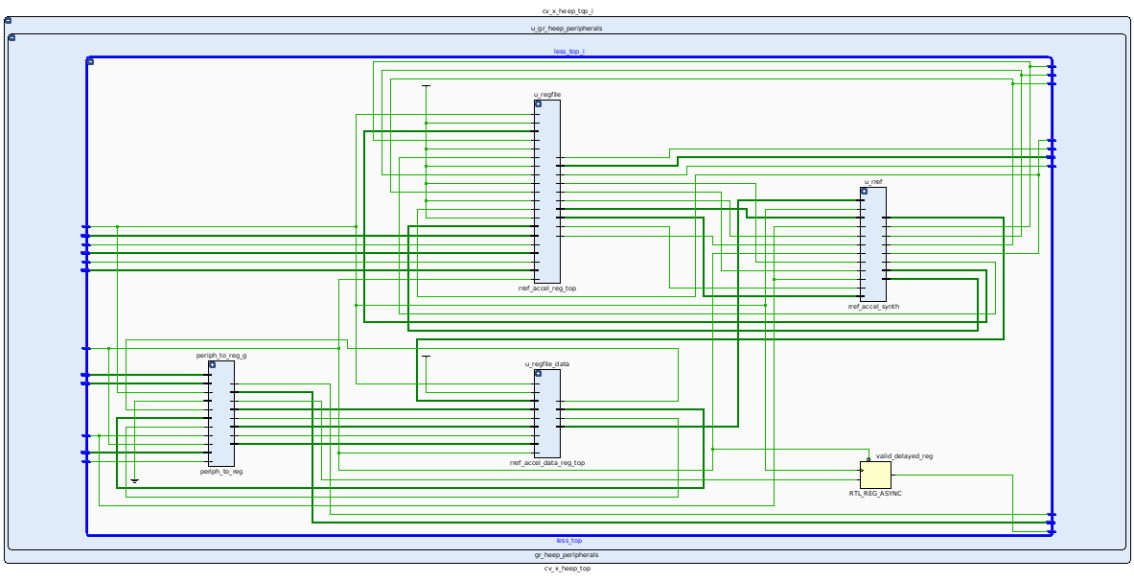

Figure 4.4: Vivado post-implementation view of the LESS RREF accelerator modules inside `less_top`.

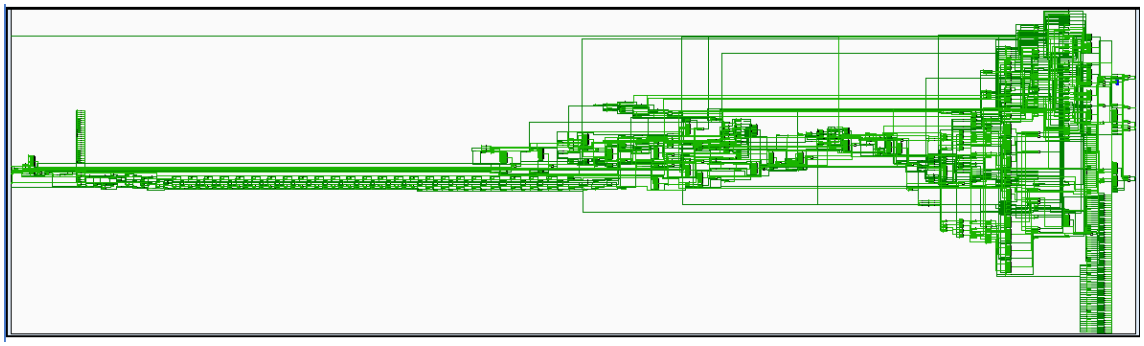

Figure 4.5: Vivado post-implementation view of the LESS RREF accelerator `rref_accel_synth` inside `less_top`.

4.2.4 Step 4: FuseSoC Core

To allow the accelerator to be compiled and integrated within the X-HEEP build system, the design is packaged as a **FuseSoC core**. FuseSoC is an open-source package manager and build system for hardware designs that simplifies the management of HDL sources, dependencies, and tool flows.
In the X-HEEP platform, FuseSoC is used to automatically collect hardware modules, resolve their dependencies, and generate the final set of files required for synthesis or simulation. Each hardware block is described through a `.core` file that specifies the source files, configuration parameters, and compilation targets associated with the module (the whole `vlsi_polito_less.core` is shown in Appendix A.4).

CAPI2 Core Description The accelerator is described using the **CAPI2** (Core API version 2) specification, which defines a structured **YAML**-based format for describing hardware IP blocks. A CAPI2 file contains several sections that specify the metadata of the core, the hardware sources, the parameters, and the available build targets.
The header of the file defines the core name and its description:

- `name`: uniquely identifies the core using a hierarchical naming convention
- `description`: short textual description of the hardware block

In this design the accelerator core is identified as

```
vlsi:polito:less
```

Filesets The `filesets` section defines the HDL sources required to build the accelerator. Each fileset groups together files of the same type and can also specify dependencies on other cores.
In this case, the fileset `files_rtl_sv` includes all the SystemVerilog modules that compose the accelerator, including:

- the automatically generated register interfaces (`rref_accel_reg_top` and `rref_accel_data_reg_top`),
- the memory blocks used to store the matrix data (`g_mem` and `g_mem_simple`),
- the control unit implementing the RREF algorithm (`rref_accel_synth`),
- the top-level wrapper (`less_top`),
- the packages linked to the previous modules.

The fileset also needs a dependency on the core `x-heep::packages`, which provides common SystemVerilog packages used throughout the X-HEEP platform used for type definitions and more.

Core Parameters The CAPI2 specification allows hardware parameters to be exposed to the build system. In this design two parameters are defined:

- `K`: number of rows of the generator matrix
- `N`: number of columns of the generator matrix

Both parameters are defined as `vlogdefine`, meaning that they are passed to the compiler as Verilog preprocessor definitions. This mechanism allows the accelerator to be configured for different matrix sizes without modifying the source code.

Targets The `targets` section defines how the core is built. The `default` target specifies the filesets and the default values for the parameters used during compilation. When the X-HEEP build flow is executed, FuseSoC reads the core file, collects the HDL sources listed in the filesets, resolves all dependencies, and passes the resulting file list and parameters to the selected backend tool (simulation, synthesis, or FPGA build).
Through this mechanism the RREF accelerator becomes a modular and reusable hardware IP that can be automatically integrated into the X-HEEP platform and compiled together with the rest of the system.

4.2.5 Step 5: Driver

After integrating the accelerator into the hardware platform, a software driver was developed to allow the processor to interact with the peripheral through the memory-mapped interface defined in the previous sections. The driver provides a software abstraction layer that hides the low-level register accesses and exposes a function interface compatible with the original software implementation of the RREF algorithm.
The driver is written in the C programming language and interacts with the accelerator using three main mechanisms:

- **Memory-mapped register accesses**, used to configure the accelerator and exchange control information.
- **DMA transfers**, used to efficiently stream the generator matrix between memory and the accelerator.
- **Interrupt handling**, used to notify the processor when different phases of the computation have completed.

Memory-Mapped Register Access The accelerator registers are accessed through Memory-Mapped I/O (MMIO) as peripheral registers are mapped into the processor address space and can therefore be accessed using normal load and store instructions.
The driver defines two small helper functions that simplify register access:

- `write_reg()` performs a 32-bit write to a register.

- `read_reg()` performs a 32-bit read to a register.

The base address of the accelerator peripheral is defined as

```
ADDR = PERIPHERAL_0_PERIPH_START_ADDRESS
```

while the offsets of the individual registers are provided by the header file `less_regs.h` generated through `regtool`. The registers in question, already introduced in chapter 4.2.2, are:

- `CTRL`: used to start the computation or the readback phase
- `STATUS`: reports errors and completion flags
- `WAS_PIVOT` and `IS_PIVOT`: pivot column vectors
- `PIVOT_REUSE_LIMIT`: configuration parameter for the algorithm

Data Packing Since the accelerator interface operates on 32-bit words, while pivot arrays are stored in software as 1-bit values, the driver includes helper functions to convert between these formats.
The function `pack_pivot_word()` groups 32 pivot bits into a single 32-bit word before writing them to the accelerator registers. Conversely, `unpack_pivot_word()` extracts individual pivot bits from the 32-bit words returned by the accelerator.
This packing mechanism reduces the number of memory accesses and matches the 32-bit word-oriented interface of the hardware registers.

Driver Initialization and Interrupt Handling The driver initialization routine `rref_driver_init()` configures the interrupt subsystem of the platform. In particular, it initializes the RISC-V Platform Level Interrupt Controller (RV-PLIC) native to X-HEEP, sets the priority of the interrupts generated by the accelerator, and enables them for the current processor hart.
The following interrupt sources are used:

- computation completed
- generator matrix output completed
- previous pivot vector output completed
- new pivot vector output completed

Each interrupt is associated with a handler (`handler_irq_rref`) that updates a set of software flags. These flags are stored in the global variable `rref_irq_flags`, which is used by the driver to determine when the accelerator has finished a specific operation (through bitmasks) and cleared on exit.

DMA-Based Data Transfer The generator matrix G can be significantly larger than the control register interface. To efficiently transfer this data between system memory and the accelerator, the driver uses the DMA engine.
The DMA is configured using a transaction descriptor (`dma_trans_t`) that specifies:

- source and destination addresses
- data type (32-bit words)
- transfer size
- transfer mode and completion behavior

Both the source and destination endpoints are configured with `DMA_TRIG_MEMORY` triggers, meaning that the transfer is initiated directly by the DMA engine without requiring additional peripheral-side triggering signals. An example of the configuration used in the driver is shown below:

```
dma_target_t tgt_src = {
    .ptr = (uint8_t*)G,
    .inc_d1_du = 1,
    .type = DMA_DATA_TYPE_WORD,
    .trig = DMA_TRIG_MEMORY,
    .env = NULL
};

dma_target_t tgt_dst = {
    .ptr = (uint8_t*)SLAVE_0_START_ADDRESS,
    .inc_d1_du = 0,
    .type = DMA_DATA_TYPE_WORD,
    .trig = DMA_TRIG_MEMORY,
    .env = NULL
};
```

In this configuration the DMA streams the matrix words directly from system memory to the accelerator slave interface on the external bus. The source pointer is incremented after each transfer, while the destination pointer remains fixed because the accelerator exposes a single streaming register for incoming data.
The completion of DMA transfers is configured using `DMA_TRANS_END_POLLING`, meaning that the processor waits for the transfer to complete by polling the DMA status instead of using DMA-generated interrupts:

```
.end = DMA_TRANS_END_POLLING
```

This approach proved to be the most efficient and reliable option in the current system configuration. In particular, DMA interrupt mode was not used because, due to an implementation issue in the current platform, enabling DMA interrupts caused the user-configured interrupts to be disabled. For this reason, the polling-based completion mechanism was selected.
A similar DMA transaction is used during the readback phase to retrieve the processed generator matrix from the accelerator.

Parallel Data Transfers To minimize the overall execution time, the driver performs the transfer of the generator matrix G and the pivot vectors (`was_pivot` and `is_pivot`) in parallel with the accelerator computation whenever possible. During the download phase, the generator matrix is streamed through the DMA engine while the pivot information is written directly to the accelerator registers:

```
for (int i = 0; i < PIVOT_WORDS_COUNT; i++)
    write_reg(was_reg, pack_pivot_word(was_pivot_column, i));
```

In this way, the pivot data transfer overlaps with the DMA streaming of the matrix, reducing idle time and improving the overall execution latency.
Similarly, during the readback phase the updated matrix is retrieved via DMA while the pivot vectors are read directly from the registers:

```
uint32_t w = read_reg(is_reg);
unpack_pivot_word(&w, is_pivot_column, i);
```

This design allows the accelerator datapath, the DMA engine, and the processor to operate concurrently, effectively overlapping data movement and computation.

Accelerator Execution Flow The main driver function `generator_RREF_pivot_reuse()` implements the complete hardware execution flow.
The sequence of operations is the following:

1. The DMA engine is initialized and the pivot reuse limit is written to the configuration register.

2. The input generator matrix G is streamed to the accelerator through the DMA engine.

3. The previous pivot column vector is written to the `WAS_PIVOT` register.

4. The accelerator starts the computation phase and the processor enters a low-power wait state using the `WFI` instruction until the corresponding interrupt is received.

5. At completion the accelerator sends an interrupt (`done_COMPUTE`) that wakes the processor and a second DMA transfer retrieves the updated generator matrix from the accelerator.

6. The old pivot and new pivot vectors are read back from the accelerator registers and unpacked into software arrays.

7. The driver waits for the final output interrupts and checks the `STATUS` register for the value of the `ERROR` bit. If the error bit is set, the function returns a failure code, otherwise it returns success.

Software Integration From the software perspective, the driver function acts as a drop-in replacement for the original software implementation of the RREF pivot reuse algorithm. Instead of performing the matrix reduction operations on the CPU sequentially and in an unoptimized way, the function offloads the computation to the hardware accelerator.
The application code therefore calls the same high-level function, while the driver transparently manages the communication with the accelerator, the DMA transfers, and the synchronization with hardware events.

4.2.6 Step 6: Testing

The final step of the accelerator design workflow is dedicated to functional verification and system-level testing. This step ensures that both the RTL implementation and the deployed hardware produce correct results consistent with the algorithm specification.

ModelSim Testbenches. Several SystemVerilog testbenches were used to validate the `LESS RREF Accelerator`:

- **tb_real.sv:** This testbench drives the accelerator module with matrices imported from text files. The procedure follows three phases:

  1. *Download*: Input matrices (`G_in`) and pivot masks (`was_pivot_column_in_FULL`) are streamed into the DUT through 32-bit parallel ports.
  2. *Compute*: The accelerator performs the RREF computation. The testbench waits for the `done_COMPUTE` flag.
  3. *Upload / Readback*: The computed matrices and pivot arrays are streamed back from the DUT and compared against expected outputs imported from reference files.

  Any mismatches trigger detailed `$display` messages indicating the location of the error. If no mismatches are found, the simulation prints *TEST PASSED*.

- **tb_short.sv:** This testbench follows the same phases as `tb_real.sv` but using a smaller matrix to check on out-of-parameters implementations and also to better compare the results to scratchpad manual computations.

- **tb_g_mem.sv:** Verifies the correctness of the accelerator's internal memory module. The testbench writes and reads 32-bit words to memory addresses, checking the integrity of storage and retrieval.

- **test_fq_ops.sv:** Checks the finite-field arithmetic used in the accelerator (e.g., `fq_mul`, `fq_sub`). Golden reference computations are compared to DUT results to confirm correct modular arithmetic over $\mathbb{F}_Q$.

C Test Suite on X-HEEP. After synthesis and deployment on the `x-heep` core, system-level verification is performed in software:

- **less_driver_test.c and less_test.c**: These programs initialize test matrices in memory and invoke the `generator_RREF_pivot_reuse` function. For hardware-accelerated testing, this function is substituted with a driver function interfacing with the accelerator. While `less_driver_test.c` and `less_test.c` exercise only a single set of inputs and outputs, the Known Answer Test (KAT) suite was executed for all parameter sets ranging from 252_45 to 548_345. Each KAT test performs multiple executions of the RREF function, verifying correctness across key generation, signing, and signature verification operations.
- **Validation**: Final generator matrices and pivot arrays are compared against expected values. Helper functions `print_matrix` and `print_pivot_arrays` allow visual inspection of results.
- **Metrics**: Both small-scale (simple) and full-size tests are executed. The number of cycles used by the accelerator is collected through the `MCYCLE` CSR, used for performance evaluation.
- **Parameter Configuration:** All software parameters, including matrix dimensions K and N, category selection flags, and accelerator type, are defined in `addons.h`. For each category, the corresponding accelerator RTL was synthesized and implemented in Vivado to generate the respective `.bit` files for testing on the real board.

Summary. The combination of ModelSim simulation, software tests on `x-heep`, and full KAT execution ensures correctness of the accelerator from RTL to system-level deployment. This multi-level testing methodology validates both functional and timing correctness while confirming the hardware accelerator can handle all cryptographic operations across the full parameter space, as will be detailed in the Results chapter 5.

4.2.7 Additional Scripts

In addition to the RTL testbenches and C test programs, several auxiliary scripts were developed to automate simulation, test generation, and large-scale verification of the accelerator.

ModelSim compilation scripts. Parameterized `compile.do` scripts (Appendix A.5 and A.6) were used to automate the compilation and simulation of the accelerator within ModelSim. These scripts first define the matrix dimensions K and N as compile-time parameters and later compile the required RTL modules, packages and testbenches before launching the simulation. The scripts also automatically add relevant internal signals to the waveform viewer and execute the simulation until completion. Different variants of the script were used to test both the full accelerator testbench and the internal memory module, simplifying repeated simulations for different parameter configurations.

Python test generation script. A Python script (Appendix A.7) was developed to automatically generate C test libraries starting from reference matrix files obtained from the KATs. The script parses input matrices and pivot information from text files (e.g., `LESS_G0_before_X.txt` and `LESS_G0_after_X.txt`) and produces corresponding C source files (`test_lib_X.c`). These files contain the input generator matrix, pivot masks, and the expected golden outputs used for verification. This automation enabled the generation of large sets of deterministic tests without manual transcription of matrices used before FPGA testing was possible.

Automated verification shell script. A Bash script (Appendix A.8) was implemented to execute large batches of verification tests automatically. For each generated test library file, the script replaces the active test input, recompiles the application, and launches a Verilator-based simulation. The script collects logs, detects build or simulation failures, and scans the output for success messages such as `"Everything matches!"`. At the end of execution, a summary is printed reporting the number of successful tests, mismatches, build failures, and other warnings.

Purpose of the scripts. Together, these scripts significantly simplified the verification workflow by automating repetitive tasks such as test generation, compilation, and execution. This allowed systematic validation of the accelerator across a large number of input matrices and parameter configurations while reducing the possibility of human error during the testing process.

5 Results

This chapter evaluates the proposed RREF accelerator within the complete LESS cryptographic flow. Results are reported in terms of clock cycles, resource utilization, timing, and power consumption.
Unless otherwise stated, all measurements are obtained after place-and-route on a Zynq UltraScale+ ZCU104 FPGA (xczu7ev) at a target frequency of 15 MHz. This relatively low operating frequency is not imposed by the RREF accelerator datapath itself, but by the integration of the design within the complete X-HEEP-based system. In particular, as will be shown later in the timing analysis, the critical paths are located in the surrounding system logic rather than inside the accelerator itself. For this reason, the most meaningful performance indicator throughout this chapter is the clock-cycle reduction achieved by the proposed architecture, while time-based results must be interpreted in light of the system-level operating frequency.

5.1 Standalone RREF Acceleration

To isolate the contribution of the proposed hardware, the matrix-reduction kernel was evaluated independently. The results in Table 5.1 show a consistent speedup of roughly 20× across all security levels. The reported table values are obtained by isolating the first invocation of the function within the SIGN flow; however, equivalent behavior was observed across all other calls, confirming the stability of the measured speedup.

Table 5.1: Clock-cycle results of a single RREF function call: original vs optimized implementations. Speedup is computed as Orig/Opt.

Security Level	Orig [cycles]	Opt [cycles]	**Speedup**
1 (K=126, N=252)	86814359	4186854	**20.73**
3 (K=200, N=400)	310933265	15366401	**20.23**
5 (K=274, N=548)	792047471	37432087	**21.16**

This uniform improvement demonstrates that the accelerator itself is highly effective. The smaller overall gains observed in the full LESS flow as predicted before are therefore attributable to non-accelerated components, primarily hashing operations and control overhead. This confirms that the RREF kernel is no longer the primary performance bottleneck after hardware acceleration, in contrast to the profiling results reported in Table 2.1 of Chapter 2.

5.2 Cycle-Level Performance

Table 5.2 reports the clock-cycle comparison between the original software-oriented implementation and the optimized hardware-accelerated design for the main LESS operations. Since the operating frequency of the integrated prototype is fixed to 15 MHz for the reasons discussed above, cycle counts provide the most direct view of the architectural benefit introduced by the accelerator.
To enable reliable execution on the target platform and avoid excessive runtime and memory pressure, the entire LESS workflow was partitioned into its three main phases (KeyGen, Sign, and Verify), which were executed and measured independently. Since the KeyGen phase does not invoke the accelerated `generator_RREF_pivot_reuse` function, it exhibits no measurable speedup and is therefore omitted from Table 5.2, which focuses on the operations affected by the hardware accelerator.

Table 5.2: Clock-cycle results for the LESS design: original vs. optimized implementations for Sign and Verify. Speedup is computed as Orig/Opt.

Security	Param.	Sign [cycles]			Verify [cycles]		
level		Orig	Opt	**Speedup**	Orig	Opt	**Speedup**
1	LESS_252_45	4440307878	863666272	**5.14**	3041297991	736528516	**4.13**
1	LESS_252_68	6722680410	1310237046	**5.13**	4605871360	1136258476	**4.05**
1	LESS_252_192	19446264859	3578328322	**5.43**	12742010786	3150238465	**4.04**
3	LESS_400_102	35357406412	5225918428	**6.77**	25235481373	4461652076	**5.66**
3	LESS_400_220	76216225830	11204988794	**6.80**	52253304327	9556935647	**5.47**
5	LESS_548_137	81653258828	10411290124	**7.84**	87116518586	8790875421	**9.90**
5	LESS_548_345	293009449312	26151566441	**11.20**	219849144447	24854080054	**8.85**

Key observations. **Sign** and **Verify** both achieve substantial acceleration, with speedups ranging from approximately 4× to over 11× depending on the security level and parameter set.
The limited improvement observed for KeyGen is expected. The key-generation flow relies on the original `generator_RREF` function, whereas the proposed hardware accelerator targets the optimized `generator_RREF_pivot_reuse` function. Consequently, the accelerated design does not affect the KeyGen execution time.
This improvement is expected, as both operations repeatedly invoke the `generator_RREF_pivot_reuse` routine targeted by the proposed hardware accelerator. The repeated execution of this matrix-reduction kernel allows the accelerator to exploit its computational advantage across multiple calls, resulting in significant reductions in total execution time.
The performance gain further increases with the security level and the associated parameter size. Larger parameter sets require more frequent invocations of the matrix-reduction kernel, thereby increasing the fraction of execution time spent in the accelerated region of the design. As the accelerated portion becomes dominant, the relative impact of the remaining unoptimized operations diminishes, leading to progressively higher overall speedups.

Finally, SIGN consistently achieves higher speedup than VERIFY for the same parameter set. This behavior is explained by the larger number of calls to `generator_RREF_pivot_reuse` within the signing flow, which increases the portion of execution time benefiting from hardware acceleration.

Scaling with security level. The system level results reported in Table 5.3 confirm the expected scaling trend. The overall speedup increases from approximately 3.5× at security level 1 up to nearly 10× at level 5. This behavior is consistent with the growing appearance of the accelerated matrix-reduction function at larger parameter sizes. In particular, the higher call density of `generator_RREF_pivot_reuse` reduces the relative weight of non-accelerated components, allowing the proposed architecture to demonstrate its advantages more clearly as the problem size increases.

Table 5.3: Total clock-cycle comparison and overall speedup for the LESS design. The total cycles values for both the original and optimized implementations is obtained as the sum of the three phases (KEYGEN, SIGN and VERIFY). Total speedup is computed as the ratio between the total number of cycles of the original and optimized implementations.

Security	Param.	Total Orig cycles	Total Opt cycles	**Total Speedup**
1	LESS_252_45	8223166910	2342064045	**3.51**
1	LESS_252_68	11660255325	2778199077	**4.20**
1	LESS_252_192	32300854384	6841145525	**4.72**
3	LESS_400_102	61908573762	11003591049	**5.62**
3	LESS_400_220	128898393747	21190788031	**6.08**
5	LESS_548_137	172015671213	22448059344	**7.66**
5	LESS_548_345	513952147039	52099199775	**9.86**

5.3 Resource Utilization

Table 5.4 summarizes the post-synthesis resource usage of the complete LESS design. The equivalent slice metric (KeSlice) reported in the table is computed according to the standard normalization formula

$$\text{KeSlice} = \left(\max \left\{ \frac{\text{LUT}}{8} + BRAM \times 128,\ \frac{\text{FF}}{16} \right\} \right) \times 10^{-3}, \tag{5.1}$$

which reflects the composition of a Xilinx slice in the target FPGA device, where each slice contains up to eight LUTs and sixteen flip-flops. This normalized metric enables technology-consistent area comparisons across different configurations and is commonly adopted in FPGA-based cryptographic accelerators.

Only three hardware configurations were synthesized, one per security level. This is because the accelerator is fully parameterized by the matrix dimensions (K, N), which remain constant within each security category. The different parameter sets within

Table 5.4: Synthesis results for the top-level LESS design executing KeyGen, Sign, and Verify.

Security	Param.	Resources					Freq.	KeyGen			Sign			Verify		
level		LUT	FF	BRAM	DSP	KeSlice	[MHz]	T[ms]	AT[ms]	AT[kCC]	T[ms]	AT[ms]	AT[kCC]	T[ms]	AT[ms]	AT[kCC]
1	LESS_252_45							49457	68300	1024521	57577	79513	1192723	49101	67808	1017145
1	LESS_252_68	2858	895	8	0	1.381	15	22113	30538	458082	87349	120628	1809437	75750	104610	1569172
1	LESS_252_192							7505	10364	155471	238555	329444	4941671	210015	290030	4350479
3	LESS_400_102	3157	1193	20	3	2.954	15	87734	259166	3887524	348394	1029155	15437363	297443	878646	13179720
3	LESS_400_220							28590	84454	1266863	746999	2206635	33099536	637129	1882079	28231187
5	LESS_548_137	3649	1530	40	8	5.576	15	216392	1206601	18099103	694086	3870223	58053353	586058	3267859	49017921
5	LESS_548_345							72903	406507	6097653	1743437	9721404	145821134	1656938	9239086	138586350

the same level (e.g., LESS_252_45, LESS_252_68, LESS_252_192) correspond to distinct algorithmic instances of LESS that share the same matrix size but differ in higher-level protocol parameters. Consequently, they do not require separate hardware synthesis runs and can be grouped together in the Resources side of the table.

Lightweight footprint and system impact. As shown in Table 5.5, the accelerator maintains a modest hardware footprint across all configurations:

- LUT utilization remains below 10% of the system,
- FF usage stays below 7%,
- BRAM usage scales moderately with matrix size,
- DSP usage increases only at higher security levels.

The gradual increase in DSP utilization (from 0 to 8 blocks) is mainly driven by synthesis mapping decisions under higher fan-out and timing pressure, even though the logical parallelism of the Arithmetic Unit remains fixed at four lanes.

Although the proposed design achieves a lightweight area footprint, the relatively high cycle count of the complete LESS protocol dominates the resulting AT values in both representations. Overall, the dedicated accelerator occupies only a small fraction of the available resources of the X-HEEP system, confirming the effectiveness of the lightweight Arithmetic Unit and the memory-streaming architecture.

Table 5.5: Resource utilization of the LESS accelerator and of the complete X-HEEP SoC after synthesis and implementation.

Security	Param.	LUT			FF			BRAM			DSP		
level	set	Accel.	System	**Usage %**	Accel.	System	**Usage %**	Accel.	System	**Usage %**	Accel.	System	**Usage %**
1	LESS_252_45/68/192	2858	38603	**7.4**	895	21529	**4.2**	8	280	**2.9**	0	5	**0.0**
3	LESS_400_102/220	3157	39064	**8.1**	1193	21825	**5.5**	20	292	**6.9**	3	8	**37.5**
5	LESS_548_137/345	3649	39731	**9.2**	1530	22160	**6.9**	40	312	**12.8**	8	13	**61.5**

Area–Time metric. The Area–Time (AT) product is used to jointly evaluate hardware cost and performance. In Table 5.4, the AT metric is reported in two forms.

The first and more technology-independent metric is expressed in thousands of clock cycles:

$$AT_{\mathrm{kCC}} = \mathrm{KeSlice} \times \mathrm{kCC}, \tag{5.2}$$

where kCC denotes the execution latency in thousands of clock cycles. This formulation is independent of the operating frequency and therefore provides a more objective comparison metric. In particular, it avoids bias introduced by the relatively low operating frequency (15 MHz) of the target platform and will be especially useful in Table 5.8, where the proposed design is compared against implementations running on higher-frequency devices. For completeness, the time-based metric expressed in milliseconds is also reported:

$$AT_{\mathrm{ms}} = \mathrm{KeSlice} \times t_{\mathrm{ms}}, \tag{5.3}$$

where t_{ms} is the measured execution time in milliseconds. While AT_{ms} directly reflects wall-clock performance on the target board, it is inherently frequency-dependent.
Although the absolute resource footprint of the accelerator is small, the overall Area–Time (AT) product remains relatively large due to the still significant execution latency of the complete LESS flow.

5.4 Timing Analysis

Post-implementation timing results are reported in Table 5.6. All configurations exhibit large positive worst negative slack (WNS), exceeding +28 ns.

Table 5.6: Post-implementation timing results (Vivado).

Security level	Parameter set	WNS [ns]	WHS [ns]
1	LESS_252_45/68/192	+28.638	+0.017
3	LESS_400_102/220	+28.923	+0.010
5	LESS_548_137/345	+29.123	+0.011

This significant timing margin indicates that the design is far from timing-critical at the target operating frequency. The large positive setup slack confirms that all configurations comfortably meet the 15 MHz constraint with substantial headroom.
An inspection of the post-implementation timing reports reveals that the worst timing paths never traverse the proposed RREF accelerator. Instead, the critical paths are located in the surrounding system logic. This observation confirms that the accelerator does not impose any frequency bottleneck when integrated into the full LESS design.
The worst hold slack (WHS) values are also positive for all configurations, indicating that hold constraints are safely met after place-and-route. Although the WHS margin is naturally much smaller than the setup slack, this behavior is expected and does not raise timing concerns.
Overall, the performance of the proposed architecture is therefore not limited by the

achievable clock frequency, but rather by the total cycle count of the LESS protocol, which is dominated by memory streaming and control overhead, as discussed in previous sections.

5.5 Power Consumption

Table 5.7 reports the estimated power consumption. The total power remains approximately constant (around 750 mW) across security levels.

Table 5.7: Post-implementation power estimation obtained from Vivado at the target operating frequency.

Security level	Parameter set	Static [mW]	Dynamic [mW]	Total [mW]
1	LESS_252_45/68/192	617	137	754
3	LESS_400_102/220	617	134	752
5	LESS_548_137/345	617	138	755

The static power component (approximately 617 mW) represents the baseline consumption of the FPGA device and is largely independent of the accelerator activity. It is mainly determined by leakage currents, bias networks, and the clocking infrastructure that remain active regardless of the workload. As expected, this term is identical across all security levels.
In contrast, the dynamic power (approximately 134–138 mW) accounts for the switching activity of logic elements, BRAMs, routing, and clock distribution during operation. Unlike static power, this component depends on signal transitions and activity factors rather than solely on the nominal matrix dimensions. Consequently, it does not scale monotonically with increasing security level, but instead exhibits small variations across configurations.
This behavior is expected because:

- the control structure and clock network remain unchanged,
- the Arithmetic Unit is lightweight and mostly combinational,
- the design is dominated by memory activity rather than deeply pipelined arithmetic.

Overall, the relatively small dynamic power variation confirms that the parallel four-lane datapath does not introduce significant switching overhead, while the total power is dominated by the device static component.

5.6 Comparison with other implementations

Table 5.8 compares the proposed architecture with representative state-of-the-art FPGA implementations of post-quantum digital signature schemes, including CROSS by Karl et al.[32], CRYSTALS-Dilithium by Zhao et al.[16], and the original LESS

hardware reported by Beckwith et al.[21] The comparison includes resource utilization, operating frequency, clock cycles, and the area–time (AT) product. The proposed design was implemented on a Zynq Ultrascale+ ZCU104 (XCZU7EV) platform, while all the other referenced implementations target Xilinx Artix-7 FPGAs. Although the target devices differ, the comparison remains meaningful as all the considered designs implement digital signature schemes and are evaluated using the same performance metrics.

Since the FPGA architectures differ, the normalized area metric (KeSlice) is computed using the appropriate slice composition for each device family. For the proposed implementation on the UltraScale+ platform, the metric follows the formulation introduced in Eq. (5.1). For the referenced implementations targeting Artix-7 devices, where each slice contains four LUTs and eight flip-flops, the equivalent slice metric is computed as

$$\text{KeSlice} = \left(\max \left\{ \frac{\text{LUT}}{4} + \text{BRAM} \times 128,\ \frac{\text{FF}}{8} \right\} \right) \times 10^{-3}. \tag{5.4}$$

Table 5.8: Comparison of the proposed LESS hardware accelerator with representative state-of-the-art FPGA implementations of post-quantum digital signature schemes. The table reports resource utilization, operating frequency, clock cycles (kCC), and the area–time (AT) product for the key generation, signing, and verification operations. Runtimes are expressed in milliseconds, while the AT metric is computed both as *KeSlice*$\times t_{\text{ms}}$ and *KeSlice*$\times$ kCC (kilo clock cycles). The proposed design was implemented on a Xilinx Zynq Ultrascale+ ZCU104 (XCZU7EV), whereas the other referenced implementations target Xilinx Artix-7 devices.

Ref.	Design	Resources					Freq.	KeyGen			Sign			Verify		
		LUT	FF	BRAM	DSP	KeSlice	[MHz]	[kCC]	AT[ms]	AT[kCC]	[kCC]	AT[ms]	AT[kCC]	[kCC]	AT[ms]	AT[kCC]
Karl et al.[32]	CROSS-RSDP-1-b	27308	11820	57.5	0	19.0	111	4.107	0.708	78.033	95.016	16.3	1805	68.598	11.8	1303
Zhao et al.[16]	DILITHIUM-L2	29998	10366	11	10	8.908	96.9	4.172	0.38	37.16	28.091	2.58	250.23	4.422	0.41	39.39
	DILITHIUM-L3							5.851	0.54	52.12	44.706	4.11	398.24	6.181	0.57	55.06
	DILITHIUM-L5							8.765	0.81	78.08	48.996	4.50	436.46	9.039	0.83	80.52
Beckwith et al. LESS(2023)[21]	LESS-L1-b	54800	39900	59.5	0	21.316	200	29.1	3.10	620.30	5204.6	554.71	110941.25	5156.2	549.55	109909.56
	LESS-L1-i							77.5	8.26	1651.99	5126.4	546.37	109274.34	5093.2	542.83	108566.65
	LESS-L1-s							174.5	18.60	3719.64	4166.1	444.02	88804.59	4137.2	440.94	88188.56
	LESS-L3-b	76700	57900	102.5		32.295	167	72.1	13.94	2328.47	39237.4	7587.86	1267171.83	39146.0	7570.18	1264220.07
	LESS-L3-s							132.8	25.68	4288.78	46216.7	8937.53	1492568.33	46142.8	8923.24	1490181.73
	LESS-L5-b	104300	76700	167.5		47.515	143	134.4	44.66	6386.02	129885.6	43157.44	6171514.28	129726.1	43104.45	6163935.64
	LESS-L5-s							247.9	82.37	11778.97	87161.5	28961.39	4141478.67	87013.8	28912.31	4134460.71
OURS	LESS-L1-b (252_192)	2858	895	8	0	1.381	15	112578	10364	155471	3578328	329444	4941671	3150238	290030	4350479
	LESS-L1-i (252_68)							331703	30538	458082	1310237	120628	1809437	1136258	104610	1569172
	LESS-L1-s (252_45)							741869	68300	1024521	863666	79513	1192723	736528	67808	1017145
	LESS-L3-b (400_220)	3157	1193	20	3	2.954		428863	84454	1266863	11204988	2206635	33099536	9556935	1882079	28231187
	LESS-L3-s (400_102)							1316020	259166	3887524	5225918	1029155	15437363	4461652	878646	13179720
	LESS-L5-b (548_345)	3649	1530	40	8	5.576		1093553	406507	6097653	26151566	9721404	145821134	24854080	9239086	138586350
	LESS-L5-s (548_137)							3245893	1206601	18099103	10411290	3870223	58053353	8790875	3267859	49017921

Area Efficiency

The most evident result is the extremely compact footprint of the proposed accelerator. For LESS-L1, the complete design requires only 2,858 LUTs and 895 FFs, compared to 54,800 LUTs and 39,900 FFs reported in the 2023 LESS implementation[21]. Even at higher security levels, the area remains below 3,700 LUTs, which is more than an order of magnitude smaller than prior LESS hardware.

The same trend holds for BRAM usage. While previous LESS designs[21] require between 59.5 and 167.5 BRAM blocks, the proposed implementation uses only 8 BRAMs at L1, 20 at L3, and 40 at L5. This confirms that the architecture is strongly

optimized for silicon efficiency and targets resource-constrained FPGA platforms. In terms of KeSlice, the proposed design also achieves the lowest normalized area metric across all configurations, highlighting its minimal hardware footprint.

Operating Frequency and Clock Cycles

The proposed design operates at a lower frequency (15 MHz) than prior implementations, which target Xilinx Artix-7 devices running at much higher clock rates. As anticipated at the beginning of this chapter, this difference is mainly due to the integration of the accelerator within the complete X-HEEP SoC rather than to intrinsic limitations of the RREF datapath itself. In particular, the post-implementation timing analysis shows that the critical paths are located in the surrounding system logic, while the accelerator remains largely memory-bound and does not represent the timing bottleneck of the design.
However, due to the deliberately lightweight four-lane architecture and limited parallelism, the number of clock cycles required to complete operations is higher. For instance, at security level L1-b, KeyGen requires 112,578 kCC, compared to 29.1 kCC in the original LESS hardware[21]. This reflects a conscious trade-off: the design prioritizes minimal area over peak throughput, reusing a small arithmetic unit and streaming data from memory instead of fully parallel processing entire matrix rows. As a result, latency is higher, but the design achieves extremely low area and resource utilization.

5.7 Future Improvements

Although the proposed architecture demonstrates that a lightweight RREF accelerator can significantly improve the performance of the LESS cryptographic workflow while maintaining a small hardware footprint, several design directions could further improve the overall throughput of the system.
A first direction for improvement concerns the increase of both **computational and memory-level parallelism**. The current architecture processes four field elements per cycle, matching the 32-bit memory word organization of the generator matrix memory. While this choice keeps the design compact and resource-efficient, it limits the throughput of the row operations that dominate the RREF algorithm. Future implementations could extend the arithmetic unit with more parallel lanes together with a wider memory interface or additional memory banks, potentially enabling the processing of entire matrix rows in parallel rather than only four elements per cycle. Such an approach would significantly reduce the number of iterations required for row normalization and elimination, particularly for the larger LESS parameter sets, at the cost of increased hardware resources and memory bandwidth.
Another possible improvement concerns the reduction of memory access latency through **better overlap between memory transfers and computation**. Additional pipelining or buffering strategies could allow the accelerator to preload the next matrix elements while the arithmetic unit processes the current ones, reducing idle cycles and improving the effective utilization of the datapath.
Finally, future work could explore the acceleration of additional components of the

LESS workflow. Once the RREF kernel is accelerated, other operations such as hashing and canonical form computation represent a larger fraction of the execution time. Investigating hardware support for these operations could further improve the overall performance of the complete cryptographic protocol.

6 Conclusion

The transition toward **PQC** requires not only the development of quantum-resistant cryptographic algorithms but also efficient implementation strategies capable of supporting their significantly increased computational complexity. In particular, some PQC schemes rely heavily on large-scale linear algebra operations, which can become a major performance bottleneck when implemented purely in software on embedded systems.

This thesis investigated the acceleration of the **RREF** computation used within the **LESS** signature scheme, a code-based post-quantum cryptographic algorithm. Profiling performed using ASIP Designer revealed that the matrix reduction routine dominates the execution time of the LESS signing procedure, making it a natural candidate for hardware acceleration.

Based on this observation, a dedicated hardware architecture was developed to accelerate matrix reduction over the finite field GF(127). The accelerator was designed as a lightweight architecture based on memory streaming and limited parallel arithmetic units, and was integrated as a **loosely coupled** peripheral within the RISC-V–based X-HEEP microcontroller platform.

Experimental evaluation on a **Xilinx Zynq UltraScale+** FPGA shows that the proposed accelerator significantly improves the execution time of the RREF kernel, achieving an average speedup of approximately 20× compared to the software implementation. When integrated within the complete LESS workflow, this improvement results in overall speedups ranging from about 3.5× to nearly 10× depending on the security level, while maintaining a small hardware footprint suitable for embedded platforms.

Overall, the results show that targeted hardware acceleration of dominant computational kernels can substantially improve the practical performance of post-quantum cryptographic schemes. The proposed architecture highlights how careful design trade-offs between parallelism, memory bandwidth, and hardware complexity can enable efficient PQC implementations even in constrained environments.

Future work may explore further architectural optimizations, including increased parallelism, tighter integration with the processor, or support for additional computational kernels used in post-quantum cryptographic schemes. Extending the approach to alternative hardware platforms or ASIC implementations may further contribute to the practical deployment of PQC in real-world embedded systems.

A Code Snippets

```
/// \param G[in/out]: generator matrix K \times N
/// \param is_pivot_column[out]: N bytes, set to 1 if this column
///                  is a pivot column
/// \param was_pivot_column[out]: N bytes, set to 1 if this column
///                  is a pivot column
/// \param pvt_reuse_limit:[in]:
/// \return 0 on failure
///         1 on success
int generator_RREF_pivot_reuse(generator_mat_t *G,
                               uint8_t is_pivot_column[N],
                               uint8_t was_pivot_column[N],
                               const int pvt_reuse_limit) {
    int pvt_reuse_cnt = 0;

     // row swap pre-process - swap previous pivot elements to corresponding row to reduce likelihood of corruption
     if (pvt_reuse_limit != 0) {
         for (int preproc_col = K - 1; preproc_col >= 0; preproc_col--) {
             if (was_pivot_column[preproc_col] == 1) {
                 // find pivot row
                 uint32_t pivot_el_row = -1;
                 for (uint32_t row = 0; row < K; row = row + 1) {
                     if (G->values[row][preproc_col] != 0) {
                         pivot_el_row = row;
                     }
                 }
                 swap_rows(G->values[preproc_col], G->values[pivot_el_row]);
             }
         }
     }

     for (uint32_t row_to_reduce = 0; row_to_reduce < K; row_to_reduce++) {
         uint32_t pivot_row = row_to_reduce;
```

```
33          /*start by searching the pivot in the col = row*/
34          uint32_t pivot_column = row_to_reduce;
35          while ((pivot_column < N) && (G->values[pivot_row][
    pivot_column] == 0)) {
36              while ((pivot_row < K) && (G->values[pivot_row][
    pivot_column] == 0)) {
37                  pivot_row++;
38              }
39              if (pivot_row >= K) { /*entire column tail swept
    */
40                  pivot_column++; /* move to next col */
41                  pivot_row = row_to_reduce; /*starting from
    row to red */
42              }
43          }
44          if (pivot_column >= N) {
45              return 0; /* no pivot candidates left, report
    failure */
46          }
47          is_pivot_column[pivot_column] = 1; /* pivot found,
    mark the column*/
48
49          /* if we found the pivot on a row which has an index
    > pivot_column
50           * we need to swap the rows */
51          if (row_to_reduce != pivot_row) {
52              was_pivot_column[pivot_row] = 0; // pivot no
    longer reusable - will be corrupted during reduce row
53              swap_rows(G->values[row_to_reduce], G->values[
    pivot_row]);
54          }
55          pivot_row = row_to_reduce; /* row with pivot now in
    place */
56
57          /// NOTE: this needs explenation. We can skip the
    reduction of the pivot row, because for
58          /// the CF it doesnt matter. The only thing that is
    important for the CF is the number of
59          /// zeros, and this doest change if we reduce a
    reused pivot row.
60          if (((was_pivot_column[pivot_column] == 1) && (
    pvt_reuse_cnt < pvt_reuse_limit) && (pivot_column < K))) {
61              pvt_reuse_cnt++;
62              continue;
63          }
64
65          /* Compute rescaling factor */
66          const FQ_ELEM scaling_factor = fq_inv(G->values[
    pivot_row][pivot_column]);
```

```
67
68          /* rescale pivot row to have pivot = 1. Values at the
    left of the pivot
69           * are already set to zero by previous iterations */
70          for (uint32_t i = pivot_column; i < N; i++) {
71              G->values[pivot_row][i] = fq_mul(scaling_factor,
    G->values[pivot_row][i]);
72          }
73
74          /* Subtract the now placed and reduced pivot rows,
    from the others,
75           * after rescaling it */
76          for (uint32_t row_idx = 0; row_idx < K; row_idx++) {
77              if (row_idx != pivot_row) {
78                  FQ_ELEM multiplier = G->values[row_idx][
    pivot_column];
79                  /* all elements before the pivot in the pivot
    row are null, no need to
80                   * subtract them from other rows. */
81                  for (int col_idx = 0; col_idx < N; col_idx++)
    {
82                      FQ_ELEM tmp = fq_mul(multiplier, G->
    values[pivot_row][col_idx]);
83                      G->values[row_idx][col_idx] = fq_sub(G->
    values[row_idx][col_idx], tmp);
84                  }
85              }
86          }
87      }
88
89      return 1;
90  } /* end generator_RREF_pivot_reuse */
```

Listing A.1: generator_RREF_pivot_reuse()

```
1  {
2    name: "rref_accel",
3    clock_primary: "clk_i",
4    reset_primary: "rst_ni",
5    bus_interfaces: [
6      { protocol: "reg_iface", direction: "device" }
7    ],
8    regwidth: "32",
9    registers: [
10     {
11       name: "WAS_PIVOT",
12       desc: "Input pivot-column 32 elements (each 1 bit)",
13       swaccess: "rw",
14       hwaccess: "hrw",
15       hwqe: "true",
```

```
      hwre: "true",
      hwext: "true",
      fields: [
        { bits: "31:0"}
      ]
    },
    {
      name: "IS_PIVOT",
      desc: "Output pivot-column 32 elements (each 1 bit)",
      swaccess: "ro",
      hwaccess: "hrw",
      hwre: "true",
      hwext: "true",
      fields: [
        { bits: "31:0"}
      ]
    },
    {
      name: "CTRL",
      desc: "Control register for RREF accelerator",
      swaccess: "wo",
      hwaccess: "hro",
      fields: [
        { bits: "0", name: "START", desc: "Begin streaming G matrix and Was matrix" },
        { bits: "1", name: "START_READBACK", desc: "Enable result streaming (G, was_pivot, is_pivot)" },
      ]
    },
    {
      name: "STATUS",
      desc: "Status register of RREF accelerator",
      swaccess: "ro",
      hwaccess: "hwo",
      fields: [
        { bits: "0", name: "ERROR", desc: "1 if accelerator finishes with no pivot or with errors" },
        { bits: "1", name: "COMPUTE_DONE", desc: "Computation finished" },
        { bits: "2", name: "G_OUT_DONE", desc: "1 = G readback finished" },
        { bits: "3", name: "WAS_OUT_DONE", desc: "1 = was_pivot readback finished" },
        { bits: "4", name: "IS_OUT_DONE", desc: "1 = is_pivot readback finished" }
      ]
    },
    {
      name: "PIVOT_REUSE_LIMIT",
```

```
      desc: "Pivot Reuse limit value of RREF accelerator",
      swaccess: "wo",
      hwaccess: "hro",
      fields: [
        { bits: "31:0"}
      ]
    }
  ]
}
```

Listing A.2: rref_accel.hjson

```
{
    name: "rref_accel_data",
    clock_primary: "clk_i",
    reset_primary: "rst_ni",
    bus_interfaces: [
    { protocol: "reg_iface", direction: "device" }
    ],
    regwidth: "32",
    registers: [
        { name: "G"
          desc: "G matrix 4 bytes elements"
            swaccess: "rw",
            hwaccess: "hrw",
              hwqe: "true",
              hwre: "true",
            hwext : "true",
            fields: [
                { bits: "31:0"
                }
            ]
        }
    ],
}
```

Listing A.3: rref_accel_data.hjson

```
CAPI=2:

# Copyright 2025 POLITO
# Solderpad Hardware License, Version 2.1, see LICENSE.md for
    details.
# SPDX-License-Identifier: Apache-2.0 WITH SHL-2.1

name: "vlsi:polito:less"
description: "LESS RREF Accelerator HDL"

filesets:
```

```
11
12    files_rtl_sv:
13      depend:
14      - x-heep::packages          # standard packages from X-HEEP
15      files:
16      - hw/rtl/gen_sv/rref_accel_reg_pkg.sv
17      - hw/rtl/gen_sv/rref_accel_reg_top.sv
18      - hw/rtl/gen_sv/rref_accel_data_reg_pkg.sv
19      - hw/rtl/gen_sv/rref_accel_data_reg_top.sv
20      - hw/rtl/rref_accel_synth_pkg.sv
21      - hw/rtl/g_mem_simple.sv
22      - hw/rtl/g_mem.sv
23      - hw/rtl/rref_accel_synth.sv        # control unit
24      - wrapper/less_top.sv               # top-level wrapper
     connecting CU + DP
25      file_type: systemVerilogSource
26
27  parameters:
28    K:
29      datatype: int
30      paramtype: vlogdefine
31      description: |
32        Number of rows of Generator matrix
33      default: 3
34
35    N:
36      datatype: int
37      paramtype: vlogdefine
38      description: |
39        Number of columns of Generator matrix
40      default: 6
41
42  targets:
43    default:
44      filesets:
45      - files_rtl_sv
46      parameters:
47        - K=3
48        - N=6
```

Listing A.4: vlsi_polito_less.core

```
1  # ============================================================
2  # RREF Accelerator Simulation Script (Parameterized)
3  # ============================================================
4
5  # ---- Parameters ----
6  set K 126
7  set N 252
8
```

```
# ---- Common defines ----
set DEFINES "+define+K=$K +define+N=$N"

# ---- Compile design files ----
vlog -work work $DEFINES ./hw/rtl/rref_accel_synth_pkg.sv
vlog -work work $DEFINES ./hw/rtl/g_mem_simple.sv
vlog -work work $DEFINES ./hw/rtl/g_mem.sv
vlog -work work $DEFINES ./hw/rtl/rref_accel_synth.sv
vlog -work work $DEFINES ./hw/tb/pkg/import_matrix_pkg.sv
vlog -work work +incdir+./hw/tb/pkg +incdir+./hw/rtl $DEFINES
    ./hw/tb/test_real.sv

# ---- Simulate ----
vsim work.tb_rref_accel_synth -voptargs=+acc

# ---- Add signals to waveform ----
add wave -r /*
add wave sim:/tb_rref_accel_synth/dut/was_pivot_column_in
add wave sim:/tb_rref_accel_synth/dut/was_pivot_column_out
##add wave sim:/tb_rref_accel_synth/dut/is_pivot_column
##add wave sim:/tb_rref_accel_synth/dut/G
add wave sim:/tb_rref_accel_synth/dut/was_pivot
add wave sim:/tb_rref_accel_synth/dut/is_pivot
run -all
```

Listing A.5: compile.do

```
# ============================================================
# RREF Accelerator Simulation Script (Parameterized)
# ============================================================

# ---- Parameters ----
set K 3
set N 6

# ---- Common defines ----
set DEFINES "+define+K=$K +define+N=$N"

# ---- Compile design files ----
vlog -work work $DEFINES ./hw/rtl/rref_accel_synth_pkg.sv
vlog -work work $DEFINES ./hw/rtl/g_mem_simple.sv
vlog -work work $DEFINES ./hw/rtl/g_mem.sv
vlog -work work +incdir+./hw/tb/pkg +incdir+./hw/rtl $DEFINES
    ./hw/tb/test_g_mem.sv

# ---- Simulate ----
vsim work.tb_g_mem -voptargs=+acc

# ---- Add signals to waveform ----
add wave -r /*
```

```
#add wave -position insertpoint sim:/tb_g_mem/ref_mem
run -all
```

Listing A.6: compile_g_mem.do

```
#!/usr/bin/env python3
import os
import re

# --------------------------
# Configuration
# --------------------------
ACTUALTEST = "./sw/applications/LESS_548_345_sw"  # path to LESS.c
INPUT_FOLDER = ACTUALTEST + "/TESTS"  # folder containing LESS_G0...
OUTPUT_FOLDER = ACTUALTEST + "/TESTS"  # folder where test_lib_X.c will be written
FILE_PREFIX = "test_lib"  # output C file prefix
N_ROWS_G = 274  # number of matrix rows to read

# Make sure output folder exists
os.makedirs(OUTPUT_FOLDER, exist_ok=True)


# --------------------------
# Helper functions
# --------------------------
def read_matrix_and_pivots(filename):
    matrix = []
    is_pivot = []
    was_pivot = []

    with open(filename, "r") as f:
        lines = [la.strip() for la in f.readlines()]

    # --- G first N_ROWS_G rows ---
    g_lines = [la for la in lines[:N_ROWS_G] if la]
    for la in g_lines:
        matrix.append([int(x) for x in la.split()])

    # --- Footer ---
    footer = lines[N_ROWS_G:]
    for line in footer:
        if line.startswith("is_pivot_column"):
            nums = line.split(":", 1)[1].split()
            is_pivot = [int(x) for x in nums]

        if line.startswith("permuted_pivot_flags"):
            nums = line.split(":", 1)[1].split()
```

```
            was_pivot = [int(x) for x in nums]

    return matrix, is_pivot, was_pivot


def generate_c_file(input_file_in, input_file_out, number):
    matrix, is_pivot, was_pivot = read_matrix_and_pivots(input_file_in)
    matrix_golden, is_pivot_golden, was_pivot_golden = read_matrix_and_pivots(
        input_file_out
    )

    if not matrix or not matrix_golden:
        print(f"File vuoto: {input_file_in} / {input_file_out}")
        return

    rows = len(matrix)
    cols = max(len(r) for r in matrix)

    # pad rows to same length
    for r in matrix:
        r.extend([0] * (cols - len(r)))
    for r in matrix_golden:
        r.extend([0] * (cols - len(r)))

    output_file = os.path.join(OUTPUT_FOLDER, f"{FILE_PREFIX}_{number}.c")

    with open(output_file, "w") as f:
        f.write('#include "test_lib.h"\n#ifndef KAT\n#ifdef full\n')

        # G_in
        f.write("generator_mat_t G_in = {{\n")
        for i, row in enumerate(matrix):
            f.write("    {" + ", ".join(str(x) for x in row) + "}")
            f.write(",\n" if i < rows - 1 else "\n")
        f.write("}};\n\n")

        # was_pivot_in
        f.write("uint8_t was_pivot_in[N] = {\n    ")
        f.write(", ".join(str(x) for x in was_pivot))
        f.write("\n};\n\n")

        # G_out_golden
        f.write("const generator_mat_t G_out_golden = {{\n")
```

```
86          for i, row in enumerate(matrix_golden):
87              f.write("    {" + ", ".join(str(x) for x in row)
    + "}")
88              f.write(",\n" if i < rows - 1 else "\n")
89          f.write("}};\n\n")
90
91          # is_pivot_out_golden
92          f.write("const uint8_t is_pivot_out_golden[N] = {\n
      ")
93          f.write(", ".join(str(x) for x in is_pivot_golden))
94          f.write("\n};\n\n")
95
96          # was_pivot_out_golden
97          f.write("const uint8_t was_pivot_out_golden[N] = {\n
       ")
98          f.write(", ".join(str(x) for x in was_pivot_golden))
99          f.write("\n};\n")
100
101         f.write("#endif\n#endif\n")
102
103     print(f"File '{output_file}' generato con successo.")
104
105
106 def main():
107     before_files = [
108         f for f in os.listdir(INPUT_FOLDER) if re.match(r"
    LESS_G0_before_\d+\.txt$", f)
109     ]
110     before_files.sort(
111         key=lambda x: int(re.search(r"LESS_G0_before_(\d+)\.
    txt$", x).group(1))
112     )
113
114     for before_file in before_files:
115         match = re.search(r"LESS_G0_before_(\d+)\.txt$",
    before_file)
116         if not match:
117             print(f"WARNING: Cannot extract number from {
    before_file}")
118             continue
119
120         number = int(match.group(1))
121         after_file = f"LESS_G0_after_{number:03}.txt"
122         before_path = os.path.join(INPUT_FOLDER, before_file)
123         after_path = os.path.join(INPUT_FOLDER, after_file)
124
125         if os.path.exists(after_path):
126             generate_c_file(before_path, after_path, number)
127         else:
```

```
            print(f"WARNING: Corresponding after file not found for {before_file}")


if __name__ == "__main__":
    main()
```

Listing A.7: script_test_libs_first.py

```
#!/bin/bash

set -euo pipefail

TEST_DIR="/home/thesis/giuseppe.cutrera/Desktop/LESS_ALGO/prova_files/github2_main/LESS/sw/applications/LESS_400_102/TESTS"
MAIN_TEST="/home/thesis/giuseppe.cutrera/Desktop/LESS_ALGO/prova_files/github2_main/LESS/sw/external/lib/my_additions/test_lib.c"

OUT_DIR="verilator_outputs"
mkdir -p "$OUT_DIR"

# Counters
count_match=0
count_mismatch=0
count_build_fail=0
count_notfound=0
count_verilator_fail=0

for TESTFILE in $(find "$TEST_DIR" -maxdepth 1 -name 'test_lib_*.c' | sort -V); do
    BASENAME=$(basename "$TESTFILE")
    TESTNAME="${BASENAME%.c}"

    echo "======================================="
    echo " Running test: $TESTNAME"
    echo "======================================="

    cp "$TESTFILE" "$MAIN_TEST"

    BUILD_LOG="$OUT_DIR/${TESTNAME}_build.log"
    VERILATOR_LOG="$OUT_DIR/${TESTNAME}_verilator.log"

    echo "--> Running: make app PROJECT=less_test"
    if ! make app PROJECT=less_test > "$BUILD_LOG" 2>&1; then
        echo " Build failed: see $BUILD_LOG"
        ((count_build_fail++))
        continue
    fi
```

```
37
38      echo "--> Running: make verilator-opt"
39      if ! make verilator-opt > "$VERILATOR_LOG" 2>&1; then
40          echo " Verilator run failed: see $VERILATOR_LOG"
41          ((count_verilator_fail++))
42          continue
43      fi
44
45      # ---- Check messages ----
46
47      if grep -q "Everything matches!" "$VERILATOR_LOG"; then
48          echo "RESULT for $TESTNAME: Everything matches!"
49          ((count_match++))
50      else
51          echo "RESULT for $TESTNAME: MISMATCH - message not
    found!"
52          ((count_mismatch++))
53      fi
54
55      # Check for "not found" error lines
56      if grep -qi "not found" "$VERILATOR_LOG"; then
57          echo "  WARNING: 'not found' detected in $TESTNAME"
58          ((count_notfound++))
59      fi
60
61      echo "Finished test $TESTNAME. Logs in $OUT_DIR/"
62  done
63
64  echo ""
65  echo "==================== SUMMARY ===================="
66  echo " Matching tests       : $count_match"
67  echo " Mismatching tests    : $count_mismatch"
68  echo " Build failures       : $count_build_fail"
69  echo " Verilator failures   : $count_verilator_fail"
70  echo " 'not found' warnings : $count_notfound"
71  echo "================================================="
```

Listing A.8: script_test_libs.sh

Bibliography

[1] Mihir Bellare and Phillip Rogaway. *Introduction to Modern Cryptography*. p. 10. Sept. 2005. URL: https://web.cs.ucdavis.edu/~rogaway/classes/227/spring05/book/main.pdf.

[2] Ronald L. Rivest, Adi Shamir, and Leonard Adleman. "A Method for Obtaining Digital Signatures and Public-Key Cryptosystems". In: *Communications of the ACM* 21.2 (1978), pp. 120–126.

[3] Whitfield Diffie and Martin E. Hellman. "New Directions in Cryptography". In: *IEEE Transactions on Information Theory* 22.6 (1976), pp. 644–654.

[4] Neal Koblitz. "Elliptic Curve Cryptosystems". In: *Mathematics of Computation* 48.177 (1987), pp. 203–209.

[5] Victor S. Miller. "Use of Elliptic Curves in Cryptography". In: *Advances in Cryptology — CRYPTO '85* (1986), pp. 417–426.

[6] Peter W. Shor. "Algorithms for Quantum Computation: Discrete Logarithms and Factoring". In: *Proceedings of the 35th Annual Symposium on Foundations of Computer Science (FOCS)*. Santa Fe, NM, USA: IEEE Computer Society Press, 1994, pp. 124–134. DOI: 10.1109/SFCS.1994.365700.

[7] Kevin Townsend. *Solving the Quantum Decryption "Harvest Now, Decrypt Later" Problem*. SecurityWeek. Feb. 16, 2022. URL: https://www.securityweek.com/solving-quantum-decryption-harvest-now-decrypt-later-problem/.

[8] Michele Mosca. "Cybersecurity in an Era with Quantum Computers: Will We Be Ready?" In: *IEEE Security & Privacy* 16.5 (2018), pp. 38–41.

[9] Marie A Wright. "The advanced encryption standard". In: *Network Security* 2001.10 (2001), pp. 11–13.

[10] James Nechvatal et al. "Report on the development of the Advanced Encryption Standard (AES)". In: *Journal of research of the National Institute of Standards and Technology* 106.3 (2001), p. 511.

[11] National Institute of Standards and Technology. *Secure Hash Standard (SHS)*. FIPS PUB 180-4. 2015.

[12] Lov K. Grover. "A Fast Quantum Mechanical Algorithm for Database Search". In: *Proceedings of the 28th Annual ACM Symposium on Theory of Computing*. 1996, pp. 212–219.

[13] Lily Chen et al. *Report on Post-Quantum Cryptography*. Tech. rep. NIST IR 8105. National Institute of Standards and Technology, 2016.

[14] Daniel J. Bernstein, Johannes Buchmann, and Erik Dahmen. "Post-Quantum Cryptography". In: *Springer* (2009).

[15] Stanford University. *Quantum Cryptography*. Modern Cryptography: Theory and Applications. 2005. URL: https://cs.stanford.edu/people/eroberts/courses/soco/projects/2004-05/cryptography/quantum.html.

[16] Cankun Zhao et al. "A Compact and High-Performance Hardware Architecture for CRYSTALS-Dilithium". In: *IACR Transactions on Cryptographic Hardware and Embedded Systems* 2022.1 (Nov. 2021), pp. 270–295. DOI: 10.46586/tches.v2022.i1.270-295. URL: https://tches.iacr.org/index.php/TCHES/article/view/9297.

[17] Pierre-Alain Fouque et al. *Falcon: Fast-Fourier Lattice-based Compact Signatures over NTRU. Specifications v1.2*. Tech. rep. NIST Post-Quantum Cryptography Project, 2020. URL: https://falcon-sign.info/falcon.pdf.

[18] Daniel J. Bernstein et al. "The SPHINCS+ Signature Framework". In: *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security (CCS)*. ACM, Nov. 2019. DOI: 10.1145/3319535.3363229.

[19] Andrew Waterman and Krste Asanović. *The RISC-V Instruction Set Manual, Volume I: User-Level ISA*. Tech. rep. RISC-V International, 2019. URL: https://riscv.org/technical/specifications/.

[20] Simone Machetti et al. *X-HEEP: An Open-Source, Configurable and Extendible RISC-V Microcontroller for the Exploration of Ultra-Low-Power Edge Accelerators*. 2024. arXiv: 2401.05548 [cs.AR].

[21] Luke Beckwith et al. "A High-Performance Hardware Implementation of LESS Digital Signature Scheme". In: *Post-Quantum Cryptography: 14th International Workshop, PQCrypto 2023, College Park, MD, USA, August 16–18, 2023, Proceedings*. College Park, MD, USA: Springer-Verlag, 2023, pp. 57–90. ISBN: 978-3-031-40002-5. DOI: 10.1007/978-3-031-40003-2_3. URL: https://doi.org/10.1007/978-3-031-40003-2_3.

[22] Edoardo Persichetti and Paolo Santini. *A New Formulation of the Linear Equivalence Problem and Shorter LESS Signatures*. Cryptology ePrint Archive, Paper 2023/847. 2023. URL: https://eprint.iacr.org/2023/847.

[23] Jean-François Biasse et al. "LESS is More: Code-Based Signatures Without Syndromes". In: *Progress in Cryptology - AFRICACRYPT 2020*. Ed. by Abderrahmane Nitaj and Amr Youssef. Cham: Springer International Publishing, 2020, pp. 45–65. ISBN: 978-3-030-51938-4.

[24] Alessandro Barenghi et al. "LESS-FM: Fine-Tuning Signatures from the Code Equivalence Problem". In: *Post-Quantum Cryptography*. Ed. by Jung Hee Cheon and Jean-Pierre Tillich. Cham: Springer International Publishing, 2021, pp. 23–43. ISBN: 978-3-030-81293-5.

[25] Tung Chou, Edoardo Persichetti, and Paolo Santini. *On Linear Equivalence, Canonical Forms, and Digital Signatures*. Cryptology ePrint Archive, Paper 2023/1533. 2023. URL: https://eprint.iacr.org/2023/1533.

[26] Marco Baldi et al. *LESS: Linear Equivalence Signature Scheme*. Tech. rep. Official specification, version February 7, 2025. LESS Project, Feb. 2025. URL: https://www.less-project.com/LESS-2025-02-07.pdf.

[27] Amos Fiat and Adi Shamir. "How To Prove Yourself: Practical Solutions to Identification and Signature Problems". In: *Advances in Cryptology — CRYPTO' 86*. Ed. by Andrew M. Odlyzko. Berlin, Heidelberg: Springer Berlin Heidelberg, 1987, pp. 186–194. ISBN: 978-3-540-47721-1.

[28] Thomas Attema and Ronald Cramer. *Compressed Σ-Protocol Theory and Practical Application to Plug Play Secure Algorithmics*. Cryptology ePrint Archive, Paper 2020/152. 2020. URL: https://eprint.iacr.org/2020/152.

[29] LESS Team. *Repository for the Current Status of the LESS Submission*. https://github.com/less-sig/LESS. Commit c575939. 2025.

[30] Jerome Solinas. "Pseudo-Mersenne Prime". In: *Encyclopedia of Cryptography and Security*. Ed. by Henk C. A. van Tilborg. Boston, MA: Springer US, 2005, pp. 482–483. ISBN: 978-0-387-23483-0. DOI: 10.1007/0-387-23483-7_325. URL: https://doi.org/10.1007/0-387-23483-7_325.

[31] OpenTitan Project. *OpenTitan Documentation*. https://opentitan.org/documentation/index.html. 2026.

[32] Patrick Karl et al. *High-Performance FPGA Accelerator for the Post-quantum Signature Scheme CROSS*. Cryptology ePrint Archive, Paper 2025/1161. 2025. URL: https://eprint.iacr.org/2025/1161.

Acknowledgments

I would like to express first and foremost my deepest gratitude to Prof. Guido Masera for giving me the opportunity to work on this research and to be part of this team. His support and guidance allowed me to explore and learn more about this fascinating topic.

I am especially grateful to Ph.D. Alessandra Dolmeta and Valeria Piscopo, who closely followed me throughout this work. Their continuous support, patience, and constant availability were fundamental during every stage of the project. I am truly thankful for their valuable advice, constructive feedback, and for always taking the time to accommodate my needs or requests. Their guidance not only helped me complete this thesis, but also allowed me to learn a great deal and grow both professionally and personally.

I would like to thank my parents, Claudio and Simona, my brother Gaetano, and my entire family for always being by my side and for their unconditional support and sacrifices. Your presence has always been a source of strength and comfort, even during the most challenging moments.

I would like to dedicate a special thanks to my grandparents, Gaetano and Maria Concetta. They have always believed in me and celebrated every success as if it were their own. Their strength, their kindness, and the values they have passed on to me have shaped the person I am today.

A heartfelt thank you goes to my uncle Mimmo, who has been a role model since I was a child, for his constant encouragement and for always managing to make me smile.

Finally, I would like to thank all my friends, both near and far, Italian and international, for the many moments we shared together that made these years truly unforgettable. Your support, laughter, and companionship made this journey much more enjoyable and meaningful.

This milestone belongs to you as much as it belongs to me.
Thank you all.