Unified MEDS Accelerator

Sanjay Deshpande[1], Yongseok Lee[2], Mamuri Nawan[3], Kashif Nawaz[3], Ruben Niederhagen[4,5], Yunheung Paek[2], and Jakub Szefer[6]

[1] CASLAB, DEE, Yale University, New Haven, US,
sanjay.deshpande@yale.edu
[2] ECE, Seoul National University, Seoul, South Korea,
yslee@sor.snu.ac.kr,ypaek@snu.ac.kr
[3] CRC, Technology Innovation Institute, Abu Dhabi, UAE,
mamuri@tii.ae, kashif.nawaz@tii.ae
[4] IIS, Academia Sinica, Taipei, Taiwan,
ruben@polycephaly.org
[5] IMADA, University of Southern Denmark, Odense, Denmark
[6] Northwestern University, Evanston, US jakub.szefer@northwestern.edu

Abstract. The Matrix Equivalence Digital Signature (MEDS) scheme, a code-based candidate in the first round of NIST's Post-Quantum Cryptography (PQC) standardization process, offers competitively small signature sizes but incurs high computational costs for signing and verification. This work explores how a high-performance FPGA-based hardware implementation can enhance MEDS performance by leveraging the inherent parallelism of its computations, while examining the trade-offs between performance gains and resource costs. This work in particular proposes a unified hardware architecture capable of efficiently performing both signing and verification operations within a single combined design. The architecture jointly supports all security parameters, including the dynamic, run-time handling of different prime fields without the need to re-configure the FPGA. This work also evaluates the resource overhead of supporting different prime fields in a single design, which is relevant not only for MEDS but also for other cryptographic schemes requiring similar flexibility. This work demonstrates that custom hardware for PQC signature schemes can flexibly support different prime fields with limited resource overhead. For example, for NIST security Level I, our implementation achieves signing times of 4.5 ms to 65.2 ms and verification times of 4.2 ms to 64.5 ms utilizing 22k to 72k LUTs and 66 to 273 DSPs depending on design variant and optimization goal.

Keywords: Post-Quantum Cryptography · Digital Signature Algorithm · Matrix Equivalence Digital Signature (MEDS)

1 Introduction

The advent of quantum computing poses significant threats to classical cryptography, challenging the security foundations of many widely-used cryptographic algorithms. Quantum computers have the potential to solve specific complex mathematical problems much faster than classical computers. When a sufficiently large, error-corrected quantum computer is available, quantum computer algorithms such as Shor's algorithm can break widely-used public-key cryptosystems such as Rivest–Shamir–Adleman (RSA), Diffie-Hellman (DH), and Elliptic Curve Cryptography (ECC). To address these emerging threats, the field of Post-Quantum Cryptography (PQC) has emerged, focusing on the design and implementation of cryptographic algorithms that remain secure even in the presence of quantum computing power.

While significant progress has been made in developing theoretically secure PQC algorithms, practical deployment in real-world scenarios also requires a thorough understanding of their performance and resource requirements. Efficient implementations are critical for ensuring that these algorithms can be adopted at scale without incurring prohibitive computational costs.

This work contributes to this area by presenting a hardware implementation and evaluation of the Matrix Equivalence Digital Signature (MEDS) scheme. MEDS is a code-based digital signature scheme based on the hardness assumption of the Matrix Code Equivalence (MCE) problem. It was submitted to the National Institute of Standards and Technology (NIST) PQC Signature standardization process launched in 2023, but it did not advance to the second round due to its comparatively low performance and the novelty of its security assumptions. Given the importance of ensuring the practicality of the cryptographic schemes, this paper focuses on developing a unified high-performance hardware implementation of the MEDS signing and verification operation. We present a joint design that supports all security parameter sets, selectable at runtime, and evaluate the overhead of accommodating arithmetic for different finite fields in a single design. Our results on the overhead generalize well and can be transferred to other schemes with similar parameter set specifications.

By addressing the computational challenges associated with MEDS, we aim to contribute to the broader goal of making post-quantum cryptography viable for widespread adoption in the face of advancing quantum computing technologies. To the best of our knowledge, this work presents the first hardware implementation of MEDS. Our implementation operates in constant time (i.e., there are no timing variations depending on secret data), though it does not include additional side-channel protections.

Related Work. Gaussian elimination is an important step in MEDS as well as many other PQC schemes. Early work [HQR89] used processor arrays as large as the matrix being processed. The work in [SWM$^+$10] presented one of the first hardware accelerators for the code-based Niederreiter cryptosystem. The work used a systolic processor array for performing Gaussian systemization. Later work on Classic McEliece also used a systemizer similar to [SWM$^+$10] with smaller processor array. For example, in [WSN16,WSN17] the authors present a hardware design of a key-generation module for the Niederreiter cryptosystem. In [CCD$^+$22], the authors implemented a complete design of Classic McEliece compliant with the specification submitted to NIST standardization process including binary field systemizers. We are using a similar systemizer in our work.

Although there are similar computations in regard to matrix systemization between MEDS and Classic McEliece, compared to Classic McEliece, instead of working with binary fields, MEDS requires $\mathbb{F}_q$ with prime q. As a result, the latencies of finite field operations are larger compared to when using $\mathbb{F}_2$ or a small binary extension filed $\mathbb{F}_q^m$. This difference necessities significant design effort for the systemizer design, e.g., a different control logic and a more complex pipelining process.

Within current NIST standardization process for digital signatures schemes, in [SMA+24,HSK+23] the authors develop hardware implementations of the MAYO digital signature scheme. In [DHSY24], authors present the first hardware implementation of the SDitH digital signature scheme. [dPRS23] describes hardware FPGA implementations of the Raccoon digital signature scheme.

In [WJW+19], the authors present several hardware accelerators that work with a RISC-V core to accelerate the XMSS signature scheme. In [BCH+23], the authors present hardware implementation of the Oil and Vinegar (OV) signature scheme. In [TYD+11] authors presented an FPGA-based implementation of the multivariate-based signature scheme Rainbow. Later, in [FG18] authors presented high-speed, FPGA-based implementation of Rainbow with updated parameters for NIST's first round of PQC standardization process. Both designs are operating on small binary fields using a processor array similar to the code-based designs discussed above. Among ASIC designs, in [ZZL+23] authors present a processor designed specifically for PQC algorithms, which can support schemes such as Saber, Kyber, Dilithium, NTRU, McEliece, and Rainbow. This design also includes a module for the systemization of matrices over small binary fields, but instead of a processor array composed of several vector units, the authors here are using only a single a vector unit.

Hardware architectures for algorithms already being standardized by NIST include [SAW+23], where the authors implement a hardware design of the FALCON scheme and [LSG21,BNG21] where the authors explore hardware implementations of CRYSTALS-Dilithium (being standardized under the name "ML-DSA"). [ALCZ20,Saa24,DLK+25] explores implementations of SPHINCS+ (being standardized under the name "SLH-DSA").

We are not aware of any other hardware implementation of MEDS and our work presents a hardware design of MEDS that provides a competitive performance and resource utilization compared hardware implementations of other signature schemes.

Contributions. This paper introduces the first hardware implementation of the sign and verify operations for the PQC signature scheme MEDS. Our implementation combines both operations into a single design sharing most resources between sign and verify. We further provide joint support of all security parameter sets in one single design selectable at runtime.

The specification of MEDS introduces several challenges for its implementation:

C1: MEDS operates on dense matrices of a finite fields, demanding significant computational resources. One of the reasons MEDS was not admitted to the second round of NIST's standardization process is its longer runtime compared to competing PQC schemes.

C2: Due to the Fiat-Shamir (FS) construction of MEDS, the signing and verification operations share significant structural similarities, which lend themselves for resource optimization.

C3: To optimize signature sizes, MEDS parameter sets involve two distinct finite fields.

Given these challenges we aim to address the following research questions:

Q1: Does MEDS provide sufficient inherent parallelism to accelerate the sign and verify operations to a competitive level? What are the associated resource costs for achieving such a speed-up?

Q2: To what extent can resources be shared in a combined design for the sign and verify operations?

Q3: What is the overhead of supporting all security parameters at runtime, particularly concerning arithmetic in two distinct finite fields?

Structure of this paper. We introduce our notation and the MEDS PQC signature scheme in Section 2. The top-down design of our hardware implementation is introduced in Section 3 followed by a bottom-up description of the modules and building blocks in Section 4. Section 5 presents evaluation results and Section 6 concludes this work.

2 Preliminaries

We first introduce our notation in Section 2.1 and then explain the relevant details of the MEDS specification in Section 2.2.

2.1 Notation

We are following the notation of the MEDS specification document [CNP$^+$23a] and denote matrices with bold capital letters, e.g., $\mathbf{M}$, $\mathbf{A}$, and $\mathbf{B}$. We follow the MEDS specification in denoting submatrices with square brackets, e.g., $\mathbf{M}[a, b; c, d]$ denotes the submatrix of the intersection of rows a to b and columns c to d of matrix $\mathbf{M}$ [CNP$^+$23a, Section 2.1]. If no row or column range is provided, all rows or columns are included. $\mathbb{F}_q$ is a finite field with q elements. We define an $[m \times n, k]$ matrix rank metric code over $\mathbb{F}_q$ as k-dimensional subspace of $\mathbb{F}_q^{m\times n}$. Here, m and n are the codeword sizes and k the code dimension.

The operation $\mathsf{SF}(\mathbf{M})$ returns the systematic form of a matrix $\mathbf{M}$ if it exists or $\perp$ if not. The operation $\pi_{\mathbf{A},\mathbf{B}}(\mathbf{M})$ with $\mathbf{A} \in \mathbb{F}_q^{m\times m}$, $\mathbf{B} \in \mathbb{F}_q^{n\times n}$, and $\mathbf{M} \in \mathbb{F}_q^{k\times mn}$ first maps each row $i \in \{0, \dots, k-1\}$ of $\mathbf{M}$ to a matrix $\mathbf{P}_i \in \mathbb{F}_q^{m\times n}$ such that $\mathbf{P}_i[\lfloor j/n \rfloor; j \bmod n] = \mathbf{M}[i, j]$ for $j \in \{0, \dots, mn-1\}$. It then computes $\mathbf{P}'_i \in \mathbb{F}_q^{m\times n}$ as $\mathbf{P}'_i = \mathbf{A}\mathbf{P}_i\mathbf{B}$, maps the $\mathbf{P}'_i$ back to the rows i of a matrix $\mathbf{M}' \in \mathbb{F}_q^{k\times mn}$, and finally returns $\mathbf{M}'$ as result.

2.2 MEDS

The MEDS scheme [CNP$^+$23a,CNP$^+$23b] was one of the code-based submissions to the on-ramp to the NIST PQC signature standardization process[7] but it did not advance to the second round. It is based on the notion of Matrix Code Equivalence (MCE):

Definition 1 (Matrix Code Equivalence). *Let* $\mathcal{C}$ *and* D *be two* $[m \times n, k]$ *matrix codes over* $\mathbb{F}_q$*. We say that* C *and* $\mathcal{D}$ *are* equivalent *if there exist two invertible matrices* $\mathbf{A} \in \mathbb{F}_q^{m\times m}$ *and* $\mathbf{B} \in \mathbb{F}_q^{n\times n}$ *such that* $\mathcal{D} = \mathbf{A} \cdot \mathcal{C} \cdot \mathbf{B}$*, i.e., for all* $\mathbf{C} \in \mathcal{C},\ \mathbf{A} \cdot \mathbf{C} \cdot \mathbf{B} \in \mathcal{D}$*.*

This gives raise to the MCE problem:

Problem 1 (Matrix Code Equivalence Problem). Given two k-dimensional matrix codes $\mathcal{C}, \mathcal{D} \subset \mathbb{F}_q^{m\times n}$, find two invertible matrices $\mathbf{A} \in \mathbb{F}_q^{m\times m},\ \mathbf{B} \in \mathbb{F}_q^{n\times n}$ such that $\mathcal{D} = \mathbf{A} \cdot \mathcal{C} \cdot \mathbf{B}$.

The MCE problem is at least as hard as the linear code equivalence problem and as hard as the isomorphism of polynomials problem [BFV13], and as hard as the alternating trilinear form equivalence problem [GQT21,TDJ$^+$22]. Please refer to [CNP$^+$23a,CNP$^+$23b] for a concrete security analysis.

MEDS uses the MCE problem to construct a signature scheme from an interactive Σ-protocol using the FS transform [FS87] with t rounds and by applying some tricks to improve signature size and performance, i.e., by using s matrix codes in the public key, by using challenges with a fixed weight w, by seeding 0-responses, and by generating these seeds from a seed tree.

MEDS parameters. Table 1 shows the parameter sets of MEDS from the first round of the NIST PQC signature scheme standardization. In May 2024, the MEDS submission team announced new parameter sets in the NIST PQC mailing list[8] as reaction to refined attacks [NQT24]. However, they suggest to combine these new parameter sets with signature-size optimization techniques introduced in [CNRS24]. These optimizations require algorithmic changes compared to the MEDS Round 1 specification document [CNP$^+$23a]. We decided to provide a hardware implementation that follows the MEDS Round 1 specification and therefore do not implement the optimization from [CNRS24] since there is no concrete specification for that MEDS variant and since we are interested in analyzing the effect of supporting different prime fields in a joint design.

MEDS key generation. To generate a key pair, first chose a random $\mathbf{G}_0 \in \mathbb{F}_q^{k\times mn}$. Then choose random invertible matrices $\mathbf{A}_1, \ldots, \mathbf{A}_{s-1} \in \mathbb{F}_q^{m\times m}$ and $\mathbf{B}_1, \ldots, \mathbf{B}_{s-1} \in \mathbb{F}_q^{n\times n}$. Finally, compute the matrices $\mathbf{G}_1, \ldots, \mathbf{G}_{s-1} \in \mathbb{F}_q^{k\times mn}$ as $\mathbf{G}_i = \mathsf{SF}(\pi_{\mathbf{A}_i,\mathbf{B}_i}(\mathbf{G}_0)),\ i \in \{1, \ldots, s-1\}$. (MEDS reduces the public key size

[7] `https://csrc.nist.gov/Projects/pqc-dig-sig/round-1-additional-signatures`

[8] `https://groups.google.com/a/list.nist.gov/g/pqc-forum/c/pbT_DnPrc2A/m/ZPrIVSmFCQAJ`

Table 1: MEDS parameter sets.

Level	Parameter Set	q	n	m	k	s	t	w	pk (byte)	sig (byte)
MEDS Parameter Sets [CNP+23a, Table 2]:										
Level I	MEDS-9923	4093	14	14	14	4	1152	14	9923	9896
Level I	MEDS-13220	4093	14	14	14	5	192	20	13 220	12 976
Level III	MEDS-41711	4093	22	22	22	4	608	26	41 711	41 080
Level III	MEDS-69497	4093	22	22	22	5	160	36	55 604	54 736
Level V	MEDS-134180	2039	30	30	30	5	192	52	134 180	132 528
Level V	MEDS-167717	2039	30	30	30	6	112	66	167 717	165 464
New Parameter Sets May 2024 (signature bytes with seed tree)[8]:										
Level I		4093	26	25	25	2	144	48	21 595	5456
Level III		4093	35	34	34	2	208	75	55 520	10 786
Level V		4093	45	44	44	2	272	103	122 000	21 052

slightly by solving linear systems to obtain the first two rows of the $\mathbf{G}_i,\ i > 0$ as well known, constant vectors.) The public key of MEDS consists of the seed for randomly generating $\mathbf{G}_0 \in \mathbb{F}_q^{k\times mn}$ and $s-1$ matrices $\mathbf{G}_1, \dots, \mathbf{G}_{s-1} \in \mathbb{F}_q^{k\times mn}$. The secret key of $s-1$ pairs of invertible matrices $(\mathbf{A}_1^{-1}, \mathbf{B}_1^{-1}), \dots, (\mathbf{A}_{s-1}^{-1}, \mathbf{B}_{s-1}^{-1}) \in \mathbb{F}_q^{m\times m} \times \mathbb{F}_q^{n\times n}$.

MEDS signing. During signing, the signer commits to t matrices $\tilde{\mathbf{G}}_i \in \mathbb{F}_q^{k\times mn},\ i \in \{0, \dots, t-1\}$ using t maps of pairs of random invertible matrices $(\tilde{\mathbf{A}}_0, \tilde{\mathbf{B}}_0), \dots, (\tilde{\mathbf{A}}_{t-1}, \tilde{\mathbf{B}}_{t-1}) \in \mathbb{F}_q^{m\times m} \times \mathbb{F}_q^{n\times n}$ such that $\tilde{\mathbf{G}}_i = \mathsf{SF}(\pi_{\tilde{\mathbf{A}}_i, \tilde{\mathbf{B}}_i}(\tilde{\mathbf{G}}_0)),\ i \in \{0, \dots, t-1\}$. The signer then hashes the $\tilde{\mathbf{G}}_i$ to the commitment hash d and parses d to the challenge vector $h_0, \dots, h_{t-1}$ of weight w with $h_i \in \{0, s-1\}$. The signature finally is composed by providing the commitment hash d as well as the seeds $\tilde{\mathbf{A}}_i$ and $\tilde{\mathbf{B}}_i$ for all i where $h_i = 0$ and $\mu_i = \tilde{\mathbf{A}}_i \cdot \mathbf{A}_{h_i}^{-1}$ and $\nu_i = \mathbf{A}_{h_i}^{-1} \cdot \tilde{\mathbf{B}}_i$ for all i where $h_i > 0$.

MEDS verification. For verification, the verifier parses the challenge vector $h_0, \dots, h_{t-1}$ from the commitment hash d. The verifier then computes $\hat{\mathbf{G}}_i \in \mathbb{F}_q^{k\times mn},\ i \in \{0, \dots, t-1\}$ as $\hat{\mathbf{G}}_i = \mathsf{SF}(\pi_{\mu_i, \nu_i}(\mathbf{G}_{h_i}))$, where the matrices $\mu_i = \tilde{\mathbf{A}}_i$ and $\nu_i = \tilde{\mathbf{B}}_i$ are regenerated from the seed-tree seeds for all i with $h_i = 0$, and $\mu_i = \tilde{\mathbf{A}}_i \cdot \mathbf{A}_{h_i}^{-1}$ and $\nu_i = \mathbf{A}_{h_i}^{-1} \cdot \tilde{\mathbf{B}}_i$ are parsed from the signature for all i with $h_i > 0$. If the hash of the $\hat{\mathbf{G}}_i$ equals the commitment hash d, the verification is successful.

Cost. The most expensive sub-operations in MEDS are SF with a cost of $\mathrm{O}(mnk^2)$ finite field operations for performing Gaussian elimination on a matrix of size $k \times mn$ and π with a cost of $\mathrm{O}(kmn^2 + km^2n)$ for performing k matrix products of sizes $\mathbb{F}_q^{m\times m}$ times $\mathbb{F}_q^{m\times n}$ times $\mathbb{F}_q^{n\times n}$. Hence, the computational cost of SF and π is quartic in the security parameters with $k \approx m \approx n$.

For key generation, we need to compute SF and π each only s times, where $s \in \{4, 5, 6\}$ — but for signing and verification, we need to perform these costly operations t times with $t \in \{112, 160, 192, 608, 1152\}$. Therefore, signing and verification are 18.6 to 288 times more expensive than key generation and hence benefit the most from hardware acceleration. Since verification performs almost the same computations as signing with only little difference in the control flow, we combine both operations in a single design, sharing as many resources between these operations as possible.

3 Unified MEDS Design

We call our hardware design of MEDS a *unified* design, since it:

1. *Combines* both the MEDS signing and verification operations into a single design,

and it provides

2. *Joint* support for all parameter sets selectable at runtime,

while sharing hardware resources between both operations and all parameter sets as much as possible.

Algorithm 1 and Algorithm 2 show the signing and verification operations in detail [CNP$^+$23a, Algorithms 11 and 12]. The inputs to the signing operation are the secret key sk and the message msg that needs to be signed. The inputs to the verify operation are the public key pk and the signed message to be verified.

MEDS signing in detail. During signing, line 1 to line 10 in Algorithm 1 parse the seed of public matrix $\mathbf{G}_0 \in \mathbb{F}_q^{k \times mn}$ and the s pairs of secret matrices $\mathbf{A}_i^{-1} \in \mathbb{F}_q^{m \times m}$ and $\mathbf{B}_i^{-1} \in \mathbb{F}_q^{n \times n}$, $0 \leq i \leq s-1$, from the secret key sk. The function $\mathsf{ExpandSystMat}$ generates $\mathbf{G}_0$ from the seed $\sigma_{\mathbf{G}_0}$ (stored in the secret key) and the function $\mathsf{Decompress}$ converts the packed byte-representation of the matrices in the secret key to $\mathbf{A}_i^{-1}$ and $\mathbf{B}_i^{-1}$.

Line 12 initializes the root seed ρ and the salt α of the seed tree from the random seed δ (obtained by calling $\mathsf{Randombytes}$ in line 11) using the extendible output function XOF based on SHAKE256. (XOF takes the seed δ and the required output lengths $\ell_{\mathsf{tree\_seed}}$ and ℓ_{salt} as input and returns a tuple of $\ell_{\mathsf{tree\_seed}}$ and ℓ_{salt} bytes used as seeds ρ and α.) ρ and α are input to the $\mathsf{SeedTree}$ function in line 13 that generates the t leaf seeds of the seed tree.

The commitments $\tilde{\mathbf{G}}_i$, $0 \leq i \leq t-1$, are computed in the for-loop over i from line 15 to line 18. The matrices $\tilde{\mathbf{A}}_i \in \mathbb{F}_q^{m \times m}$ and $\tilde{\mathbf{B}}_i \in \mathbb{F}_q^{n \times n}$ are generated using the function $\mathsf{ExpandInvMat}$ from seeds $\sigma_{\tilde{\mathbf{A}}}$ and $\sigma_{\tilde{\mathbf{B}}}$ (which are obtained by adding salt and address to the leaf seed σ_i of the seed tree as multi-target attack countermeasure). The map π is computed in line 19 and the final $\tilde{\mathbf{G}}_i$ as the systematic form using the function SF in line 20. With some low probability (see Section 4.4), the systemization in line 20 might fail. In that case, the process for the affected iteration i needs to be repeated starting in line 15.

Once the $\tilde{\mathbf{G}}_i, 0 \leq i \leq t-1$ have been computed, they are hashed together with the message msg into the commitment hash d in line 23. This hash d is then parsed into the weight-w vector $(h_0, \ldots, h_{t-1}) \in \{0, \ldots, s-1\}^t$ using function ParseHash in line 24.

Then, the responses for the cases of $h_i \neq 0$ are computed in the loop from line 26 to line 31 by computing $\mu_i = \tilde{\mathbf{A}}_i \cdot \mathbf{A}_{h_i}^{-1}$ and $\nu_i = \mathbf{B}_{\mathbf{h_i}}^{\mathbf{-1}} \cdot \tilde{\mathbf{B}}_i$ and by storing them as packed bytes using the function Compress. Finally, the path in the seed tree is constructed from $h_0, \ldots, h_{t-1}$ using the function ParseHash and the signed message $\mathsf{msg_{sig}}$ is returned as the w matrices μ_i and ν_i for $h_i \neq 0$ compressed to packed byte strings $v_0, \ldots, v_{w-1}$, the seed tree path p, the commitment hash d, the salt α, and the message msg.

MEDS verify in detail. For verification, line 1 to line 6 in Algorithm 2 re-generate $\mathbf{G}_0$ from its seed (line 2) and parse the public key matrices $\mathbf{G}_i \in \mathbb{F}_q^{k \times mn}$, $1 \leq i \leq s-1$, from the public key pk. Line 7 and line 8 parse the seed tree path p, commitment hash d, the salt α, and the message msg from the signed message $\mathsf{msg_{sig}}$. The commitment hash d is then parsed into the challenge string $h_0, \ldots, h_{t-1}$ and the seeds $\{\sigma_i \mid 0 \leq i \leq t-1,\ h_i = 0\}$ are recovered from the partial seed tree.

The commitments $\hat{\mathbf{G}}_i$, $0 \leq i \leq t-1$, are then recovered in the for-loop over i from line 12 to line 27. If $h_i = 0$, the matrices $\mu_i \in \mathbb{F}_q^{m \times m}$ and $\nu_i \in \mathbb{F}_q^{n \times n}$ are re-generated from seeds derived from σ_i (which are obtained by adding salt and address; if $h_i > 0$, they are parsed from the signature $\mathsf{msg_{sig}}$. The map π is computed in line 24 and the final $\hat{\mathbf{G}}_i$ as the systematic form using the function SF in line 25.

Finally, the matrices $\hat{\mathbf{G}}_i, 0 \leq i \leq t-1$ are serialized and hashed together with the message msg to recompute the commitment hash d' in line 28. If $d = d'$, the signature verified successfully and the message msg is returned in line 30. Otherwise, verification has failed and an error is returned in line 32.

Comparing the main loop in line 14 to line 22 in Algorithm 1 with the main loop in line 12 to line 27 in Algorithm 2 shows that the main operations of obtaining invertible matrices, applying the π and SF operations, and computing a commitment hash are structurally identical. Hence, in our combined design, resources for performing these computations can be shared efficiently between the sign and the verify operation with little control overhead.

3.1 Top-down Overview of Our MEDS Hardware Design

Figure 1 shows the operation flow of our combined hardware implementation. Data flow unique to the sign operation is shown in red and unique to the verify operation in blue. First, both operations expand $\mathbf{G}_0$ from seed $\sigma_{\mathbf{G}_0}$, which is parsed from sk or pk respectively, using the module `ExpandSystMat`.

Signature generation starts with the generation of the seed δ of length $\ell_{\mathsf{sec\_seed}}$ from a randomness source. We assume that the seed δ will be initialized by a top-level hardware module, e.g., by interfacing to a True Random Number Generator (TRNG) or by implementing the NIST Known Answer Test (KAT)

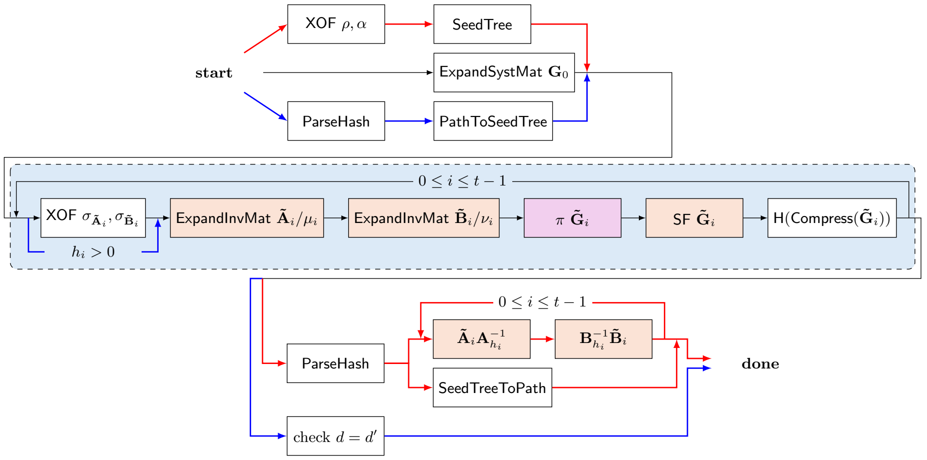

XOF ρ, α
SeedTree
start
ExpandSystMat $\mathbf{G}_0$
ParseHash
PathToSeedTree
$0 \leq i \leq t-1$
XOF $\sigma_{\tilde{\mathbf{A}}_i}, \sigma_{\tilde{\mathbf{B}}_i}$
ExpandInvMat $\tilde{\mathbf{A}}_i / \mu_i$
ExpandInvMat $\tilde{\mathbf{B}}_i / \nu_i$
π $\tilde{\mathbf{G}}_i$
SF $\tilde{\mathbf{G}}_i$
H(Compress($\tilde{\mathbf{G}}_i$))
$h_i > 0$
$0 \leq i \leq t-1$
ParseHash
$\tilde{\mathbf{A}}_i \mathbf{A}_{h_i}^{-1}$
$\mathbf{B}_{h_i}^{-1} \tilde{\mathbf{B}}_i$
SeedTreeToPath
done
check $d = d'$

Fig. 1: Overview of the operational flow of our combined sign and verify module, red lines for sign only, blue lines for verify only.

Pseudo Random Number Generator (PRNG). The seed δ is then expanded using the module XOF to the root seed ρ of length $\ell_{\mathsf{tree\_seed}}$ of the seed tree and a salt α of length ℓ_{salt}. In our hardware design, we accomplish this again using a SHAKE256 module via a XOF wrapper. Then the next step is the seed tree generation using the root seed ρ and the salt α with the SeedTree module. Verification instead parses the hash and then generates the partial seed tree from the seed tree path in the signature. We implement these operations in the modules ParseHash and PathToSeedTree.

For the main loop in sign and verify we need to expand the seeds σ_i from the seed tree leaves into seeds $\sigma_{\tilde{\mathbf{A}}_i}$ and $\sigma_{\tilde{\mathbf{B}}_i}$ and expand these seeds to $\tilde{\mathbf{A}}_i$ and $\tilde{\mathbf{B}}_i$ (i.e., μ_i and ν_i respectively for verify) using a ExpandInvMat module. For signing, we store each σ_i in a memory which is later used to recompute $\sigma_{\tilde{\mathbf{A}}_i}$ and $\sigma_{\tilde{\mathbf{B}}_i}$ and then expand these seeds to $\tilde{\mathbf{A}}_i$ and $\tilde{\mathbf{B}}_i$ for computation of μ_i and ν_i. (Expanding the seeds is skipped in verify for $h_i > 0$ when instead μ_i and ν_i are parsed from the signature. In this case, we use the systemizer in the ExpandInvMat module to verify that μ_i and ν_i are invertible.) We then feed the matrices into a Pi module together with the matrix $\mathbf{G}_0$, compute the systematic form $\tilde{\mathbf{G}}_i$ of the result using the module SF. We pull the absorb-function of the hash function H into the loop to avoid the need to store all $\tilde{\mathbf{G}}_i$. We will describe the implementation of this loop in detail in Section 4.10.

The steps after the main loop are different for sign and verify. For sign, we also absorb the message msg into the hash state and finalize H to obtain the challenge vector. Then, to compute the final loop, we recompute $\sigma_{\tilde{\mathbf{A}}_i}$ and $\sigma_{\tilde{\mathbf{B}}_i}$ using σ_i, and then expand them to generate $\tilde{\mathbf{A}}_i$ and $\tilde{\mathbf{B}}_i$ for the computation of μ_i and ν_i using a matrix-matrix multiplication module. Eventually, we compute the seed tree path for the signature using SeedTreeToPath. For verify, we only need to check that $d = d'$ and set the return value of the module correspondingly.

Therefore, we implement the hardware modules `ExpandSystMat`, `SHAKE256`, `XOF`, `SeedTree`, `ExpandInvMat`, `Pi`, `SF`, `CompressH`, `ParseHash`, `MatMul`, and a combined module for `PathToSeedTree` and `SeedTreeToPath` in our hardware design. The modules are also listed in Figure 1, except for `SHAKE256` that is used internally by other modules.

3.2 Joint Security Parameter Design

One of the main questions we want to investigate in this work is the hardware overhead of supporting several prime fields for different security parameter sets selectable at runtime in one joint hardware design. For modules that require both 11-bit arithmetic in $\mathbb{F}_{2039}$ and 12-bit arithmetic in $\mathbb{F}_{4093}$, we are using an input signal `i_sel_4093` that is pulled high for $\mathbb{F}_{4093}$ and low for $\mathbb{F}_{2039}$. The bus width for field elements is set to 12 bit for the joint design, with the most significant bit set to zero when $\mathbb{F}_{2039}$ is being selected. Therefore, the cost for memory storing field elements is defined by the larger field in the joint design. A similar design approach of constructing field arithmetic modules capable of operating over two distinct prime fields was adopted in [AMI^{+}23,MR24], where the authors integrated CRYSTALS-Kyber (which operates over a 12-bit prime field polynomial coefficients) and CRYSTALS-Dilithium (which utilizes a 23-bit prime field polynomial coefficients). In contrast, MEDS features prime fields that differ by only a single bit, enabling support for multiple fields with minimal overhead.

If the Verilog code is synthesised for one specific field only, we are fixing `i_sel_4093` to constant 0 or 1 respectively so that the optimization step during synthesis can remove unnecessary logic automatically. This enables us to get precised resource estimates for hardware designs supporting only one respective field and for the joint design with otherwise identical logic.

The remaining security parameters besides the prime field are matrix dimensions n, m, and k, the number s of pk matrices, as well as t and s for the challenge vector. All these other parameters mostly affect the upper limit of loops and can easily be selected using multiplexers at runtime at little resource overhead. We pre-compute related pre-defined constant values (e.g., counter widths) and provide them as macros and module parameters during synthesis (and simulation).

4 Implementation

For the description of our implementation of the MEDS signature scheme, we take a bottom-up approach. The different buildong blocks and sub-modules are described in the following including finite field arithmetic, matrix operations, hashing and seed expansion, the seed tree and eventually the complete sign and verify module.

4.1 Finite Field Arithmetic

As described in specified in Section 2.2, all underlying arithmetic operations multiplication, addition, and inversion in the MEDS signature scheme are performed in the prime field $\mathbb{F}_q$ where q is either the 11-bit prime $2039 = 2^{11} - 9$ or

Table 2: Resource consumption and performance of the $\mathbb{F}_q$ arithmetic functions targeting an Xilinx Artix 7 (`xc7a200t`) FPGA. [†]2× in parallel.

		Resources							
Operation	**Field**	**Area**		**Memory**		**Cycles**	**Freq.**	**Time**	**Time×Area**
		(LUT)	(DSP)	(FF)	(BRAM)	(cyc.)	(MHz)	(us)	
Addition	$\mathbb{F}_{2039}$	27	0	23	—	1	244	0.004	0.110×10^3
	$\mathbb{F}_{4093}$	28	0	25	—	1	221	0.005	0.127×10^3
	Joint	33	0	25	—	1	201	0.005	0.164×10^3
Multiplication	$\mathbb{F}_{2039}$	30	1	40	—	4	361	0.011	0.333×10^3
	$\mathbb{F}_{4093}$	44	1	41	—	4	357	0.011	0.494×10^3
	Joint	46	1	45	—	4	343	0.012	0.536×10^3
Inverse[†]	$\mathbb{F}_{2039}$	0	0	1	1.0	1	392	0.003	—
	$\mathbb{F}_{4093}$	0	0	1	1.5	1	392	0.003	—
	Joint	12	0	1	2.5	1	269	0.004	0.045×10^3

the 12-bit prime $4093 = 2^{12} - 3$. We implement these operations as integer arithmetic modulo the prime q as described in the following subsections. An overview of the resource consumption and performance for each module is provided in Table 2.

Modular Addition. To perform modular additions, we use traditional carry-based integer adders followed by modular reduction. As the size of the field elements is small (11 or 12 bit), there is no need for any specialized technique to optimize the carry-based adders. For performing the modular reduction after the integer addition, we use multiplexer-based logic: Our modular reduction unit checks if the output is larger than the modulus q and if so, it subtracts the modulus from the output. Our modular addition unit consist of only one register stage between the addition and the reduction unit and has a input-independent latency of one clock cycle.

For the joint design, depending on the input signal `i_sel_4093`, either 2039 or 4093 is selected in the reduction via a multiplexer to be subtracted from the temporary result. Since the two-complements of 2039 and 4093 only differ in two positions, this selector only affects two bits and most bits remain constant and can be absorbed into the logic during optimization in the design synthesis. However, since carries depend on input data in the joint design, the subtraction requires slightly more logic than the entirely constant operands for the $\mathbb{F}_{2039}$ and $\mathbb{F}_{4093}$ designs. Supporting two prime fields in the joint design comes at only moderate additional cost (see Table 2): Overall, the joint design requires 33 LUTs and 25 registers while the $\mathbb{F}_{2039}$ design requires 27 LUTs and 23 registers and the $\mathbb{F}_{4093}$ design requires 28 LUTs and 25 registers.

Modular Multiplication. Since the size of each element in $\mathbb{F}_q$ is 11 or 12 bits, we can use the DSPs available on the target FPGA (AMD Artix 7 `xc7a200t-3`) to perform integer multiplication followed by full reduction. Using the DSPs helps to avoid long critical paths in the FPGA design. For modular reduction after integer multiplication, we design a specific modular reduction unit targeting the q value. As q is in a pseudo-Mersenne form ($q = 2^{11} - 9 = 2039$ or $q = 2^{12} - 3 = 4093$), we use the folding technique to perform the modular reduction [HGG07]. As the size of the multiplied output is up to $2\lceil \log_2(q) \rceil$ bits (i.e., 22 resp. 24 bits), we take the most-significant $\lceil \log_2(q) \rceil$ bits (i.e., 11 resp. 12 bits), of the output, multiply them by $2^{\lceil \log_2(q) \rceil} - q$ (i.e., $9 = 1001_b$ or $3 = 11_b$ — since the binary representation of the factor is sparse, we simple shift-and-add instead of multiplying), and add this to the least significant $\lceil \log_2(q) \rceil$ bits (i.e., 11 resp. 12 bits). As the result of the addition may have more than $\lceil \log_2(q) \rceil$ bits, we repeat this folding process until all most-significant bits are folded and until there is no more carry generated from addition after folding. Eventually, we use a multiplexer at the intermediate folded result r to identify corner cases $r \geq q$ and in case we detect one of these cases, we subtract the modulus q from r.

As for addition, many computations in the joint design are the same for both fields. The only difference in the folding reduction step is that for $q = 2039$ we need to shift the top bits by 9 and for $q = 4093$ by 3, which only requires a multiplexer. Also the final conditional subtraction just requires a multiplexer to select the prime similar to addition. Hence, the resource overhead of supporting both primes is low: The joint design requires 46 LUT and 45 registers, the $\mathbb{F}_{2039}$ design 30 LUT and 40 registers, and the $\mathbb{F}_{4093}$ design 44 LUT and 41 registers. The maximum frequency is similar but lowest for the joint design.

The input-independent latency for our modular multiplication unit for all variants is four clock cycles. One register is placed after the multiplication unit followed by two registers to pipeline the reduction unit efficiently to reduce the overall critical path. The output is not registered by default.

Modular Multiply-and-Add. We also provide a dedicated module for $\mathbb{F}_q$ multiply-and-add, which accepts three inputs $a, b, c \in \mathbb{F}_q$ and computes $a \cdot b + c$. As for $\mathbb{F}_q$ multiplication, we are using a DSP, now exploiting its multiply-and-add capabilities to include the addition without incurring additional logic cost. The only extra resource requirements compared to modular multiplication are for flip-flops to register the additional operand. The reduction step is performed in the same manner as described above for modular multiplication. Since the resource cost for this operation is equivalent to that of multiplication, it is not separately shown in Table 2.

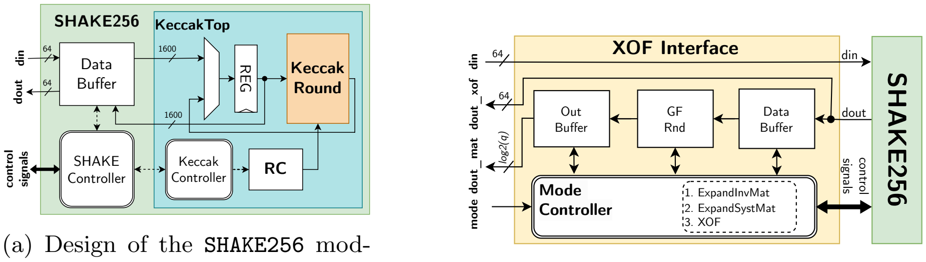
Modular Inversion. This operation is mostly required for matrix systemization (described in detail in Section 4.4). To perform that computation efficiently, we require low-latency field inversion. To achieve this and since the size of the finite field $\mathbb{F}_q$ with 11 and 12 bits is relatively small, we decided not to use expensive, high-latency inversion algorithms like the extended Euclidean algorithm, Fermat's little theorem or Montgomery inversion for our modular inversion module. Instead, we trade computation for memory and precompute the inverse of all elements in $\mathbb{F}_q$ at compile time and store them as look-up table in Block RAM (BRAM). Hence, we have a input-independent latency of only one clock-cycle for performing modular inversion. To use the memory resources as efficiently as possible, we are exploiting the dual-ported interface of the BRAM to perform two independent inversions in parallel, hence halving the effective resource cost.

The joint design for this operation provides look-up tables for both primes and some additional logic for selecting the output of the requested field. The resource requirements are shown on Table 2.

Summary. The resource evaluation in Table 2 demonstrates that hardware support for multiple fields does not impose a significant cost, provided the fields are "related," meaning they share the same reduction algorithm. Similar works that support two distinct prime fields [AMI+23,MR24] do not present a fine-grained analysis of the overhead introduced at the modular arithmetic level due to dual-field support; instead, they primarily emphasize the associated overhead at the level of polynomial multiplication units. In our reduction module, when employing the folding technique, efficient resource sharing can be achieved under the condition that the primes have a low Hamming distance. Consequently, supporting different prime fields in hardware is not prohibitively resource-intensive if the fields are selected carefully. This can make using different prime fields a viable design choice for cryptographic primitives when using different fields offers meaningful benefits, such as reduced signature or ciphertext sizes.

4.2 SHAKE256 and XOF Interface

In the context of MEDS SHAKE-256 is used for pseudo-random seed expansion and for hashing. SHAKE-256 [Dwo15] is a variant of the Keccak family of algorithms and uses the Keccak f-1600 permutation with an 1600 bit internal state. It provides hash outputs of arbitrary lengths at 256-bit cryptographic security.



(a) Design of the `SHAKE256` module.

(b) Design of the `XOF` interface module.

Fig. 2: Hardware designs of the `SHAKE256` and `XOF` interface modules.

Our `SHAKE256` module is built on a round-based Keccak implementation with additional auxiliary circuits (buffers) as shown in Figure 2a. 64 bit wide input words are concatenated to produce a sequence of 1600 bits, which is input to the `KeccakTop` module. After 24 rounds of permutations, the output sequence of 1600-bits is returned as a sequence of 64 bit words to the `SHAKE256`. The

	Resources					
Module	**Area**			**Memory**	**Cycles**	**Freq.**
	(LUT)	(DSP)	(FF)	(BRAM)	(cyc.)	(MHz)
SHAKE-256(Ours)	4 468	0	2 708	0	24	226.30
SHAKE-256 [Saa24]	5 443	0	2 445	0	24	100.00
SHAKE-256 [DXN+23]	4 797	0	1 845	0	74	166.00

[Saa24] constrained to 100 MHz, maximum frequency not reported.

Table 3: Synthesis results of our SHAKE-256 module targeting Artix 7 (xc7a200t) FPGA.

KeccakTop module has a buffer for storing the round constants (RC) and a multiplexer for switching between the initial and round-based inputs, as depicted in Figure 2a.

Table 3 shows the synthesis results of the SHAKE256 module. Compared to prior work in [Saa24] and [DXN+23], we report the highest maximum frequency (though [Saa24] is constrained to 100 MHz) at lower LUT usage but larger FF consumption at the same 24 cycles latency as [Saa24] as opposed to 74 cycles latency of [DXN+23]. Therefore, our design is quite competitive and provides an interesting trade-off between memory (FF) and overall wall clock time (due to the high frequency).

XOF Interface. The SHAKE256 module is also used as eXtendible Output Function (XOF) to generate pseudo-random data. We provide an XOF interface module as wrapper around the SHAKE256 module. Figure 2b shows the XOF interface module connected to the SHAKE256 module. There are two main functions in this module, to generate bit streams (e.g., for seeds) and to generate $\mathbb{F}_q$ field elements. The bit stream output is returned as sequence of 64 bit words through the dout_xof port and $\mathbb{F}_q$ output is returned via the dout_mat output port with a width of $\lceil \log_2(q) \rceil$ bits, i.e., 11 bits NIST Level V parameter sets and 12 bits width for the other levels (see Table 1).

In total, there are three modes of operation supported by this module:

- The ExpandInvMat mode generates a matrix $\mathbf{M} \in \mathbb{F}_q^{d \times d}$ from a seed input. The modes returns rejection-sampled $\mathbb{F}_q$ elements one by one through the dout_mat port in row-major order.
- The ExpandSysMat mode is used to generate matrices with k rows and mn columns in systematic form, i.e., $\mathbf{G}_0 \in \mathbb{F}_q^{k \times mn}$, from an input seed ($\sigma_{\mathbf{G}_0}$. In this mode, elements of the matrix are provided through the dout_mat port one by one in row-major order including the identity matrix at the left of $\mathbf{G}_0$.
- The XOF mode operates as XOF with any desired output length indicated by input i_dout_length (1 bytes to $2^{16} - 1$ bytes output length).

The XOF interface module only adds a small resource overhead of about 150 LUT and about 130 registers to the SHAKE256 module.

4.3 Matrix Multiplication

Matrix multiplication is one of the most used operations in the MEDS It is used in the π operation and for computing the μ_i and ν_i during signing. Consequently, we take care to design an efficient matrix multiplication unit that is parameterizable at compile time to enable performance trade-offs. As mentioned in Section 3.1, matrices in MEDS are specified to be sampled in row-wise order.

The general task of this module is to compute $\mathbf{C} = \mathbf{A} \cdot \mathbf{B}$ for matrices $\mathbf{A} \in \mathbb{F}_q^{d_1 \times d_2}$, $\mathbf{B} \in \mathbb{F}_q^{d_2 \times d_3}$, and $\mathbf{C} \in \mathbb{F}_q^{d_1 \times d_3}$ for some matrix dimensions d_1, d_2, and d_3. Instead of traditional schoolbook matrix multiplication with limited parallelism, we are using a vector-based approach similar to standard SIMD techniques in our matrix multiplication design. Asymptotically more efficient matrix multiplication algorithms do not apply to MEDS due to the relatively small matrix dimensions. We start with multiplying all first column elements $\mathbf{A}[i;0]$ from matrix $\mathbf{A}$ with all first column-block elements $\mathbf{B}[i;0,p_{\text{mm}}-1]$ of matrix $\mathbf{B}$, generating a temporary sum $\mathbf{C}'[i;0,p_{\text{mm}}-1]$. We store these temporary sums in the corresponding location of the result buffer of $\mathbf{C}$ in a BRAM. We then multiply all first column elements $\mathbf{A}[i;0]$ from matrix $\mathbf{A}$ with all second column-block elements $\mathbf{B}[i;p_{\text{mm}},2p_{\text{mm}}-1]$ of the matrix $\mathbf{B}$ and store these temporary sums $\mathbf{C}'[i;p_{\text{mm}},2p_{\text{mm}}-1]$ in the result buffer. We continue iterating over the first column elements $\mathbf{A}[i;0]$ until we processed all column-block elements $\mathbf{B}[i;jp_{\text{mm}},(j+1)p_{\text{mm}}-1]$. We then move on to the second column elements $\mathbf{A}[i;1]$ of matrix $\mathbf{A}$ and repeat the process of iterating over words $\mathbf{B}[i;0,p_{\text{mm}}-1]$ of matrix $\mathbf{B}$ while adding the result to the temporary sum $\mathbf{C}'[i;0,p_{\text{mm}}-1]$ that is stored in the result buffer. We repeat this process until we have multiplied all columns of matrix $\mathbf{A}$ with all row blocks of matrix $\mathbf{B}$ to accumulate the final result $\mathbf{C}$.

This approach allows us to take advantage of the pipelining of the $\mathbb{F}_q$ multiplier and also allows us to perform single-element additions in each iteration. This design also allows us to stream the inputs to the multiplication unit and to make best use of all pipelining registers. The latency l_{mm} of our matrix multiplication unit can be computed from the matrix dimensions d_1, d_2, and d_3, the number of pipeline stages l_{mmpipe}, and the performance parameter p_{mm} (i.e., the column-block width) with the following formula:

$$l_{\text{mm}}(d_1, d_2, d_3) = l_{\text{mmpipe}} + \frac{d_1 \cdot d_2 \cdot d_3}{p_{\text{mm}}}.$$

Based on which column from matrix $\mathbf{A}$ is being processed and since the matrices are stored in a column-block format, the element from matrix $\mathbf{A}$ is chosen from the memory word that stores the desired column using a variable shifter as shown in Figure 4. For matrix $\mathbf{B}$, the memory addresses are accessed sequentially and complete memory words are being processed, i.e., multiplied by a scalar from matrix $\mathbf{A}$ and added to a column-block memory word from matrix $\mathbf{C}$.

Performance and resource demands for the different parameter sets and design variants are shown in Table 4. Since the arithmetic latencies of the field-specific variants and the joint design are identical, the number of cycles depends

Table 4: Comparison of the time and area for our Matrix Multiplication module targeting Xilinx Artix 7 (`xc7a200t`) FPGA.

			Resources							
d_1, d_2, d_3	p_{mm}	q	**Area**		**Memory**		**Cycles**	**Freq.**	**Time**	**Time×Area**
			(LUT)	(DSP)	(FF)	(BRAM)	(cyc.)	(MHz)	(us)	
				Parameter-set specific designs:						
$14, 14, 14$	2	4 093	279	2	197	0.5	1 380	218	6.34	2×10^3
$22, 22, 22$	2	4 093	294	2	206	0.5	5 332	218	24.50	7×10^3
$30, 30, 30$	2	2 039	298	2	206	0.5	13 508	195	69.20	21×10^3
$14, 14, 14$	4	4 093	383	4	317	1.0	792	218	3.64	1×10^3
$22, 22, 22$	4	4 093	396	4	332	1.0	2 912	218	13.38	5×10^3
$30, 30, 30$	4	2 039	387	4	320	1.0	7 208	218	33.11	13×10^3
$14, 14, 14$	8	4 093	598	8	565	1.5	400	218	1.84	1×10^3
$22, 22, 22$	8	4 093	625	8	580	1.5	1 460	199	7.32	5×10^3
$30, 30, 30$	8	2 039	606	8	556	1.5	3 608	199	18.09	11×10^3
					Joint designs:					
$14, 14, 14$	2	both	335	2	220	0.5	1 380	199	6.93	2×10^3
$22, 22, 22$							5 332		26.79	9×10^3
$30, 30, 30$							13 508		67.88	23×10^3
$14, 14, 14$	4	both	460	4	348	1.0	792	217	3.65	2×10^3
$22, 22, 22$							2 912		13.42	6×10^3
$30, 30, 30$							7 208		33.23	15×10^3
$14, 14, 14$	8	both	753	8	612	1.5	400	196	2.04	2×10^3
$22, 22, 22$							1 460		7.43	6×10^3
$30, 30, 30$							3 608		18.36	14×10^3

only on the matrix dimensions d_1, d_2, and d_3 as well as the degree of parallelization p_{mm}. The maximum frequency is in about the same range for all cases with about 10% variance. The joint design, however, requires more computational logic resources for the joint field arithmetic (see Section 4.1) and for supporting different matrix dimensions in the same design.

The result **C** of the matrix multiplication is stored in BRAM. The BRAM components in Xilinx Artix 7 are dual-ported with a size of 18 kbit and 36 kbit. The smaller variant is counted as "0.5 BRAM". The word widths are 18 bit and 36 bit respectively, which can be doubled by using both ports. The total sortage capacity of one 0.5 BRAM is sufficient to store the largest matrix variant, which requires $30 \cdot 30 \cdot 12 = 10.8$ kbit of storage. However, since the maxium BRAM word with when using both ports is 36 bit respective 72 bit, we require one 0.5 BRAM for $p_{\text{mm}} = 4$ to access a word width of $2 \cdot 12\,\text{bit} = 24\,\text{bit}$ (respective $2 \cdot 11\,\text{bit} = 22\,\text{bit}$), one full BRAM for $p_{\text{mm}} = 4$ to access a word width of $4 \cdot 12\,\text{bit} = 48\,\text{bit}$ (respective $4 \cdot 11\,\text{bit} = 44\,\text{bit}$) and 1.5 BRAMs for $p_{\text{mm}} = 4$.

4.4 Matrix Systemization

Our hardware design for matrix systemization follows the state-of-the art such as [SWM+10,WSN16,WSN17,CCD+22,ZZC+23,BCH+23]: We are using a proces-

Table 5: Comparison of the time and area for our Systemizer module targeting Xilinx Artix 7 (`xc7a200t`) FPGA.

			Resources							
Matrix Size	p_{mm}	q	**Area**		**Memory**		**Cycles**	**Freq.**	**Time**	**Time×Area**
rows×cols			(LUT)	(DSP)	(FF)	(BRAM)	(cyc.)	(MHz)	(us)	
Parameter-set specific designs:										
14×196	2	4 093	943	6	1 076	1.5	9 345	185	50.52	48×10^3
22×484	2	4 093	1 067	6	1 334	1.5	57 393	177	324.21	346×10^3
30×900	2	2 039	1 178	6	1 511	1.0	199 393	199	1 002.15	$1\,181 \times 10^3$
14×196	4	4 093	2 172	20	2 670	3.0	2 718	177	15.39	33×10^3
22×484	4	4 093	2 288	20	3 155	3.0	15 702	183	85.94	197×10^3
30×900	4	2 039	2 388	20	3 408	2.0	53 222	173	308.32	736×10^3
14×196	8	4 093	6 442	72	7 030	7.5	794	193	4.12	27×10^3
22×484	8	4 093	6 645	72	8 192	7.5	4 069	166	24.54	163×10^3
30×900	8	2 039	6 783	72	8 939	5.0	13 490	170	79.24	537×10^3
Joint designs:										
14×196	2	both	1 263	6	1 595	2.5	9 345	177	52.67	67×10^3
22×484							57 393		323.47	409×10^3
30×900							199 393		1 123.78	$1\,419 \times 10^3$
14×196	4	both	2 778	20	3 685	5.0	2 718	173	15.72	44×10^3
22×484							15 702		90.82	252×10^3
30×900							53 222		307.84	855×10^3
14×196	8	both	8 333	72	9 138	10.0	794	169	4.69	39×10^3
22×484							4 069		24.04	200×10^3
30×900							13 490		79.71	664×10^3

sor array of a quadratic shape that processes column-blocks of the input matrix in several rounds consisting of several steps. For a detailed description of the operation of the systemizer, we refer to [CCD+22, Section 3].

The design in [BCH+23] combines matrix systemization and matrix multiplication in one design with the goal to share resources between these two operations. However, we are using a pipelined overall design to increase signing and verification performance and hence we need individual modules for matrix systemization and multiplication. Due to the pipelined overall design, all modules are under load most of the time and resources are in use in parallel — and hence do not need to be shared to achieve efficiency.

The design in [ZZC+23] reduces memory cost by streaming out parts of the systemized matrix as complete columns become available during computation without storing the entire matrix on-chip. This optimization does not apply to MEDS (except if external memory is used), since the larg $k \times nm$ matrices $\tilde{\mathbf{G}}_\mathbf{i}$ and $\hat{\mathbf{G}}_\mathbf{i}$ need to be fed row-wise into the hash function.

In contrast to earlier designs aiming at the systemization of matrices over $\mathbb{F}_2$ as they appear, e.g., in binary codes used for the code-based McEliece cryptosystem [SWM+10,WSN16,CCD+22] or matrices over binary fields as used, e.g., in code-based [WSN17] as well as multivariate cryptosystems [FG18,BCH+23], we are dealing with matrices over medium-sized prime fields, which means that we are dealing with a longer arithmetic latency: while binary field arithmetic in earlier work has a latency of one cycle, our latencies for multiplication and addition are several cycles. This does not affect the overall process of the systemization, but we need to take the latencies into consideration to avoid race conditions during the parallel processing of that data. For example, our design needs to wait for different amounts of overhead cycles between consecutive rounds to wait for data of the previous round being written back to memory depending on the number of column blocks that are being processed in between. Depending on the size of the processor array, the arithmetic latencies add up and we quickly get an overall latency of the processor array of dozens of cycles. However, in contrast to earlier designs [WSN16,WSN17,CCD^{+}22] we do pipeline computation between consecutive rounds as much as possible.

Similar to [BCH^{+}23], we are not too concerned about systemization failures, which happen only very rarely for our field sizes: The probability that a random $n \times m$ matrix ($m \geq n$) over a finite field $\mathbb{F}_q$ has a systematic form is the same as that of an $n \times n$ matrix over $\mathbb{F}_q$ being full rank, i.e.,

$$\prod_{i=0}^{n-1} \left(1 - \frac{1}{q^{n-i}}\right).$$

For large n and q, this probability is approximately $1-\frac{1}{q}$. Hence, for $q = 4093$, we have a probability of success for the SF operation in line 20 of about 99.98% and for $q = 2039$ of about 99.96%. In other words, we expect a failure to systemize a matrix over $\mathbb{F}_q$ roughly every q attempts — for $q = 4093$ we expect one failure roughly every 4093 and for $q = 2039$ roughly every 2039 systemization attempts. Therefore, we omit some of the tweaks descried in [CCD^{+}22]: As we do not require a mechanism to detect failure as early as possible but only at the end of systemization, we do not need store all operations that are generated in pivoting steps in-place in the data memory for later use. Instead, we can use a relatively small additional memory as in [WSN16,WSN17,BCH^{+}23] to store the operations only of the current pivoting step, overwriting those of previous steps. Similar to these designs, we detect systemization failure at the end of the computation in the final pivoting step.

Figure 6 shows an overview of the systemizer module. We use the performance parameter p_{sys} to specify the number of both rows and columns of the processor array. The number of columns corresponds to the column-block width of the corresponding BRAM that stores the matrix data. We are grouping the processor elements of each row into a vector Arithmetic Logic Units (ALUs) of width p_{sys}.

Figure 6 also shows the logic of each vector unit in more detail. Each processor element in row i and column j of the processor array stores the data of its current pivoting row (if any) in registers `SAR[i,j]`. Each processor row receives inputs from the previous row (or from data memory) via input wires `SA_row_in[i,j]` and passes on its outputs via output wires `SA_row_in_next[i,j]` to the next row (or back to data memory).

At the beginning of each pivoting round, the registers `SAR[i,j]` are empty. Once a pivot matrix row reaches the corresponding processor row, it is stored in registers SAR[i,j]. The pivoting rows need to be normalized, which requires a scalar inversion and a scalar-vector multiplication. Later in the process, once a pivoting row has been found, incoming rows are reduced by the pivoting row, which requires a scalar multiplication and a subtraction.

Now, as mentioned before, the prime-filed arithmetic operations have a considerable latency. Since processor array operates in a pipelined fashion, the result of the normalization is already required for the next input. Overcoming the latencies of the normalization would require several buffer stages. Also, we would like to use only one vector unit computing scalar-vector multiplication and vector addition, not two for normalization and reduction.

Our solution to this problem is to delay the normalization of pivoting rows until they are read out of the processor array at the end of the corresponding step. This means that a new pivot row can be stored in SAR[i,j] unmodified and is available without further latency in the next cycle. For the normalization and reduction we now can use just one single vector ALU, but we need to prepare the respective inputs at the cost of several cycles of latency. The vector ALU now always multiplies SAR[i,j] by a scalar factor either for normalization or for reduction and adds an input vector to the result which is either all zero for normalization or SA_row_in[i,j] for reduction. We need one additional scalar-scalar multiplication per processor row for normalizing the pivot row towards reducing follow-up rows.

In a pipelined process, we set norm to the inverse of SAR[i,i] (once a pivot round has been found, we set norm to one otherwise). We then compute SA_row_in[i,i] times norm and store the result in mul. We then compute the negative of mul if we are reducing non-pivot lines but keep it untouched when normalizing the pivot row at the end of the step. Finally, we compute the output vector SA_row_in_next[i,] as scalar mul times vector SAR[i,] plus vector SA_row_in[i,] in case of reduction or plus the zero vector in case of normalization. All respective inputs need to be delayed using registers until they are consumed in the pipeline.

The top part of Figure 6 shows the pipelined modules that prepare the inputs for and control the behavior of the vector unit and the bottom part shows the vector unit itself with p_{sys} multiply-and-add units (see Section 4.1). The overall latency of each processor row is defined by the sequential operation of the top part and the parallel latency of bottom part as the latencies of one multiplication, one addition, and one multiply-and-add operation plus some additional latencies to optimize data paths of control logic.

The performance of our systemizer design is shown in Table 5. The maximum frequency ranges between 166 MHz and 199 MHz. The joint design requires the same number of cycles but its maximum frequency is rather at the low end of the range. We were not able to force the synthesis tool to synthesise the lookup tables for the finite-field inversion into BRAM — the tool is using LUT resources instead. Since the joint design needs to include both tables, there is a significant increase in LUT usage compared to the $\mathbb{F}_{4093}$ design.

Table 6: Comparison of the time and area for our Pi module targeting Xilinx Artix 7 (`xc7a200t`) FPGA for $p_{\pi mm} = 4$.

			Resources							
k, m, n	$p_{\pi pm}$	q	**Area**		**Memory**		**Cycles**	**Freq.**	**Time**	**Time×Area**
			(LUT)	(DSP)	(FF)	(BRAM)	(cyc.)	(MHz)	(us)	
Parameter-set specific designs:										
$14, 14, 14$	2	4 093	1 329	8	660	2.0	25 474	218	117.03	155.0×10^3
$22, 22, 22$	2	4 093	1 610	8	706	2.0	91 354	218	419.68	675.0×10^3
$30, 30, 30$	2	2 039	1 655	8	688	2.0	223 474	218	1 026.64	$1\,697.0 \times 10^3$
$14, 14, 14$	4	4 093	2 172	16	1 248	4.0	13 588	218	62.42	135.0×10^3
$22, 22, 22$	4	4 093	2 204	16	1 313	4.0	48 724	196	249.17	548.0×10^3
$30, 30, 30$	4	2 039	2 457	16	1 269	4.0	119 188	196	609.53	$1\,494.0 \times 10^3$
$14, 14, 14$	8	4 093	3 350	32	2 438	8.0	6 796	218	31.22	104.0×10^3
$22, 22, 22$	8	4 093	3 526	32	2 536	8.0	24 364	218	111.93	394.0×10^3
$30, 30, 30$	8	2 039	4 028	32	2 437	8.0	59 596	218	273.78	$1\,101.0 \times 10^3$
Joint designs:										
$14, 14, 14$	2	both	1 891	8	751	2.0	25 474	217	117.44	197.0×10^3
$22, 22, 22$							91 354		421.14	705.0×10^3
$30, 30, 30$							223 474		1 030.22	$1\,725.0 \times 10^3$
$14, 14, 14$	4	both	2 856	16	1 392	4.0	13 588	217	62.64	190.0×10^3
$22, 22, 22$							48 724		224.62	682.0×10^3
$30, 30, 30$							119 188		549.46	$1\,668.0 \times 10^3$
$14, 14, 14$	8	both	4 732	32	2 661	8.0	6 796	217	31.33	174.0×10^3
$22, 22, 22$							24 364		112.32	623.0×10^3
$30, 30, 30$							59 596		274.74	$1\,523.0 \times 10^3$

4.5 Pi Module

The algorithm for the π operation is given in [CNP+23a, Algorithm 7] and was introduced in Section 2.1. To recap, the $\pi_{\mathbf{A},\mathbf{B}}(\mathbf{G})$ operation for $\mathbf{A} \in \mathbb{F}_q^{m\times m}$, $\mathbf{B} \in \mathbb{F}_q^{n\times n}$, and $\mathbf{G} \in \mathbb{F}_q^{k\times mn}$ is defined as follows: arrange the rows of matrix $\mathbf{G}$ into a k separate $m\times n$ matrices $\{\mathbf{P}_0, ..., \mathbf{P}_{k-1}\}$ and then compute $\mathbf{P}'_i = \mathbf{A}\cdot\mathbf{P}_i\cdot\mathbf{B}$. Finally, we get the result as matrix $\mathbf{G}' \in \mathbb{F}_q^{k\times mn}$ by packing $\mathbf{P}'_i$ into row i of $\mathbf{G}'$.

In our implementation of the π operation, we avoid the first step of arranging the rows of matrix $\mathbf{G}$ into k separate $m \times n$ matrices. Instead, while generating matrix $\mathbf{G}$ using the `ExpandSystMat` module(described in Section 4.6), the matrix is already arranged in k separate BRAM units. Therefore, we can directly perform the $\mathbf{A}\cdot\mathbf{P}_i\cdot\mathbf{B}$ operation using our matrix multiplication unit (described in Section 4.3). We first perform $\mathbf{T} = \mathbf{A}\cdot\mathbf{P}$ and then $\mathbf{P}' = \mathbf{T}\cdot\mathbf{B}$. Figure 5 shows the design of our π module.

We provide a performance parameter $p_{\pi\mathrm{pm}}$ for the k matrix multiplications such that we can select number of matrix multiplications to be performed in parallel (the "pm" strands for parallel matrix multiplications). We control the number of parallel finite field operations in the matrix multiplication units of the π operation individually by setting the performance parameter p_{mm} of the matrix multiplication unit (see Section 4.3) to $p_{\pi\mathrm{mm}}$ for the π operation. This gives us two different performance parameters for our π design: first, the number $p_{\pi\text{mm}}$ of prime field ALUs per matrix multiplication and second, the number $p_{\pi\text{pm}}$ of parallel matrix multiplications. The latency l_π of our π operation can be computed with following formula:

$$l_\pi(k,m,n) = \frac{k}{p_{\pi\text{pm}}}\left(l_{\text{mm}}(m,m,n) + l_{\text{mm}}(m,n,n)\right) = \frac{k}{p_{\pi\text{pm}}}\left(2l_{\text{mmpipe}} + \frac{mmn+mnn}{p_{\pi\text{mm}}}\right)$$

with l_{mm} and l_{mmpipe} as defined in Section 4.3.

The synthesis results for our `Pi` module for different configurations are shown in Table 6. The maximum frequency is similar for all variants. The larger matrix dimensions for $k, m, m = 30$ seem to outweigh the smaller resource requirement of $\mathbb{F}_{2039}$ arithmetic. The resource overhead of the joint design compared to the field-specific variants is moderate.

4.6 ExpandSystMat, ExpandInvMat, and XOF

ExpandSystMat. As shown in Algorithm 1 and Algorithm 2, the ExpandSystMat operation is used to generate matrix $\mathbf{G}_0 \in \mathbb{F}_q^{k\times mn}$, which has the shape $\mathbf{G}_0 = (I_k|\mathbf{G}_0')$ with $\mathbf{G}_0' \in \mathbb{F}_q^{k\times mn-k}$, where $I_k \in \mathbb{F}_q^{k\times k}$ is the identity matrix and the sub-matrix $\mathbf{G}_0'$ consists of finite field elements pseudo-randomly sampled using the `SHAKE256` module with the seed σ_{G_0}. The matrix elements are populated row-wise. As the elements of the matrix are in $\mathbb{F}_q$, the sampling of these elements involves rejection sampling, i.e., each sampled 16-bit value v is truncated to $\lceil \log_2(q) \rceil$ bits and values $v \geq q$ are discarded. Since MEDS is using semi-Mersenne primes, the rejection probability is relatively low but non-negligible with $9/2039 = 0.44\%$ for $\mathbb{F}_{2039}$ and $3/4093 = 0.7\%$ for $\mathbb{F}_{4093}$.

Since SHAKE256 is used in multiple modules for MEDS, in our hardware design we share one SHAKE256 instance among several different modules through the `XOF` interface shown in Figure 3. Our hardware design for `ExpandSystMat` makes use of the `XOF` interface, which provides the mode `ExpandSystMat` to generate the sampled matrix elements. The `XOF` interface also provides an option to output the left square identity part of the matrix in row-wise order followed by the sampled elements from the `SHAKE256` module. The `XOF` interface also handles the rejection sampling based on the choice of parameter set at runtime.

The `XOF` interface outputs one element sequentially after another along with a valid signal. Since $\mathbf{G}_0$ is used in the `Pi` module as shown in line 19 of Algorithm 1, it is necessary to store the matrix data in a compatible manner for the matrix multiplication module so that it is easily accessible by the `Pi` module.

The `ExpandSystMat` module stores the matrix elements generated by the `XOF` interface and, based on the selected performance parameter $p_{\pi\text{mm}}$ of the `Pi` module (described in Section 4.5), clubs $p_{\pi\text{mm}}$ elements together using a shift register. Each row of $\mathbf{G}_0$ is stored separately in one of k BRAMs as shown in Figure 3 such that the rows can be treated as matrices $\mathbf{P}_i \in \mathbb{F}_q^{m\times n}$ in the k matrix multiplications of the `Pi` module. The values for m, n, and k can be configured at runtime based on the choice of the parameter set.

ExpandInvMat. The ExpandInvMat operation takes a seed (e.g., $\sigma_{\tilde{\mathbf{A}}}$) and the matrix dimensions d as an input and generates an invertible matrix (e.g., $\tilde{\mathbf{A}} \in \mathbb{F}_q^{n \times n}$ for $d = n$). As defined in [CNP+23a, Algorithm 8]. It is used in both signing and verification operations as shown in Algorithm 1 and Algorithm 2. The operations involved in ExpandInvMat are as follows: firstly, the seed $\sigma_{\tilde{\mathbf{A}}}$ is absorbed into a SHAKE256 state, from which then pseudo-random bits are sampled into $d \times d$ finite field elements. Similar to the ExpandSystMat operation, the $\mathbb{F}_q$ elements are sampled row-wise and rejection sampling is applied to discard elements greater than or equal to q. The generated matrix is then checked for invertability and if it is not invertible, then another $d \times d$ field elements are sampled from the current state of SHAKE256 and invertability is checked again. This process is repeated until an invertible matrix has been found.

In our hardware design, we accomplish this operation by loading the seed into the `XOF` interface module (described in Section 4.2), by selecting the `ExpandInvMat` mode from the mode controller, and by setting the matrix dimension d at runtime. The `XOF` interface then provides the $d \times d$ number of matrix elements sequentially one element after another similar to ExpandSystMat.

The next step is to check the invertability of the matrix. We use a `Systemizer` module (described in Section 4.4) for this purpose: An invertible matrix has a systematic form, i.e., the result of Gaussian elimination must be the identity matrix. Therefore, if systemization of a square matrix is successful, then the matrix is invertible.

We store the elements in row-wise order in a BRAM in column-block format where BRAM location stored a word of p_{sys} elements. After all elements have been stored in BRAM, the `Systemizer` module is initiated and the `Systemizer` notifies the `ExpandInvMat` controller if systemization was successful, i.e, if the matrix is invertible or not. If it is invertible, the `ExpandInvMat` module returns success. Otherwise, the `ExpandInvMat` controller requests another stream of $d \times d$ elements from the current state of the `XOF` interface and repeats the invertability check until a invertible matrix is found.

XOF. As shown in Algorithm 1 and Algorithm 2, the XOF operation is used in multiple instances throughout the signing and verification operation. The XOF operation takes variable sized seed inputs and generates a pseudo-random bit stream using SHAKE256. The input seed size and output pseudo-random bit-stream legnth are also provided as an input to XOF.

In our hardware design, we use the `XOF` interface module described in Section 4.2 for all the XOF operations. The `XOF` interface module provides a `XOF` mode in which the interface expects three inputs: the seed, the seed size, and the output size. The seed is loaded in to the `SHAKE256` module which then generates the pseudo-random bits. Since the logic involved to initialize the `XOF` interface with the `XOF` mode and to retrieve the output data is comparatively simple, we do not have a separate module for the XOF operation. This logic simply is part of the main controller, the Stage 1 logic, and the `ParseHash` module that are performing XOF operations in line 12, line 16, and line 24 of Algorithm 1 and line 21 and line 9 of Algorithm 2 respectively.

4.7 SeedTree, SeedTreeToPath, and PathToSeedTree

SeedTree. The seed tree (used in line 13 of Algorithm 1) is structured as a binary tree. Starting at the root node, the seed value of each node is expanded to the two seed values of its child nodes using XOF. The module `SeedTree` iteratively generates the seed tree by reading seeds from and storing them into a BRAM. Each seed in the seedtree is of width $\ell_{\mathsf{tree\_seed}}$ (values given in [CNP$^+$23a, Table 4]). The `SeedTree` module executes hash operations starting from the top node down to the lower nodes and from left to right. The inputs and outputs of the hash computation performed at each node are as follows:

$$(\rho_{i+1,2j}|\rho_{i+1,2j+1}) = \text{SHAKE256}(\sigma|\rho_{i,j}|a)$$

Where | means byte concatenation, σ is the salt provided as input to SeedTree, ρ is the seed data of the parent node at position i on level j in the tree, $\rho_{i+1,2j}$ and $\rho_{i+1,2j+1}$ are the new seeds on the next level for the left and right child nodes, and $a = 2^i - 1 + j$ is the node index. The process of generating the seed tree starts from the top root node and sequentially generates seed data for the nodes on higher levels.

In MEDS sign (given in Algorithm 1), the SeedTree method is used to compute and return the seed data of the leaf nodes. However, depending on the parameter set of MEDS, only t seeds of the leftmost leaf nodes are used in the subsequent operations. For example, in the MEDS-9923 parameter set, we need $t = 1152$ seeds. Therefore, the seed tree has $\lceil \log_2(1152) \rceil = 11$ levels with 2048 leaf-node seeds, of which only the $t = 1152$ seeds of the leftmost leaf nodes are used.

In our hardware design we take a straight-forward approach, by constructing the entire seed tree structure which requires

$$\sum_{i=0}^{h-1} 2^i = 2^h - 1$$

hash operations, where $h = \lceil \log_2(t) \rceil$ is the height of the tree. Therefore for the earlier discussed example of MEDS-9923 with $t = 1152$, the total number of requires hash operations is 2047. Figure 7 illustrates the hardware design of our `SeedTree` module. The seeds are stored in chunks of 64-bits since the `XOF` module has a 64-bit interface (as detailed in Section 4.2). The root seed is loaded into the first words of the BRAM and the rest of the seed tree is constructed in breadth-first traversal order.

SeedTreeToPath. The operation SeedTreeToPath (described in [CNP$^+$23a, Section 3.3]) is used in line 32 in Algorithm 1. It takes the root seed, salt and the challenge vector $(\{h_0, h_1, ..., h_{t-1}\} \in \{0, 1, ..., s-1\})$ as input, generates the complete seed tree, and then for each i with $h_i \neq 0$ removes the seed of the i^{th} leaf node as well as all corresponding parent nodes up to the root node. The output then are the root nodes of the remaining sub-trees that for the signature path.

In our implementation of the SeedTreeToPath operation, we simply skip the generation of the seed tree and instead use the seed tree data that already had been generated in line 13 of Algorithm 1 as described earlier in this subsection. We also do not explicitly delete the leafs and their parent nodes but just copy the valid seeds that form the signature path into a separate contiguous path memory (shown in Figure 3).

PathToSeedTree. The operation PathToSeedTree is used in the MEDS verify algorithm (line 10 in Algorithm 2). This function (as described in [CNP+23a, Section 3.3]) takes the challenge vector ($\{h_0, h_1, ..., h_{t-1}\} \in \{0, 1, ..., s-1\}$), seed tree path, and the salt as input and recomputes a partial seed tree consisting of the leaf nodes that are required for generating μ_i and ν_i for all i with $h_i = 0$. The output from this operations are the leaf nodes of the partial seed tree.

In our `PathToSeedTree` module, we identify the i values with $h_i \neq 0$ and compute the corresponding memory address of the BRAM for the respective seed given in the input path and copy the seed to that location. After copying the seed, we then expand the partial seed tree in a depth-first traversal order until we reach the leaf nodes. We repeat this operation for all seeds given in the path to obtain the entire partial tree.

4.8 Compress and Hash

The Compress operation of the MEDS specification addresses the difference between the size of $\mathbb{F}_q$ elements and the basic word size of computer architectures by converting field elements into a byte stream. This compression reduces the size of the public key and signatures for storage and transmission. For the purpose within the MEDS sign and verify operations, the compressed byte stream is also used as input to the hash operation.

In hardware, we can easily define the width of buses and register banks (and to some degree of memory words) precisely to match number of bits required to represent $\mathbb{F}_q$ elements. However, to feed $\mathbb{F}_q$ elements into the interface of the `SHAKE256` module, we still need to pack them into a byte stream as the `SHAKE256` module takes 64-bit words as input.

The `CompressHash` module receives the data of matrices $\tilde{\mathbf{G}}_i$ and feeds concatenated data bits in 64-bit blocks into the `SHAKE256` module while omitting the identity part of the $\tilde{\mathbf{G}}_i$ as they are in systematic form. Once all $\tilde{\mathbf{G}}_i$ have been absorbed by the `SHAKE256` module, the message msg is feed into the `SHAKE256` module as well and eventually the hashing is finalized and the message digest d is returned as result.

4.9 Parse Hash

The ParseHash operation takes the message digest d as bit string of length ℓ_{digest} as well as s, t, and w as inputs and generates the challenge string h as weight-w vector of t elements of the set $\{0, \ldots, s-1\}$. As specified in [CNP+23a, Algorithm 9], the ParseHash operation performs the following steps: Firstly, the input d is loaded into a XOF. From the XOF, depending upon the value of t, either one or two bytes are drawn. From the these bytes only the least significant $\lceil \log_2(t) \rceil$ bits are considered and assigned to the integer variable i_{pos}. If $i_{\text{pos}} < t$, then the value of $_{\text{pos}}$ is selected for a non-zero location of vector h. If $i_{\text{pos}} \geq t$, then the value is discarded and the process is repeated until it succeeds. Additionally, the i_{pos} value is also discarded in case we find duplicates, i.e., if $h_{i_{\text{pos}}} \neq 0$. Following that, one more byte is drawn from XOF to get value v. Only the least significant $\lceil \log_2(s) \rceil$ bits are considered and assigned to v. If $0 < v < s$, then v is assigned to the $h_{i_{\text{pos}}}$ location. If $v = 0$ or $v \geq s$, then v is discarded and the process is repeated until succcess. This process of finding i_{pos} followed by v is repeated until w non-zero elements are found. The generation of the challenge string h operates on public data and therefore timing security is not relevant.

In our hardware design, our `ParseHash` module implements this operation by loading d into the `XOF` interface module (described in Section 4.2) and selecting the `XOF` mode from the mode controller. Since s, t, and w are fixed for a given parameter set (see Table 1), we fix these values at synthesis time. As specified in Section 4.2, the `XOF` interface module provides 64-bit data output blocks along with a valid signal. We put these into a shift register and draw one or two bytes from the shift register at a time for the assignment to i_{pos} and v. Since the maximum value of i_{pos} is $t - 1$ (which is comparatively a small number), for the duplicate detection of i_{pos} we use a bit-array map: We initialize a bit array register with all zeros. As the new i_{pos} values are generated, we pull the signal corresponding to i_{pos} to high and compare the signal bus to the state stored in the bit array register. If the new assignment and the previous state are the same, a duplicate is detected. Otherwise, we update the bit array register and proceed. This technique allows us to perform duplicate detection on-the-fly (with zero-cycle latency) at a cost of only one t bit register as opposed to check a BRAM location for duplicates with a one-cycle latency. Furthermore, in our hardware design, rather than initializing an array in BRAM with t elements for h, where large amount of locations will be zero, we only store i_{pos} and v values sequentially in two separate BRAMs of depth w. The advantage of storing i_{pos} and v separately is described in Section 4.10.

4.10 Signing and Verification

Figure 3 shows the block diagram of the overall hardware design combining both MEDS sign and verify algorithms detailed in Algorithm 1 and Algorithm 2 respectively by sharing hardware modules between both operations.

In sign, we assume that byte deserialization of the input data for the secret key sk into the seed $\sigma_{\mathbf{G}_0}$ and the matrices $\mathbf{A}_i^{-1}$ and $\mathbf{B}_i^{-1}$ using the `Decompress` module are performed on a higher layer. We also leave the composition of the signed message $\mathsf{msg}_{\mathsf{sig}}$ including the serialization of output data μ_i and ν_i using the `Compress` module to a higher layer and just provide read ports to its components as output ports to the design. We also omit the implementation of the random number generator $\mathsf{Randombytes}$ and instead assume that the random seed δ is provided from a higher level.

Similarly, in verification, we assume that byte deserialization of the input data for the public key pk into the seed $\sigma_{\mathbf{G}_0}$ and deserialization of signed message $\mathsf{msg}_{\mathsf{sig}}$ into p, d, α, and msg are performed at a higher layer.

We note that, in our hardware design, the operations `XOF`, `ExpandInvMat`, `SeedTree`, and `PathToSeedTree` share the same SHAKE256 module as shown in Figure 3 but that we use a separate SHAKE256 module that is shared between `ExpandSystMat`, `ParseHash` for the combined `Compress` and `Hash` operation (line 23 in Algorithm 1 and line 28 in Algorithm 2). The reason for this will be described in the discussion of the loop pipeline later on in this section.

The most expensive operations π and `SF` are in the main for-loops of the sign and verify operations i.e., line 14 to line 22 in Algorithm 1 and line 12 to line 27 in Algorithm 2 respectively. Initially, we had considered to construct a large finite field vector ALU for the use in both π and `SF`. However, the required multiplexing and control logic likely would have been inefficient, complex, error prone, and hard to maintain. Also, π requires about two times more finite field multiplications compared to `SF` but no finite field inversions and the different matrices that are systemized in `SF` and `ExpandInvMat` have different dimensions. A joint large finite field vector ALU would hence lead to a less fine-grained control over performance vs. area trade-offs between those operations. Therefore, to obtain high performance and high efficiency, we decided to implement this loop in a pipelined fashion with four pipeline stages for the main operations of:

1. generating $\tilde{\mathbf{A}}_i$ and $\tilde{\mathbf{B}}_i$ (line 15 to line 18 in Algorithm 1) (or) generating μ_i and ν_i (line 20 to line 23 in Algorithm 2),
2. computing the π operation (line 19 in Algorithm 1 or line 24 in Algorithm 2),
3. computing the `SF` operation (line 20 in Algorithm 1 or line 25 in Algorithm 2),
4. and by pulling the absorb operation of SHAKE256 for the XOF operation (line 23 in Algorithm 1 or line 28 in Algorithm 2) into the loop.

Stage 1: This stage consists of operations involved in lines 15 to 18 of Algorithm 1 and lines 20 to 23 and lines 14 to 18 of Algorithm 2. In signing, first a leaf seed (σ_i) concatenated with the salt (α) and the iteration number (i) is extended into two seeds $\sigma_{\tilde{A}_i}$, $\sigma_{\tilde{B}_i}$ using the `XOF` module. Then $\sigma_{\tilde{A}_i}$ and $\sigma_{\tilde{B}_i}$ are used to generate invertible matrices $\tilde{A}_i$ and $\tilde{B}_i$. In our hardware design, we accomplish the generation of $\sigma_{\tilde{A}_i}$ and $\sigma_{\tilde{B}_i}$ using the `SHAKE256` module and the `XOF` interface module by selecting the "XOF mode". The expanded seeds are stored in `sigmaAB` BRAM. Then, the `ExpandInvMat` module uses the seed $\sigma_{\tilde{A}_i}$ and the `XOF` interface and `SHAKE256` module by selecting the "ExpandInvMat" mode. This generates data for matrix $\tilde{A}_i$ and the `Systemizer` module inside the `ExpandInvMat` module is used to check if the matrix is invertible or not. If it is not invertible then, the module draws another matrix from the current `SHAKE256` state and performs another check for invertability of the matrix. This process is repeated until an invertible matrix is found. Once it is ensured that the matrix is invertible then $\tilde{B}_i$ is generated in similar fashion. We note that, for each iteration i, we store the related σ_i value in a BRAM inside Stage 1 for later use in computing μ_i and ν_i matrices.

In verification, if the value of $h_i = 0$ then all operations in stage 1 are similar to signing except here we use $\sigma_{\hat{A}_i}$ and $\sigma_{\hat{B}_i}$ and generate invertible matrices μ_i and ν_i. And in the case of $h_i > 0$, we skip the sampling part inside the `ExpandInvMat` and instead copy the μ_i and ν_i from the input into the BRAM where sampled μ_i and ν_i are stored and check for their invertability. In case the μ_i or ν_i are non-invertible, then a global "systemization fail" signal is generated, which will trigger "invalid signature" output.

Stage 2: This stage consists of the π operation (i.e., line 19 of Algorithm 1 and line 24 of Algorithm 2). In signing, this operation involves matrix multiplication of rows of $\mathbf{G}_0 \in \mathbb{F}_q^{k \times mn}$ with matrix $\tilde{\mathbf{A}}_i \in \mathbb{F}_q^{m \times m}$ and with $\tilde{\mathbf{B}}_i \in \mathbb{F}_q^{n \times n}$. As shown in Figure 3, in our hardware design, this operation is accomplished by the `Pi` module using $\mathbf{G}_0$ from `G0 BRAM` and $\tilde{\mathbf{A}}_i$, $\tilde{\mathbf{B}}_i$ (generated in Stage 1). The operation of the `Pi` module is described in detail in Section 4.5.

For verification, similar operations are performed as for signing except that $\mathbf{G}_{hi}$, μ_i, and ν_i are being processed in place of $\mathbf{G}_0$, $\tilde{\mathbf{A}}_i$, and $\tilde{\mathbf{B}}_i$.

Stage 3: This stage consists of operation `SF` and the checking if the $\tilde{\mathbf{G}}_i$ has systematic form (i.e., line 20 of Algorithm 1 or line 25 of Algorithm 2). We use a `Systemizer` module (described in Section 4.4) to compute the systematic form. As described in Section 4.4, there is a possibility that the input matrix does not have a systematic form. We describe how we handle this situation in detail below. This step is the same for sign and verify with corresponding inputs.

Stage 4: This stage consists of operations for compressing and hashing matrices $\tilde{\mathbf{G}}_i$ together with the message `msg` (i.e., line 23 of Algorithm 1). Although the compress and hash operation are not part of the for loop in the algorithm, in our hardware design we move the compression and the hash-absorb operation inside the for loop. This avoids the need to store all $\tilde{\mathbf{G}}_i$ values in the memory, which would require a significant amount of BRAM storage on the FPGA. For this, we use a dedicated `SHAKE256` module for Stage 4. This `SHAKE256` module digests each $\tilde{\mathbf{G}}_i$ and maintains the hash state. Hence, we only need to store one $\tilde{\mathbf{G}}_i$ in the memory at a given time. This step is the same for sign and verify.

After finishing all t iterations of the loop, the message input is absorbed into the state as well and finally we squeeze the hash state to compute the hash value d.

Pipeline Control and Memory Buffering. The pipelined part of our hardware design is highlighted in blue in Figure 3. The pipelining in our hardware design varies from the traditional register-based pipelining: Since the data we move from one stage to another are relatively large matrices, we use BRAMs as buffers between the pipeline stages. These memory buffers are shown as blush pink blocks in Figure 3. The `MemCopy` modules copy the data from the internal memories of each stage and move the data to the memory buffers, which can then be consumed in the next stage.

As the workload of each stage is different, the number of clock cycles taken by each stage is also different. Therefore, to balance the clock cycles taken by each stage, we introduce the following five performance parameters:

S_1: Stage 1 performance parameter that corresponds to the performance parameter p_{sys} (see Section 4.4) of the matrix systemizer for checking the invertability of $\tilde{A}_i$ and $\tilde{B}_i$ in the sign operation respective μ_i and ν_i in verify. S_1 controls the number of rows and columns in the processor array used inside the matrix systemizer.

S_2: Stage 2 performance parameter that corresponds to the performance parameter $p_{\pi\text{mm}}$ (see Section 4.5) of the matrix multiplication unit inside the `Pi` module. S_2 controls the vector width inside the matrix multiplication unit.

S_{mat}: Stage 2 performance parameter that corresponds to the performance parameter $p_{\pi\text{pm}}$ of the `Pi` module used inside stage 2. S_{mat} controls number of matrix multiplications performed in parallel.

S_3: Stage 3 performance parameter that corresponds to the performance parameter p_{sys} (see Section 4.4) of the large $k \times m \cdot n$ matrix systemizer used inside stage 3 for computing the systematic form of $\tilde{G}_i$ in sign respective $\hat{G}_i$ in verify. S_3 controls the number of rows and columns in the processor array used inside the matrix systemizer.

S_4: Stage 4 performance parameter that corresponds to the shifter width used inside the compress and hash module (see Section 4.8). Larger S_4 leads to fewer memory accesses while compressing and hashing the matrices $\tilde{G}_i$ respective $\hat{G}_i$.

Choosing suitable performance parameters. Recall that the goal behind choosing the performance parameters is to balance the pipeline stages so that all the modules in the pipeline are busy with their respective workloads. Let us look at the methodology we followed using the example (shown in Table 7) for the Security Level I parameter sets: Firstly, we fix the value of $S_{mat} = 1$ and then we assign all possible values to S_1, S_2, S_3, and S_4 based on the MEDS parameters up to $m = n = k = 14$ as shown in Table 7. To further lower the latency of stage 2, the S_{mat} parameter can be tweaked. The latencies reported in Table 7 also include the clock cycles required for buffering data from one stage to another. Based on these latencies, we chose a value for each S_1, S_2, S_3, and S_4 to meet our optimization goal (performance or resource consumption),

As an example for parameters with moderate resource requirement, we can select $S_1 = 1, S_2 = 7, S_3 = 6$, and $S_4 = 6$ (marked in Table 7) such that the cycles counts are roughly equal. Setting $S_{mat} = 4$ in this example brings down the cycle count of stage 2 to about $12\,132/4 = 3033$ (plus the unaffected overhead for data movement between the stages).

Based on this method, we propose different configurations for specific parameter sets as well as for a unified design that aim at balancing the pipeline stages in all parameter sets as shown in Table 8. We propose two variants of configurations: 1) a balanced design choice and 2) a high performance design choice as shown in Table 8. The choice to parameters is not just limited to the combinations provided in Table 8. Several such parameter combinations can be constructed based on resources available on the target device. We note that when

Table 7: Latencies of all pipeline stages for the Security Level 1 parameter sets with performance parameter $S_{mat} = 1$.

$S_1 = S_2 = S_3 = S_4$	Stage 1	Stage 2	Stage 3	Stage 4
1	3 534	83 364	48 191	16 599
2	1 432	41 812	14 877	8 521
3	1 142	29 926	8 350	5 883
4	1 104	23 990	5 702	4 503
5	1 042	18 054	4 020	3 751
6	1 120	18 054	3 557	3 178
7	980	12 132	2 477	2 782
8	1 032	12 118	2 294	2 522
9	1 084	12 118	2 153	2 484
10	1 136	12 118	2 054	2 371
11	1 188	12 118	1 955	2 652
12	1 240	12 118	1 912	2 197
13	1 292	12 118	1 869	2 137
14	975	6 195	1 643	1 796

Table 8: Configuration table showing selected configurations for $S_1, \ldots, S_4, S_{mat}$.

Design Choice	**Parameter Set**	q	S_1	S_2	S_3	S_4	S_{mat}
Balanced	Level I — MEDS-9923 & MEDS-13220	4 093	1	7	6	6	3
	Level III — MEDS-41711 & MEDS-69497	4 093	1	11	8	9	5
	Level V — MEDS-134180 & MEDS-167717	2 039	1	10	10	12	5
	Unified Level I–V	both	1	10	10	12	5
High Perf.	Level I — MEDS-9923 & MEDS-13220	4 093	2	14	14	14	4
	Level III — MEDS-41711 & MEDS-69497	4 093	2	11	22	22	8
	Level V — MEDS-134180 & MEDS-167717	2 039	2	15	14	15	8
	Unified Level I–V	both	2	14	14	14	4

selecting the parameters for a unified design, to avoid resource wastage, we limit the values of parameters based on m, n, and k values of smallest parameter set.

Data flow. To cope with the different ready-times of each stage that results from the different cycle counts, we resort to a handshake-based control mechanism to control the flow of data between the stages. After each stage is done with its computation, it sends a handshake signal to the next stage to indicate that the output data is ready for consumption and waits for the response from the next stage. This handshake mechanism is handled by the `PipelineController` module shown in Figure 3.

As mentioned above in Stage 3, an additional challenge in the pipeline design is the handling of possible systemization failures in some iteration i in Stage 3. In this case, we need to repeat the operations of the previous stages starting again at iteration i. We flush all the data from Stages 1 and 2, which are working on data that belongs to iterations $i + 1$ and $i + 2$, restart Stage 1 from iteration i, and step by step refill the pipeline. Stage 4 is not affected by this as it is working on data that belongs to iteration $i - 1$. The reason we restart from iteration i is because of the on-the-fly `CompressHash` operation in Stage 4. The data fed into the `SHAKE256` module has to be in sequential order to produce the correct hash value. This means that we would need memory buffers after Stage 3 to store data related to iterations $i + 1$ and $i + 2$ and wait until data from iteration i becomes ready to be loaded into module `CompressHash`. We avoid this expensive memory buffering by simply restarting the pipeline. As the failure probability of the systemizer is low (as specified in Section 4.4), the overhead of restarting the pipeline is marginal. To restart Stage 1 with iteration i, Stage 1 requires to backup the seed for i, i.e., σ_i generated in line 16 of Algorithm 1. After Stage 3 has completed successfully, the stored σ_i is discarded. The logic related to flushing and restaring is also handled by the `PipelineController` module.

Other operations. After the loop is finished, the `CompressHash` module requests the message `msg` as data stream from input port `message_in` and the message is loaded into the `SHAKE256` module to compute the hash value d. Following that, for verification, this d value is compared against the input d value to ensure the verfication of the signature. Where as for signing, the value d is then loaded into the `ParseHash` module shown in Figure 3 and described in Section 4.9, which parses the hash value and generates the vector h. The `ParseHash` module also captures i values where $h_i > 0$ while generating t. This is useful for the following computation of μ_i and ν_i. Rather than iterating over all t values, we only iterate over the w indices where $h_i > 0$. To compute matrices μ_i and ν_i, We first generate the matrices $\tilde{\mathbf{A}}_i$ and $\tilde{\mathbf{B}}_i$ using the σ_i seeds stored in Stage 1 and by reusing the logic from Stage 1. The reason for regenerating the matrices $\tilde{\mathbf{A}}_i$ and $\tilde{\mathbf{B}}_i$ is to avoid storing the t - $n \times n$ matrices and t - $m \times m$ matrices, which would otherwise consume a significant amount of BRAM. Therefore, we opted for regenerating the matrices instead. Then, we reuse the `MatMul` units of the `Pi` module. The result is then stored in BRAMs `mu` and `nu` respectively. If the `Pi` module is configured to use multiple matrix multiplication units, the `MuNuComputation` module computes μ_i and ν_i matrices in parallel. In parallel to the computation of μ_i and ν_i, the h vector along with ρ and α are loaded into the `SeedTreeToPath` module, which generates the seed path. The μ_i, ν_i, p, d, α values finally can be accessed by a top-level module through the output ports `mu`, `nu`, `path`, `d`, and `alpha` respectively as shown in Figure 3.

5 Evaluation

Table 9 shows the time and area results for our MEDS signing and verification hardware design for all parameter sets targeting an AMD `xc7a200t-3` FPGA. We note that the maximum clock frequency is limited by two different factors: 1) In case of MEDS-9923, the critical path lies inside the duplicate detection logic inside the `ParseHash` module described in Section 4.9. We use a bit-vector mapping technique to perform the duplicate detection and since the value t is large in case of MEDS-9923, the fully combinatorial variable shifter and comparator is quite large. 2) In all other cases, the critical path lies inside the sampling unit of `XOF` interface (shown in Figure 3).

As discussed in Section 4.10, we propose two different design variants for our implementation: “Balanced” and “High Performance”. The results for these choices are presented in Table 9. While the high performance design requires only 4.5 ms to 29.5 ms for signing and a similar amount of time for verification, it also takes up significant resources, mainly in terms of BRAM utilization. We note that while the storage required for the data remains the same in both the design choices, due to the wider memory word width in the High Performance variant, the synthesis tool needs to use multiple BRAM units.

Additionally, for each of the design variants, the design allows further flexibility to be synthesized for a specific parameter set or for a unified design that allows us to choose any of the six different parameter choices at run-time. While the unified design does not impact the cycles taken for signing and verification compared to specific designs, it does have an impact on resource utilization. However, it can be seen from Table 9 that the resource utilization of the unified design is close to the utilization of the parameter set specific designs for the biggest parameter sets (i.e., MEDS-134180 and MEDS-167717).

We note that to the best of our knowledge this is the first and only MEDS hardware implementation. Therefore, our primary comparison is with the optimized software implementation provided along with the MEDS specification [CNP$^+$23a]. We note that while our design is running at a frequency range of 115 MHz to 133 MHz, the software implementations results are reported for a CPU running at a frequency of 1.9 GHz. We note that both of our design choices outperform the optimized software implementation by a significant margin.

Comparison to Related Work. Table 10 presents a comparison to a number of other PQC signature schemes that have been presented in literature. We are unaware of other MEDS hardware implementations, so no other MEDS works are included in the table. Many works present data for the same XC7A100T FPGA that we use and they generally report similar frequencies as we. Specific comparisons with these schemes to our implementation is difficult as different schemes are based on different mathematical problems. Further, some related works use high-end FPGAs, which generally give much better performance.

Compared to the hardware (co-)designs of some other PQC schemes shown in Table 10, our hardware implementation of MEDS is in the middle field and requires a medium amount of resources. Some of the other designs are much more efficient than our implementation, however this is due to the high computational cost inherent to the MEDS specification in particular compared to lattice-based schemes. Nevertheless, the performance our implementation is comparable to several other PQC schemes such as XMSS, SPHINCS+/SLH-DSA, SDitH, and Raccoon. We believe that this shows that the quality of our implementation is on par with other work in this field.

6 Conclusion

In the introduction in Section 1, we raised three research questions:

Research Question Q1 asks if there is sufficient inherent parallelism in MEDS to speed up the sign and verify operations. Given that there is plenty parallelism on the low level multiplying and systemizing matrices as well as on the high level iterating over t independent computations, there is indeed ample opportunity to accelerate MEDS. We provide performance parameters to control the low level parallelization and we pipeline the main loop, achieving a significant speed up for MEDS signing and verification. For e.g., for security level I, our balanced design achieves a speed-up of 4-5× for both signing and verification times, whereas our high-performance design achieves a speed-up of 9-10× for both signing and verification times when compared to the optimized software reference implementation. Notably, these gains are achieved while our design operates in the frequency range of 115 MHz to 132 MHz, whereas the optimized software implementation runs at 1.9 GHz.

However, Q1 also asks about the resource cost of accelerating MEDS. Since the computational cost of MEDS sign and verification is significant, selecting large performance parameters for a high-performance design results in high resource cost.

Research Question Q2 asks to what extend resources can be shared between the sign and verify operations in a shared design implementing both operations. Since the main loop operates very similar in both sign and verify, the resources of sign can be reused by verify with only little control logic overhead. Sign requires some additional computations at the end to compute the responses for the signature, which can simply be skipped by verify.

Research Question Q3 asks about the overhead of supporting all parameter sets selectable at runtime in a single joint design. The overhead for additional control logic for supporting different matrix dimensions and challenge lengths is marginal. To our surprise, supposing arithmetic in multiple prime fields also comes at only a small overhead as for the fields specified in MEDS, most resources can be shared between both fields. Only slightly more logic than needed for the larger field is required to support both fields.

We conclude that — if the fields are chosen carefully — supporting multiple fields in a joint hardware implementation is very much feasible. Specifically our results show that arithmetic for pseudo-Mersenne primes with small Hamming distance can be implemented jointly with small overhead.

References

ALCZ20. Dorian Amiet, Lukas Leuenberger, Andreas Curiger, and Paul Zbinden. Fpga-based sphincs$^+$ implementations: Mind the glitch. In *23rd Euromicro Conference on Digital System Design, DSD 2020, Kranj, Slovenia, August 26-28, 2020*, pages 229–237. IEEE, 2020.

AMI$^+$23. Aikata Aikata, Ahmet Can Mert, Malik Imran, Samuel Pagliarini, and Sujoy Sinha Roy. Kali: A crystal for post-quantum security using kyber and dilithium. *IEEE Transactions on Circuits and Systems I: Regular Papers*, 70(2):747–758, 2023.

BCH$^+$23. Ward Beullens, Ming-Shing Chen, Shih-Hao Hung, Matthias J. Kannwischer, Bo-Yuan Peng, Cheng-Jhih Shih, and Bo-Yin Yang. Oil and vinegar:

Modern parameters and implementations. *IACR TCHES*, 2023(3):321–365, 2023.

BFV13. Charles Bouillaguet, Pierre-Alain Fouque, and Amandine Véber. Graph-theoretic algorithms for the “isomorphism of polynomials” problem. In Thomas Johansson and Phong Q. Nguyen, editors, *EUROCRYPT 2013*, volume 7881 of *LNCS*, pages 211–227. Springer, Heidelberg, May 2013.

BNG21. Luke Beckwith, Duc Tri Nguyen, and Kris Gaj. High-performance hardware implementation of crystals-dilithium. In *International Conference on Field-Programmable Technology, (IC)FPT 2021, Auckland, New Zealand, December 6-10, 2021*, pages 1–10. IEEE, 2021.

CCD$^+$22. Po-Jen Chen, Tung Chou, Sanjay Deshpande, Norman Lahr, Ruben Niederhagen, Jakub Szefer, and Wen Wang. Complete and improved FPGA implementation of classic McEliece. *IACR TCHES*, 2022(3):71–113, 2022.

CNP$^+$23a. Tung Chou, Ruben Niederhagen, Edoardo Persichetti, Lars Ran, Tovohery Hajatiana Randrianarisoa, Krijn Reijnders, Simona Samardjiska, and Monika Trimoska. MEDS — Matrix Equivalence Digital Signature. Technical report, National Institute of Standards and Technology, 2023. available at `https://csrc.nist.gov/Projects/pqc-dig-sig/round-1-additional-signatures`.

CNP$^+$23b. Tung Chou, Ruben Niederhagen, Edoardo Persichetti, Tovohery Hajatiana Randrianarisoa, Krijn Reijnders, Simona Samardjiska, and Monika Trimoska. Take your MEDS: Digital signatures from matrix code equivalence. In Nadia El Mrabet, Luca De Feo, and Sylvain Duquesne, editors, *AFRICACRYPT 23*, volume 14064 of *LNCS*, pages 28–52. Springer Nature, July 2023.

CNRS24. Tung Chou, Ruben Niederhagen, Lars Ran, and Simona Samardjiska. Reducing signature size of matrix-code-based signature schemes. In Markku-Juhani O. Saarinen and Daniel Smith-Tone, editors, *Post-Quantum Cryptography - 15th International Workshop, PQCrypto 2024, Oxford, UK, June 12-14, 2024, Proceedings, Part I*, volume 14771 of *Lecture Notes in Computer Science*, pages 107–134. Springer, 2024.

DHSY24. Sanjay Deshpande, James Howe, Jakub Szefer, and Dongze Yue. Sdith in hardware. *IACR Trans. Cryptogr. Hardw. Embed. Syst.*, 2024(2):215–251, 2024.

DLK$^+$25. Sanjay Deshpande, Yongseok Lee, Cansu Karakuzu, Jakub Szefer, and Yunheung Paek. Sphincslet: An area-efficient accelerator for the full sphincs+ digital signature algorithm. *ACM Trans. Embed. Comput. Syst.*, April 2025. Just Accepted.

dPRS23. Rafaël del Pino, Thomas Prest, Mélissa Rossi, and Markku-Juhani O. Saarinen. High-order masking of lattice signatures in quasilinear time. In *2023 IEEE Symposium on Security and Privacy*, pages 1168–1185. IEEE Computer Society Press, May 2023.

Dwo15. Morris J Dworkin. Sha-3 standard: Permutation-based hash and extendable-output functions. 2015.

DXN$^+$23. Sanjay Deshpande, Chuanqi Xu, Mamuri Nawan, Kashif Nawaz, and Jakub Szefer. Fast and efficient hardware implementation of HQC. In Claude Carlet, Kalikinkar Mandal, and Vincent Rijmen, editors, *Selected Areas in Cryptography - SAC 2023 - 30th International Conference, Fredericton, Canada, August 14-18, 2023, Revised Selected Papers*, volume 14201 of *Lecture Notes in Computer Science*, pages 297–321. Springer, 2023.

FG18. Ahmed Ferozpuri and Kris Gaj. High-speed fpga implementation of the nist round 1 rainbow signature scheme. In *2018 International Conference on ReConFigurable Computing and FPGAs (ReConFig)*, pages 1–8, 2018.

FS87. Amos Fiat and Adi Shamir. How to prove yourself: Practical solutions to identification and signature problems. In Andrew M. Odlyzko, editor, *CRYPTO'86*, volume 263 of *LNCS*, pages 186–194. Springer, Heidelberg, August 1987.

GQT21. Joshua A. Grochow, Youming Qiao, and Gang Tang. Average-case algorithms for testing isomorphism of polynomials, algebras, and multilinear forms. In Markus Bläser and Benjamin Monmege, editors, *38th International Symposium on Theoretical Aspects of Computer Science, STACS 2021, March 16-19, 2021, Saarbrücken, Germany (Virtual Conference)*, volume 187 of *LIPIcs*, pages 38:1–38:17. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2021.

HGG07. William Hasenplaugh, Gunnar Gaubatz, and Vinodh Gopal. Fast modular reduction. In *18th IEEE Symposium on Computer Arithmetic (ARITH-18 2007), 25-27 June 2007, Montpellier, France*, pages 225–229. IEEE Computer Society, 2007.

HQR89. Bertrand Hochet, Patrice Quinton, and Yves Robert. Systolic gaussian elimination over gf(p) with partial pivoting. *IEEE Trans. Computers*, 38(9):1321–1324, 1989.

HSK$^+$23. Florian Hirner, Michael Streibl, Florian Krieger, Ahmet Can Mert, and Sujoy Sinha Roy. Whipping the MAYO signature scheme using hardware platforms. Cryptology ePrint Archive, Paper 2023/1267, 2023. `https://eprint.iacr.org/2023/1267`.

LSG21. Georg Land, Pascal Sasdrich, and Tim Güneysu. A hard crystal - implementing dilithium on reconfigurable hardware. In Vincent Grosso and Thomas Pöppelmann, editors, *Smart Card Research and Advanced Applications - 20th International Conference, CARDIS 2021, Lübeck, Germany, November 11-12, 2021, Revised Selected Papers*, volume 13173 of *Lecture Notes in Computer Science*, pages 210–230. Springer, 2021.

MR24. Suraj Mandal and Debapriya Basu Roy. Kid: A hardware design framework targeting unified ntt multiplication for crystals-kyber and crystals-dilithium on fpga. In *2024 37th International Conference on VLSI Design and 2024 23rd International Conference on Embedded Systems (VLSID)*, pages 455–460, 2024.

NQT24. Anand Kumar Narayanan, Youming Qiao, and Gang Tang. Algorithms for matrix code and alternating trilinear form equivalences via new isomorphism invariants. In Marc Joye and Gregor Leander, editors, *Advances in Cryptology - EUROCRYPT 2024 - 43rd Annual International Conference on the Theory and Applications of Cryptographic Techniques, Zurich, Switzerland, May 26-30, 2024, Proceedings, Part III*, volume 14653 of *Lecture Notes in Computer Science*, pages 160–187. Springer, 2024.

Saa24. Markku-Juhani O. Saarinen. Accelerating SLH-DSA by two orders of magnitude with a single hash unit. *IACR Cryptol. ePrint Arch.*, page 367, 2024. To appear in CRYPTO 2024, August, 2024.

SAW$^+$23. Michael Schmid, Dorian Amiet, Jan Wendler, Paul Zbinden, and Tao Wei. Falcon takes off - A hardware implementation of the falcon signature scheme. *IACR Cryptol. ePrint Arch.*, page 1885, 2023.

SMA$^+$24. Oussama Sayari, Soundes Marzougui, Thomas Aulbach, Juliane Krämer, and Jean-Pierre Seifert. Hamayo: A fault-tolerant reconfigurable hardware implementation of the MAYO signature scheme. In Romain Wacquez and Naofumi Homma, editors, *Constructive Side-Channel Analysis and Secure Design - 15th International Workshop, COSADE 2024, Gardanne, France, April 9-10, 2024, Proceedings*, volume 14595 of *Lecture Notes in Computer Science*, pages 240–259. Springer, 2024.

SWM$^+$10. Abdulhadi Shoufan, Thorsten Wink, H. Gregor Molter, Sorin A. Huss, and Eike Kohnert. A novel cryptoprocessor architecture for the mceliece public-key cryptosystem. *IEEE Trans. Computers*, 59(11):1533–1546, 2010.

TDJ$^+$22. Gang Tang, Dung Hoang Duong, Antoine Joux, Thomas Plantard, Youming Qiao, and Willy Susilo. Practical post-quantum signature schemes from isomorphism problems of trilinear forms. In Orr Dunkelman and Stefan Dziembowski, editors, *EUROCRYPT 2022, Part III*, volume 13277 of *LNCS*, pages 582–612. Springer, Heidelberg, May / June 2022.

TYD$^+$11. Shaohua Tang, Haibo Yi, Jintai Ding, Huan Chen, and Guomin Chen. High-speed hardware implementation of rainbow signature on fpgas. In Bo-Yin Yang, editor, *Post-Quantum Cryptography*, pages 228–243, Berlin, Heidelberg, 2011. Springer Berlin Heidelberg.

WJW$^+$19. Wen Wang, Bernhard Jungk, Julian Wälde, Shuwen Deng, Naina Gupta, Jakub Szefer, and Ruben Niederhagen. XMSS and embedded systems. In Kenneth G. Paterson and Douglas Stebila, editors, *SAC 2019*, volume 11959 of *LNCS*, pages 523–550. Springer, Heidelberg, August 2019.

WSN16. Wen Wang, Jakub Szefer, and Ruben Niederhagen. Solving large systems of linear equations over GF(2) on fpgas. In Peter M. Athanas, René Cumplido, Claudia Feregrino, and Ron Sass, editors, *International Conference on ReConFigurable Computing and FPGAs, ReConFig 2016, Cancun, Mexico, November 30 - Dec. 2, 2016*, pages 1–7. IEEE, 2016.

WSN17. Wen Wang, Jakub Szefer, and Ruben Niederhagen. FPGA-based key generator for the niederreiter cryptosystem using binary goppa codes. In Wieland Fischer and Naofumi Homma, editors, *CHES 2017*, volume 10529 of *LNCS*, pages 253–274. Springer, Heidelberg, September 2017.

ZZC$^+$23. Yihong Zhu, Wenping Zhu, Chen Chen, Min Zhu, Zhengdong Li, Shaojun Wei, and Leibo Liu. Mckeycutter: A high-throughput key generator of classic mceliece on hardware. In *60th ACM/IEEE Design Automation Conference, DAC 2023, San Francisco, CA, USA, July 9-13, 2023*, pages 1–6. IEEE, 2023.

ZZL$^+$23. Yihong Zhu, Wenping Zhu, Chongyang Li, Min Zhu, Chenchen Deng, Chen Chen, Shuying Yin, Shouyi Yin, Shaojun Wei, and Leibo Liu. Repqc: A 3.4-uj/op 48-kops post-quantum crypto-processor for multiple-mathematical problems. *IEEE Journal of Solid-State Circuits*, 58(1):124–140, 2023.

Appendix

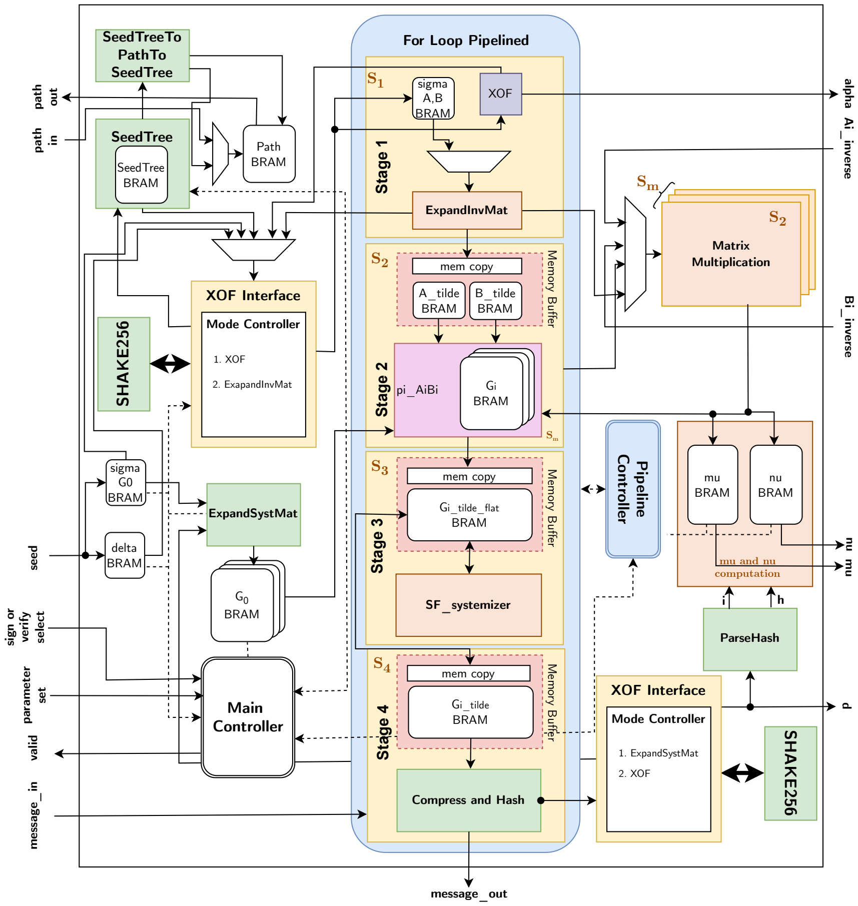


Fig. 3: High-level design overview of the MEDS signing operation.

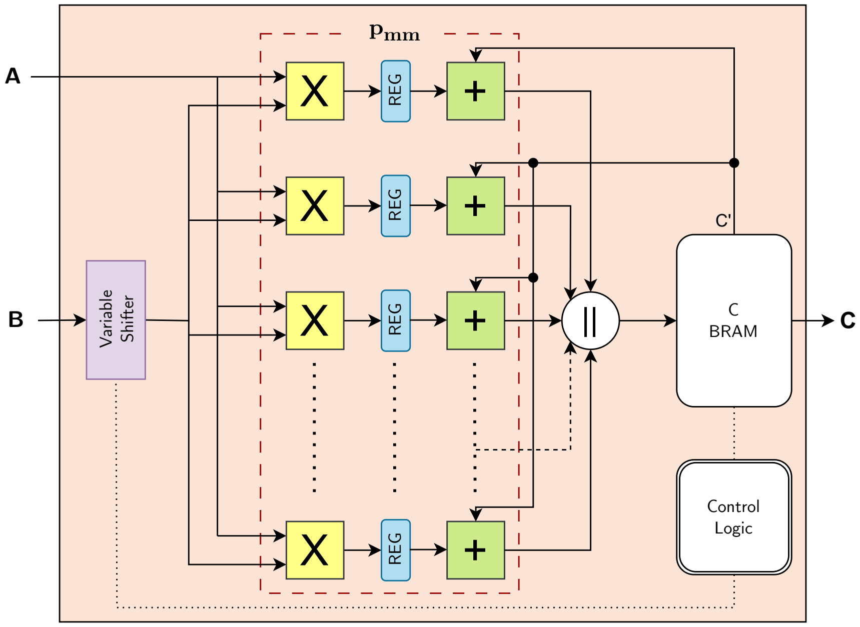


Fig. 4: Hardware design of the matrix multiplication module.

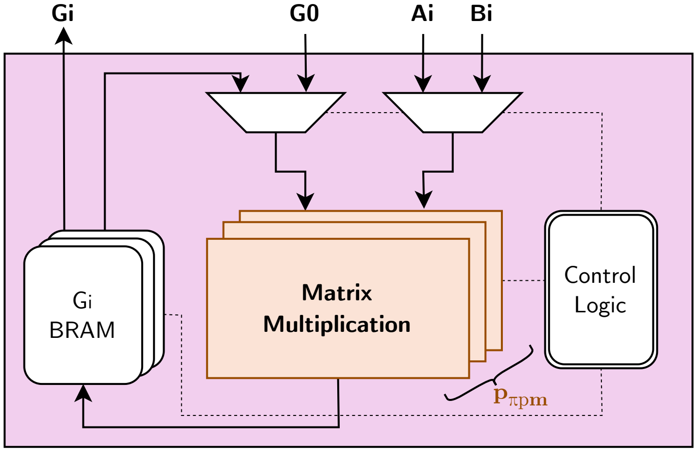


Fig. 5: Hardware design of the π module.

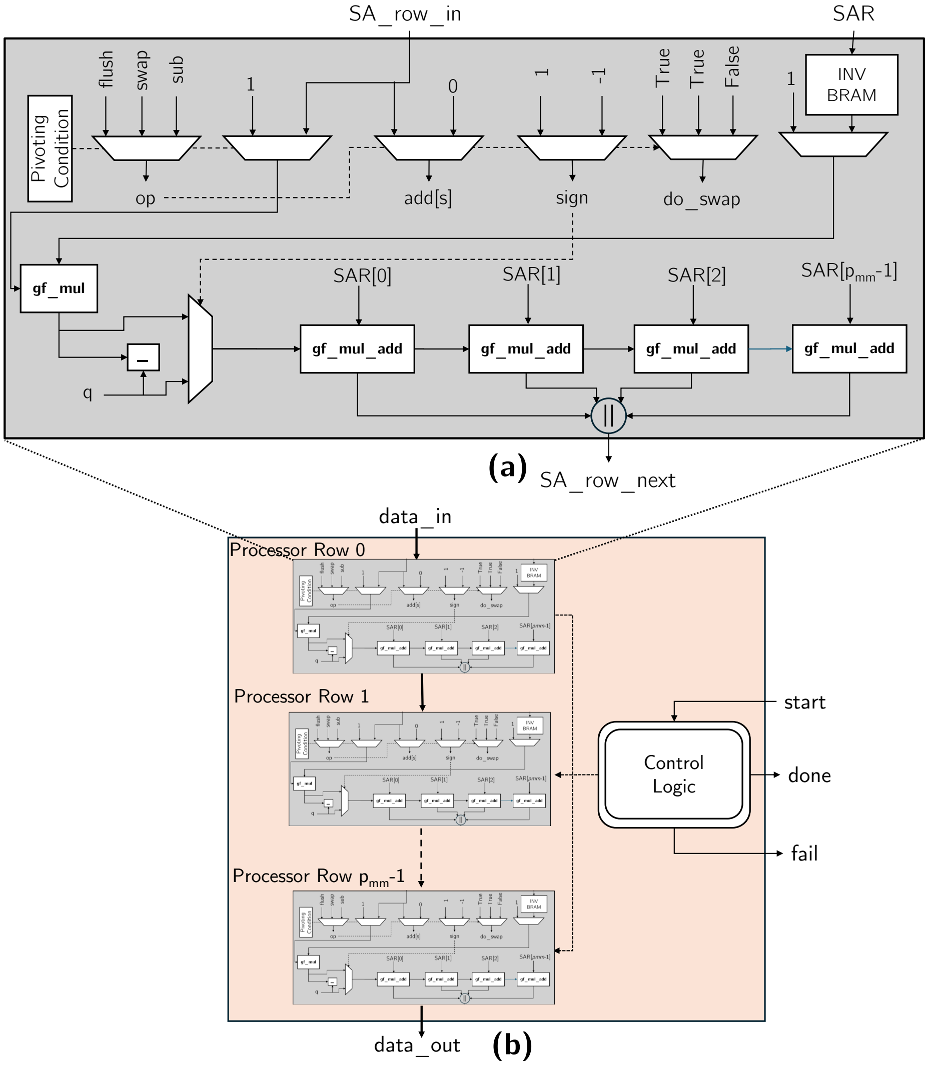


Fig. 6: Systemizer module with p_{sys} vector ALUs of width p_{sys}.

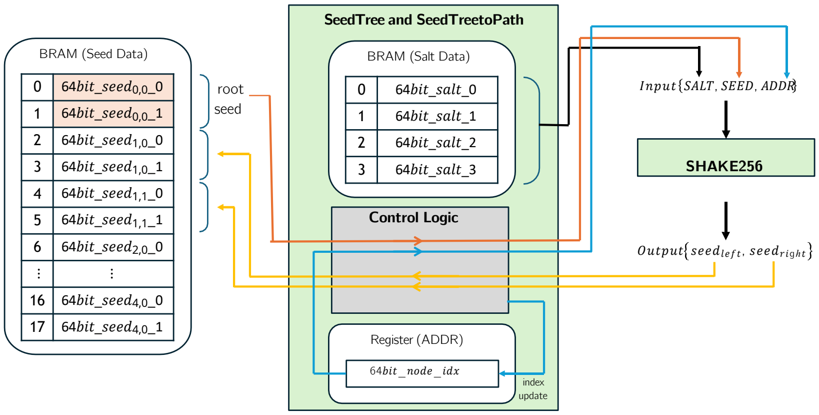


Fig. 7: `SeedTree` module architecture with SHAKE256.

Algorithm 1: MEDS sign (from [CNP+23a])

```
   Input: secret key $\mathsf{sk} \in \mathcal{B}^{\ell_{\mathsf{sk}}}$, message $\mathsf{msg} \in \mathcal{B}^{\ell_{\mathsf{msg}}}$
   Output: signed message $\mathsf{msg}_{\mathsf{sig}} \in \mathcal{B}^{\ell_{\mathsf{sig}}+\ell_{\mathsf{msg}}}$
1  $f_{\mathsf{sk}} \leftarrow \ell_{\mathsf{sec\_seed}}$;
2  $\sigma_{\mathbf{G}_0} \leftarrow \mathsf{sk}[f_{\mathsf{sk}}, f_{\mathsf{sk}} + \ell_{\mathsf{pub\_seed}} - 1]$;
3  $f_{\mathsf{sk}} \leftarrow f_{\mathsf{sk}} + \ell_{\mathsf{pub\_seed}}$;
4  $\mathbf{G}_0 \in \mathbb{F}_q^{k \times mn} \leftarrow \mathsf{ExpandSystMat}(\sigma_{\mathbf{G}_0})$;
5  forall $i \in \{1, \ldots, s-1\}$ do
6      $\mathbf{A}_i^{-1} \in \mathbb{F}_q^{m \times m} \leftarrow \mathsf{Decompress}(\mathsf{sk}[f_{\mathsf{sk}}, f_{\mathsf{sk}} + \ell_{\mathbb{F}_q^{m \times m}}], m, m)$;
7      $f_{\mathsf{sk}} \leftarrow f_{\mathsf{sk}} + \ell_{\mathbb{F}_q^{m \times m}}$;
8  forall $i \in \{1, \ldots, s-1\}$ do
9      $\mathbf{B}_i^{-1} \in \mathbb{F}_q^{n \times n} \leftarrow \mathsf{Decompress}(\mathsf{sk}[f_{\mathsf{sk}}, f_{\mathsf{sk}} + \ell_{\mathbb{F}_q^{n \times n}}], n, n)$;
10     $f_{\mathsf{sk}} \leftarrow f_{\mathsf{sk}} + \ell_{\mathbb{F}_q^{n \times n}}$;
11 $\delta \in \mathcal{B}^{\ell_{\mathsf{sec\_seed}}} \leftarrow \mathsf{Randombytes}(\ell_{\mathsf{sec\_seed}})$;
12 $\rho \in \mathcal{B}^{\ell_{\mathsf{tree\_seed}}}, \alpha \in \mathcal{B}^{\ell_{\mathsf{salt}}} \leftarrow \mathsf{XOF}(\delta, \ell_{\mathsf{tree\_seed}}, \ell_{\mathsf{salt}})$;
13 $\sigma_0, \ldots, \sigma_{t-1} \in \mathcal{B}^{\ell_{\mathsf{tree\_seed}}} \leftarrow \mathsf{SeedTree}_t(\rho, \alpha)$;
14 forall $i \in \{0, \ldots, t-1\}$ do
15     $\sigma'_i \in \mathcal{B}^{\ell_{\mathsf{salt}}+\ell_{\mathsf{tree\_seed}}+4} \leftarrow \Big(\alpha | \sigma_i | \mathsf{ToBytes}(2^{1+\lceil \log_2(t) \rceil} + i, 4)\Big)$;
16     $\sigma_{\tilde{\mathbf{A}}_i}, \sigma_{\tilde{\mathbf{B}}_i} \in \mathcal{B}^{\ell_{\mathsf{pub\_seed}}}, \sigma_i \in \mathcal{B}^{\ell_{\mathsf{tree\_seed}}} \leftarrow \mathsf{XOF}(\sigma'_i, \ell_{\mathsf{pub\_seed}}, \ell_{\mathsf{pub\_seed}}, \ell_{\mathsf{tree\_seed}})$;
17     $\tilde{\mathbf{A}}_i \in \mathrm{GL}_m(q) \leftarrow \mathsf{ExpandInvMat}(\sigma_{\tilde{\mathbf{A}}_i}, m)$;
18     $\tilde{\mathbf{B}}_i \in \mathrm{GL}_n(q) \leftarrow \mathsf{ExpandInvMat}(\sigma_{\tilde{\mathbf{B}}_i}, n)$;
19     $\tilde{\mathbf{G}}_i \in \mathbb{F}_q^{k \times mn} \leftarrow \pi_{\tilde{\mathbf{A}}_i, \tilde{\mathbf{B}}_i}(\mathbf{G}_0)$;
20     $\tilde{\mathbf{G}}_i \in \mathbb{F}_q^{k \times mn} \cup \{\perp\} \leftarrow \mathsf{SF}(\tilde{\mathbf{G}}_i)$;
21     if $\tilde{\mathbf{G}}_i = \perp$ then
22         goto line 15;
23 $d \in \mathcal{B}^{\ell_{\mathsf{digest}}} \leftarrow$
       $\mathsf{H}(\mathsf{Compress}(\tilde{\mathbf{G}}_0[; k, mn-1]) \mid \ldots \mid \mathsf{Compress}(\tilde{\mathbf{G}}_{t-1}[; k, mn-1]) \mid \mathsf{msg})$;
24 $h_0, \ldots, h_{t-1} \in \{0, \ldots, s-1\} \leftarrow \mathsf{ParseHash}_{s,t,w}(d)$;
25 $f_v \leftarrow 0$;
26 forall $i \in \{0, \ldots, t-1\}$ do
27     if $h_i > 0$ then
28         $\mu_i \in \mathbb{F}_q^{m \times m} \leftarrow \tilde{\mathbf{A}}_i \cdot \mathbf{A}_{h_i}^{-1}$;
29         $\nu_i \in \mathbb{F}_q^{n \times n} \leftarrow \mathbf{B}_{h_i}^{-1} \cdot \tilde{\mathbf{B}}_i$;
30         $v_{f_v} \in \mathcal{B}^{\ell_{\mathbb{F}_q^{m \times m}} + \ell_{\mathbb{F}_q^{n \times n}}} \leftarrow (\mathsf{Compress}(\mu_i) \mid \mathsf{Compress}(\nu_i))$;
31         $f_v \leftarrow f_v + 1$;
32 $p \in \mathcal{B}^{\ell_{\mathsf{path}}} \leftarrow \mathsf{SeedTreeToPath}_t(h_0, \ldots, h_{t-1}, \rho, \alpha)$;
33 return $\mathsf{msg}_{\mathsf{sig}} \in \mathcal{B}^{w(\ell_{\mathbb{F}_q^{m \times m}} + \ell_{\mathbb{F}_q^{n \times n}}) + \ell_{\mathsf{path}} + \ell_{\mathsf{digest}} + \ell_{\mathsf{salt}} + \ell_{\mathsf{msg}} = \ell_{\mathsf{sig}} + \ell_{\mathsf{msg}}} =$
       $(v_0 \mid \ldots \mid v_{w-1} \mid p \mid d \mid \alpha \mid \mathsf{msg})$;
```

Algorithm 2: MEDS verify (adapted from [CNP+23a])

```
Input: public key $\mathsf{pk} \in \mathcal{B}^{\ell_{\mathsf{pk}}}$, signed message $\mathsf{msg}_{\mathsf{sig}} \in \mathcal{B}^{\ell_{\mathsf{sig}}+\ell_m}$
Output: message $\mathsf{msg} \in \mathcal{B}^{\ell}_{m}$ or $\perp$
$\sigma_{\mathbf{G}_0} \leftarrow \mathsf{pk}[0, \ell_{\mathsf{pub\_seed}} - 1]$;
$\mathbf{G}_0 \in \mathbb{F}_q^{k \times mn} \leftarrow \mathsf{ExpandSystMat}(\sigma_{\mathbf{G}_0})$;
$f_{\mathsf{pk}} \leftarrow \ell_{\mathsf{pub\_seed}}$;
forall $i \in \{1, \ldots, s-1\}$ do
    $\mathbf{G}_i \in \mathbb{F}_q^{k \times mn} \leftarrow \mathsf{DecompressG}(\mathsf{pk}[f_{\mathsf{pk}}, f_{\mathsf{pk}} + \ell_{G_i}])$;
    $f_{\mathsf{pk}} \leftarrow f_{\mathsf{pk}} + \ell_{G_i}$;
$p \in \mathcal{B}^{\ell_{\mathsf{path}}} \leftarrow \mathsf{msg}_{\mathsf{sig}}[\ell_{\mathsf{sig}} - \ell_{\mathsf{digest}} - \ell_{\mathsf{salt}} - \ell_{\mathsf{path}}, \ell_{\mathsf{sig}} - \ell_{\mathsf{digest}} - \ell_{\mathsf{salt}} - 1]$;
$d \in \mathcal{B}^{\ell_{\mathsf{digest}}}, \alpha \in \mathcal{B}^{\ell_{\mathsf{salt}}}, \mathsf{msg} \in \mathcal{B}^{*} \leftarrow$
  $\mathsf{msg}_{\mathsf{sig}}[\ell_{\mathsf{sig}} - \ell_{\mathsf{digest}} - \ell_{\mathsf{salt}}, \ell_{\mathsf{sig}} - \ell_{\mathsf{salt}} - 1], \mathsf{msg}_{\mathsf{sig}}[\ell_{\mathsf{sig}} - \ell_{\mathsf{salt}}, \ell_{\mathsf{sig}} - 1], \mathsf{msg}_{\mathsf{sig}}[\ell_{\mathsf{sig}}, ]$;
$h_0, \ldots, h_{t-1} \in \{0, \ldots, s-1\} \leftarrow \mathsf{ParseHash}_{s,t,w}(d)$;
$\sigma_0, \ldots, \sigma_{t-1} \in \mathcal{B}^{\ell_{\mathsf{tree\_seed}}} \leftarrow \mathsf{PathToSeedTree}_t(h_0, \ldots, h_{t-1}, p, \alpha)$;
$f_{m_s} \leftarrow 0$;
forall $i \in \{0, \ldots, t-1\}$ do
    if $h_i > 0$ then
        $\mu_i \in \mathbb{F}_q^{m \times m} \leftarrow \mathsf{Decompress}(\mathsf{msg}_{\mathsf{sig}}[f_{m_s}, f_{m_s} + \ell_{\mathbb{F}_q^{m \times m}} - 1], m, m)$;
        $\nu_i \in \mathbb{F}_q^{n \times n} \leftarrow$
          $\mathsf{Decompress}(\mathsf{msg}_{\mathsf{sig}}[f_{m_s} + \ell_{\mathbb{F}_q^{m \times m}}, f_{m_s} + \ell_{\mathbb{F}_q^{m \times m}} + \ell_{\mathbb{F}_q^{n \times n}} - 1], n, n)$;
        $f_{m_s} \leftarrow f_{m_s} + \ell_{\mathbb{F}_q^{m \times m}} + \ell_{\mathbb{F}_q^{n \times n}}$;
        if $\mu_i \notin \mathrm{GL}_m(q)$ or $\nu_i \notin \mathrm{GL}_n(q)$ then
            return $\perp$;
    else
        $\sigma'_i \in \mathcal{B}^{\ell_{\mathsf{salt}}+\ell_{\mathsf{tree\_seed}}+4} \leftarrow \Big(\alpha | \sigma_i | \mathsf{ToBytes}(2^{1+\lceil \log_2(t) \rceil} + i, 4)\Big)$;
        $\sigma_{\hat{\mathbf{A}}_i}, \sigma_{\hat{\mathbf{B}}_i} \in \mathcal{B}^{\ell_{\mathsf{pub\_seed}}}, \sigma_i \in \mathcal{B}^{\ell_{\mathsf{tree\_seed}}} \leftarrow \mathsf{XOF}(\sigma'_i, \ell_{\mathsf{pub\_seed}}, \ell_{\mathsf{pub\_seed}}, \ell_{\mathsf{tree\_seed}})$;
        $\mu_i \in \mathrm{GL}_m(q) \leftarrow \mathsf{ExpandInvMat}(\sigma_{\hat{\mathbf{A}}_i}, m)$;
        $\nu_i \in \mathrm{GL}_n(q) \leftarrow \mathsf{ExpandInvMat}(\sigma_{\hat{\mathbf{B}}_i}, n)$;
    $\hat{\mathbf{G}}_i \in \mathbb{F}_q^{k \times mn} \leftarrow \pi_{\mu_i, \nu_i}(\mathbf{G}_{h_i})$;
    $\hat{\mathbf{G}}_i \in \mathbb{F}_q^{k \times mn} \cup \{\perp\} \leftarrow \mathsf{SF}(\hat{\mathbf{G}}_i)$;
    if $\hat{\mathbf{G}}_i = \perp$ then
        return $\perp$;
$d' \in \mathcal{B}^{\ell_{\mathsf{digest}}} \leftarrow$
  $\mathsf{H}(\mathsf{Compress}(\hat{\mathbf{G}}_0[; k, mn-1]) \mid \ldots \mid \mathsf{Compress}(\hat{\mathbf{G}}_{t-1}[; k, mn-1]) \mid \mathsf{msg})$;
if $d = d'$ then
    return $\mathsf{msg}$;
else
    return $\perp$;
```

	Resources				Time		
Param. Set	**Area**		**Memory**		**Freq.**	**Sign**	**Verify**
	(LUT)	(DSP)	(FF)	(BRAM)	(MHz)	(Kcyc)	(Kcyc)
Security Level I							
Balanced – $\{S_1, S_2, S_3, S_4, S_{mat}\}$ = **{1, 7, 6, 6, 3}**							
MEDS-9923	24 308	66	20 541	112	116	7 562	7 485
MEDS-13220	21 680	66	19 491	70	130	1 334	1 252
High Performance – $\{S_1, S_2, S_3, S_4, S_{mat}\}$ = **{2, 14, 14, 14, 3}**							
MEDS-9923	46 842	259	44 875	193	115	3 375	3 331
MEDS-13220	44 080	259	43 832	151	132	591	555
Security Level III							
Balanced – $\{S_1, S_2, S_3, S_4, S_{mat}\}$ = **{1, 11, 8, 9, 5}**							
MEDS-41711	34 036	130	31 243	206	123	9 820	9 537
MEDS-69497	33 178	130	30 751	187	125	2 909	2 531
High Performance – $\{S_1, S_2, S_3, S_4, S_{mat}\}$ = **{2, 11, 22, 22, 8}**							
MEDS-41711	80 525	601	91 038	205	129	6 129	5 954
MEDS-69497	79 653	601	90 544	186	129	1 805	1 575
Security Level V							
Balanced – $\{S_1, S_2, S_3, S_4, S_{mat}\}$ = **{1, 10, 10, 12, 5}**							
MEDS-134180	38 332	163	39 570	285	130	11 273	9 041
MEDS-167717	38 011	163	39 450	328	133	8 154	5 324
High Performance – $\{S_1, S_2, S_3, S_4, S_{mat}\}$ = **{2, 15, 14, 15, 8}**							
MEDS-134180	61 595	337	60 975	290	127	5 489	4 570
MEDS-167717	60 620	337	60 898	288	131	3 862	2 700
Unified Design							
Balanced – $\{S_1, S_2, S_3, S_4, S_{mat}\}$ = **{1, 10, 10, 12, 5}**							
MEDS-9923	44 818	163	43 163	344.5	113	5 228	5 151
MEDS-13220						943	861
MEDS-41711						13 416	13 119
MEDS-69497						3 870	3 471
MEDS-134180						11 273	9 040
MEDS-167717						8 154	5 323
High Performance – $\{S_1, S_2, S_3, S_4, S_{mat}\}$ = **{2, 14, 14, 14, 4}**							
MEDS-9923	72 888	273	63 453	386.5	113	3 375	3 331
MEDS-13220						591	555
MEDS-41711						10 206	10 118
MEDS-69497						2 709	2 599
MEDS-134180						11 096	10 680
MEDS-167717						6 789	6 264
Reference Software Implementation on an AMD Ryzen 7 PRO 5850U CPU [CNP+23a]							
MEDS-9923	—	—	—	—	1 900	518 050	515 580
MEDS-13220	—	—	—	—	1 900	88 900	87 480
MEDS-41711	—	—	—	—	1 900	1 467 000	1 461 970
MEDS-55604	—	—	—	—	1 900	387 270	380 700
MEDS-134180	—	—	—	—	1 900	1 629 840	1 612 570
MEDS-167717	—	—	—	—	1 900	961 800	938 890

Table 9: Comparison of the time and area for our `Sign` and `Verify` modules targeting Xilinx Artix 7 (`xc7a200t`) FPGA. (In the time-area column, t_s is the time for sign, t_v the time for verify.)

Design	Algorithm	FPGA	f-max	Sign	Veify	Resources
			(MHz)	(ms)	(ms)	(logic/FF/DSP/BR)
[BCH+23]	ov-Ip-pkc	XC7A200T	100	0.08	0.69	37k/25k/2/81
[BCH+23]	ov-V-pkc+skc	XC7A200T	100	28.57	3.93	83k/41k/4/359
[SMA+24]	MAYO 1	Arm/Zynq-7020	100	28.60	—	21k/13k/11/129
[HSK+23]	MAYO 1	Artix-7	75	0.43	0.05	106k/38k/2/45.5
[DHSY24]	SDitH L1 GF256	XC7A200T	164	41.00	52.90	17k/9k/0/164.5
[DHSY24]	SDitH L3 GF251	XC7A200T	164	276.10	183.60	34k/31k/472/521.5
[dPRS23]	Raccoon-128 2 shares	RISC-V/XC7A100T	78	30.70	18.40	10k/4k/3/—
[dPRS23]	Raccoon-128 32 shares	RISC-V/XC7A100T	78	284.10	17.86	
[WJW+19]	XMSS SHA256 $h = 10$	RISC-V/Cyclone V	145	9.95	5.80	7k/10k/—/145
[LSG21]	Dilithium-III F	XC7A100T	145	0.85	0.23	30k/11k/45/21
[BNG21]	Dilithium-V	Artix-7	116	0.21	0.12	53k/28k/16/29
[Saa24]	SLH-DSA-SHAKE-128f	RISC-V/XC7A100T	100	49.00	4.40	14k/—/—/—
[Saa24]	SLH-DSA-SHA2-256s	RISC-V/XC7A100T	100	69 620.10	8.90	
[ALCZ20]	SPHINCS+-128f-simple	XC7A100T	250	1.01	0.16	48k/73k/1/11.5
[ALCZ20]	SPHINCS+-256s-robust	XC7A100T	250	36.10	0.20	50k/76k/1/30
[SAW+23]	Falcon-512	ZCU104	188	4.20	—	23k/26k/101/23
[SAW+23]	Falcon-512	ZCU104	214	—	0.62	12k/8k/15/13
[SAW+23]	Falcon-1024	ZCU104	188	8.70	—	45k/41k/182/37
[SAW+23]	Falcon-1024	ZCU104	214	—	1.30	13k/9k/2/4

Table 10: Existing FPGA-based hardware implementations of various PQC signature schemes. For the listed related work, if the prior work implemented different variants of an algorithm, the fastest design is listed. The "—" indicates that the corresponding parameter was not specified or not implemented.