A High-Performance Hardware Implementation of the LESS Digital Signature Scheme

Luke Beckwith[1,2], Robert Wallace[1], Kamyar Mohajerani[1], and Kris Gaj[1]

[1] George Mason University, Fairfax, VA, 22030, USA
{lbeckwit, rwalla, mmohajer, kgaj}@gmu.edu
[2] PQSecure Technologies, Boca Raton, FL, 33431, USA
luke.beckwith@pqsecurity.com

Abstract. In 2022, NIST selected the first set of four post-quantum cryptography schemes for near-term standardization. Three of them - CRYSTALS-Kyber, CRYSTALS-Dilithium, and FALCON - belong to the lattice-based family and one - SPHINCS$^+$ - to the hash-based family. NIST has also announced an "on-ramp" for new digital signature candidates to add greater diversity to the suite of new standards. One promising set of schemes - a subfamily of code-based cryptography - is based on the linear code equivalence problem. This well-studied problem can be used to design flexible and efficient digital signature schemes. One of these schemes, LESS, was submitted to the NIST standardization process in June 2023. In this work, we present a high-performance hardware implementation of LESS targeting Xilinx FPGAs. The obtained results are compared with those for the state-of-the-art hardware implementations of CRYSTALS-Dilithium, SPHINCS$^+$, and FALCON.

Keywords: Code-Based Cryptography · Post-Quantum Cryptography · Hardware Acceleration · FPGA · Digital Signatures.

1 Introduction

The first set of post-quantum cryptography schemes was selected for standardization by NIST in 2022 [3]. These algorithms are intended to replace current public key standards, such as RSA and Elliptic Curve Cryptosystems, which are vulnerable to quantum attacks through the use of Shor's algorithm [28]. These new standards are built upon computationally hard problems that are secure against classical and quantum computing attacks. Three of the new standards are lattice-based algorithms: CRYSTALS-Kyber, CRYSTALS-Dilithium, and FALCON. The fourth, SPHINCS$^+$, is a hash-based algorithm. CRYSTALS-Dilithium, FALCON, and SPHINCS$^+$ are all digital signature schemes, while CRYSTALS-Kyber is a Key Encapsulation Mechanism (KEM). The primary recommendations from NIST and the NSA for most applications are CRYSTALS-Kyber and CRYSTALS-Dilithium due to their relatively small key sizes and high performance [3], [23]. FALCON is well suited for applications that require small signatures and fast verification but has complex and slow key generation and

signing. Compared to the other selected algorithms, SPHINCS$^+$ has lower performance and larger signatures. However, it has a mature security basis making it a more conservative option [3].

NIST intends to standardize additional algorithms to diversify the suite of new standards. This intent includes algorithms optimized for specific types of applications and algorithms of different cryptographic families. There are currently three code-based KEMs that have advanced to the fourth round for further evaluation, and NIST announced an "on-ramp" for new digital signature candidates with the submission deadline on June 1, 2023.

One of these new digital signature algorithms is LESS [8], a code-based digital signature scheme. Unlike many previous code-based algorithms, which are based on the Syndrome Decoding Problem (SDP), LESS builds its security on the difficulty of determining the linear isometry between two codes [12]. This security basis allows the use of smaller parameters than those typically required for algorithms based on SDP, enabling more practical key and signature sizes.

In this work, we present a high-performance hardware implementation of the LESS digital signature scheme. Our hardware architecture implements all base parameter sets and provides substantial improvements over the software implementation. The hardware implementation is also protected against timing attacks as all operations are constant-time with respect to the secret values. The implementation is publicly available at `github.com/GMUCERG/LESS`.

2 Previous Work

Since LESS is built using a different framework than previous code-based digital signature schemes, there are no existing hardware implementations we can make a direct comparison to. However, there are at least partial implementations of all the algorithms selected for standardization.

The NIST-selected digital signature algorithms have received varying levels of implementation work. CRYSTALS-Dilithium has received the most effort with several high-performance and lightweight hardware implementations [32], [10], [11], [21], [17]. A unified hardware design for CRYSTALS-Dilithium and Saber was presented in [1]. A similar work on a unified implementation of CRYSTALS-Kyber and CRYSTALS-Dilithium was presented in [2]. Additionally, software/hardware co-designs of CRYSTALS-Dilithium were reported in [34], [22], [33], [20]. Of particular relevance to this paper is the pure hardware implementation by Zhao et al. [32], which is the highest performance implementation reported thus far. SPHINCS$^+$ has one full implementation which targets high performance [4]. FALCON has received considerably less effort, with the only hardware implementation reported thus far being the implementation of the verification operation [11]. Additionally, a software/hardware co-design of the verification operation of FALCON was reported in [20].

Hardware implementations of Gaussian elimination over $GF(p)$, performing an operation similar to that of the Row Reduced Echelon Form unit described in this paper, were reported in [18], [19]. Additionally, hardware implementations

of Gaussian elimination over a different class of fields, $GF(2^m)$, were reported in [7], [6], [13], [27], [29], [30], [15], [31], [25], [26].

None of the previously reported designs can be easily adapted for the implementation of the LESS signature scheme. The major differences stem from the use of a) much larger matrix dimensions, which prevent the use of systolic array architectures, b) different field, which affects the complexity of addition, subtraction, multiplication, and inversion, and c) different expected output - the row reduced echelon form, rather than the (unreduced) row echelon form, also known as the upper triangular matrix.

3 Background

3.1 Generator, permutation, and monomial matrices

As this work discusses the code-based cryptosystem LESS, there are several important concepts from coding theory that must be defined.

A fundamental object in coding theory is the generator matrix. A generator matrix $G \in \mathbb{F}^{k\times n}$ defines an $[n, k]$-code by the operation $c = mG$, where $m \in \mathbb{F}^k$ is the message and $c \in \mathbb{F}^n$ is the corresponding codeword. The same code can also be defined using the parity check matrix $H \in \mathbb{F}^{(n-k)\times n}$. The parity check matrix can be used to check if a given vector c is a codeword by verifying that $Hc^T = 0$.

The generator matrix is said to be in standard form if the k leftmost columns are the identity matrix, that is if $G = (I_k|M)$ with $M \in \mathbb{F}^{(k-n)\times n}$. If G is in standard form, the corresponding parity matrix can be expressed as $H = [-M^T|I_{n-k}]$. If the generator is in standard form, the first k entries of any code word will simply be the message itself. Thus, the code is said to be systematic in its first k positions.

For applications that use the functionality of the code, it is beneficial to represent the generator matrix in its standard form because it enables easy derivation of the parity check matrix. For LESS, however, we are only concerned with determining if two matrices produce equivalent codes. That is, is there an invertible matrix $S \in GL_k(q)$ for $G, G' \in \mathbb{F}^{k\times n}$ such that $G = SG$. Thus, converting to any unique representation of a code is sufficient. The standard form representation is unique and thus would work for this purpose, but it is not required. The Reduced Row Echelon Form (RREF) is also a unique representation.

RREF is an extension of Row Echelon Form (REF), which is defined as follows: a matrix is in row echelon form if the following properties hold: (1) All rows consisting of only zeroes are at the bottom of the matrix, (2) the leading entry of every non-zero row is to the right of the leading entry of every row above it. That is, they form a staircase pattern. A matrix is then in RREF if it meets all the requirements of REF, all of the leading entries of non-zero rows are 1, and each column containing a leading 1 has zeros in all of its other entries. Pseudocode for converting a matrix to RREF is provided in Algorithm 1.

Other important concepts for LESS are permutation and monomial matrices. A permutation matrix is a square binary matrix that has exactly one entry of 1

in each row and each column and 0s elsewhere. A monomial matrix is a matrix with the same non-zero pattern as a permutation matrix. However, unlike a permutation matrix, where the non-zero entry must be 1, in a monomial matrix, the non-zero entry can be any non-zero value in F_q^*.

Algorithm 1: Converting a $k \times n$ matrix to Reduced Row Echelon Form (RREF)

Input: Matrix $G \in \mathbb{Z}_q^{k \times n}$
Output: Matrix $G \in \mathbb{Z}_q^{k \times n}$

for row_id_to_reduce $\in [0, k-1]$ **do**
  valid_pivot $\leftarrow 0$
  for col_id $\in$ [row_id_to_reduce, $n-1$] **do**
    for row_id $\in$[row_id_to_reduce, $k-1$] **do**
      if (G[row_id][col_id] > 0) *and* (valid_pivot $==$ *0*) **then**
        pivot_row_id $\leftarrow$ row_id
        pivot_col_id $\leftarrow$ col_id
        valid_pivot $\leftarrow 1$
  swap_row(G[row_id_to_reduce], G[pivot_row_id])
  $m \leftarrow G$[row_id_to_reduce][pivot_col_id]$^{-1}$ mod q
  for col_id $\in [0, n-1]$ **do**
    G[row_id_to_reduce][col_id] $\leftarrow m \cdot G$[row_id_to_reduce][col_id] mod q
  for row_id $\in [0, k-1]$ **do**
    if row_id $\neq$ row_id_to_reduce **then**
      $m \leftarrow G$[row_id][pivot_col_id]
      for col_id $\in$ [pivot_col_id, $n-1$] **do**
        tmp $\leftarrow m \cdot G$[row_id_to_reduce][col_id] mod q
        G[row_id][col_id] $\leftarrow G$[row_id][col_id]$-$ tmp mod q

3.2 LESS

LESS is a code-based digital signature scheme based on the difficulty of the Linear Equivalence Problem (LEP). LESS was first introduced by Biasse et al. [12] and was later expanded upon by Barenghi et al. [9] and Persichetti [24]. This work focuses on the most recent version of the scheme that was submitted to the NIST standardization process [8]. The linear equivalence problem can be defined as follows: given two generator matrices $G, G' \in \mathbb{F}_q^{k \times n}$ which generate codes $\mathbb{C}, \mathbb{C}'$, determine if the two corresponding codes are linearly equivalent. That is, does there exist matrices $S \in GL_k(q)$ and $P \in M_n$ such that $G' = SGP$.

The digital signature scheme is created by first defining a sigma protocol using the linear equivalence problem and then converting it to a non-interactive signature using the Fiat-Shamir transformation [16]. In the sigma protocol, there are two users involved: the prover, who is attempting to prove they know the secret corresponding to a public key, and a verifier, who is trying to confirm

Table 1: Parameter sets for LESS and resulting data sizes.

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

the identity of the prover. The private key is a monomial matrix $Q \in M_n$, and the public key is $G_1 = RREF(G_0 Q)$, where G_0 is a publicly available generator matrix. The prover first generates a commitment by sampling a random monomial $\tilde{Q}$ and calculating $\tilde{G} = RREF(G_0\tilde{Q})$. They then hash $\tilde{G}$ and send the hash to the verifier as the commitment. The verifier then responds with a single-bit challenge. If the challenge is 0, then the prover responds with $\tilde{Q}$, and the verifier checks the response by checking that the hash of G_0 multiplied by the response equals the commitment. If the challenge is 1, the prover responds with $Q^{-1}\tilde{Q}$, and the verifier checks the response by verifying that the product of G_1 and the response matches the commitment. Note that this holds true because $G_1 Q^{-1}\tilde{Q} = G_0 Q Q^{-1}\tilde{Q} = G_0\tilde{Q}$, which matches the commitment.

With each round of the protocol, an imposter has a $\frac{1}{2}$ chance of deceiving the verifier by guessing what the challenge will be. The difficulty of deceiving the verifier can be increased by repeating the protocol multiple times or by creating additional pairs of public and private matrices. LESS takes advantage of both approaches. This protocol can be converted into a digital signature scheme by having the prover pre-compute numerous commitments and then using an agreed-upon function to self-generate an unpredictable challenge. In the case of LESS, this is accomplished by hashing the commitment matrices with the message appended and using a variant of the Fisher-Yates shuffle to generate a challenge with a fixed number of non-zero entries.

The parameters for the version of LESS this work implements are described in Table 1. Parameters are provided for three security levels corresponding to the NIST-defined security levels 1, 3, and 5. There are multiple parameter sets for each security level. They aim for different optimization metrics. The balanced parameter sets, denoted by "b", seek to minimize the combined size of the public key and signature. The small parameter sets, denoted by "s", aim to minimize the signature size. Level 1 also has an intermediate parameter set, denoted by "i". The first three parameters relate to the codes used in LESS: n and k define the dimensions of the generator matrices, and q is the modulus of the coefficients. The following three relate to the LESS protocol: t defines the number of challenges, i.e., how many rounds of the sigma protocol are simulated.

The number of non-zero challenges is defined by ω, and the number of pairs in the key is defined by s.

The short parameter sets reduce the signature size by increasing the number of pairs in the key. This means fewer iterations of the protocol are required to reach the security threshold, and consequently, the number of responses in the signature is smaller. However, this comes at the cost of larger keys.

The descriptions of key generation, signing, and verification for LESS are provided in Algorithms 2, 3, and 4. In a key generation, the user generates s key pairs. The first pair is simply the public parameter G_0 and the identity matrix. All following pairs are generated by sampling a random Q_i and calculating $G_i = RREF(G_0 Q_i)$. Note that Q_i is assumed to be inverted when sampled, so for the calculation of the public key, we must invert the monomial before multiplication. This assumption removes the need for inverting the monomial in signature generation. The seeds used to sample the secret monomials are all derived from a single input seed $seed_{sk}$. The calculated matrices are serialized to minimize their size in the public key.

Algorithm 2: LESS-KeyGen() [8]

Input: None

Output: $\mathsf{sk} = (\mathsf{seed}_1, \ldots, \mathsf{seed}_{s-1})$: private key, where $\mathsf{seed}_i \in \{0,1\}^\lambda$ is employed to derive $\mathbf{Q}_i^{-1}$. The first entry of the private key $\mathbf{Q}_0 = \mathbf{I}_n$ is not stored.
$\mathsf{pk} = (\mathsf{seed}_0, \mathbf{G}_1, \ldots, \mathbf{G}_{s-1})$: public key, where $\mathbf{G}_i \in \mathbb{F}_q^{k \times n}$ is stored as the non-pivot columns and their positions via the CompressRREF subroutine.

Data: CSPRNG($seed, \mathbb{S}_{\mathsf{RREF}}$): Samples a generator matrix in RREF from the output of SHAKE using the provided seed.
CSPRNG($seed, \mathsf{M}_n$): Samples a monomial matrix from the output of SHAKE using the provided seed.
RREF($\mathbf{G}$): Converts input generator into RREF.
CompressRREF($\mathbf{G}_i$): Encodes pivot locations and non-pivot columns of generator matrix in RREF.

$\mathbf{G}_0 \leftarrow \text{CSPRNG}(\mathsf{pk}[0], \mathbb{S}_{\mathsf{RREF}})$
for $i \leftarrow 1$ **to** $s-1$ **do**
  $\mathsf{sk}[i] \xleftarrow{\$} \{0,1\}^\lambda$
  $\mathbf{Q} \leftarrow \text{CSPRNG}(\mathsf{sk}[i], \mathsf{M}_n)$
  $\mathbf{Q}_i \leftarrow \mathbf{Q}^{-1}$
  $\mathbf{G}_i \leftarrow \text{RREF}(\mathbf{G}_0 \mathbf{Q}_i)$
  $\mathsf{pk}[i] \leftarrow \text{CompressRREF}(\mathbf{G}_i)$
return $(\mathsf{sk}, \mathsf{pk})$

In a slightly simplified approach to signing, first t commitments are generated by sampling random monomials $\tilde{Q}_i$ and calculating $\tilde{G}_i = RREF(G_0 \tilde{Q}_i)$. All the

commit matrices are then encoded and hashed with the message to generate the challenge seed d. The challenge seed is then used to seed the XOF function from which the challenge is parsed. For the zero challenges, the seed used to sample the corresponding monomial serves as the response. For the non-zero challenges, $Q_{x_i}^{-1}Q_i$ is the response. The signatures are composed of the challenge seed and all responses.

An optimization can be performed that significantly reduces the size of the signatures. Instead of transmitting the entire monomial for the non-zero challenges, we can instead transmit only the columns corresponding to the pivot columns of the result. This reduces the transmission overhead of these monomials by a factor of two. However, to recover from the missing information of the monomial, additional processing is required after the RREF operation. The non-pivot columns are lexicographically minimized and sorted to remove the impact of the scaling and permutation operations of the monomial multiplication. These operations are combined into a single function called `PrepareDigestInput`. For further details, we refer to the LESS specification [8].

Another optimization is used for the seeds of the challenge monomials. Rather than defining the seeds using simple expansion of a root seed by an XOF, the seeds are defined as the leaves of a binary tree derived from the root seed. So, the seeds are generated by recursively hashing an input seed until the required number of leaves is reached. Then the signature size can be reduced by sending the tree nodes needed to recreate the target leaves rather than sending the leaves themselves.

In verification, the challenge seed is first expanded into the challenge in the same manner as in signing, and the leaves of the seed tree are regenerated from the path. For all responses t, the monomial is decoded or resampled and multiplied by the corresponding generator matrix. When the monomial is sampled from a seed, we use the same `PrepareDigestInput` algorithm to regenerate the commitment. When the monomial is decoded from the response, we use the standard RREF operation and minimize and sort the non-pivot columns of the result. All the generator matrices are then hashed, and the result is compared with the challenge seed in the signature. If they match, the signature is accepted.

4 Hardware Architecture

In this section, we discuss the design of our implementation of LESS. We begin with a brief description of the top-level architecture before discussing the details of the submodules and operation schedule. The datapath of packed matricies and seeds is $W = 64$. For all parameter sets, $n > q$. So for portions of the design that transmit data of both width $\lceil \log_2(n) \rceil$ and $\lceil \log_2(q) \rceil$, we use a width of $\lceil \log_2(n) \rceil$.

4.1 Top Level Architecture

The top-level datapath of the hardware architecture is shown in Fig. 1. The hardware modules implementing all the operations of LESS are partitioned into

Algorithm 3: LESS-SIGN($\mathsf{sk}, \mathsf{msg}, \mathsf{pk}$) [8]

Input: $\mathsf{sk} = (\mathsf{seed}_1, \ldots, \mathsf{seed}_{s-1})$: private key, where $\mathsf{seed}_i \in \{0,1\}^\lambda$ is employed to derive $\mathbf{Q}_i^{-1}$. The first entry of the private key which is the identity matrix $\mathbf{Q}_0 = \mathbf{I}$ is not stored.
$\mathsf{pk}[0] = \mathsf{seed}_0$: first element of the public key employed to derive $\mathbf{G}_0$ in RREF at runtime
msg: message to be signed, as a sequence of bits

Output: $\sigma = (\mathsf{rsp}_1, \ldots, \mathsf{rsp}_t, \mathbf{d})$: signature composed by a salt salt, ω ZKID protocol responses $\mathsf{rsp}_i, 0 \leq i < t$, the seed-tree path $\mathsf{treepath}$ and a digest $\mathbf{d}$

Data: CSPRNG($seed, \mathbb{S}_{t,\omega}$): Samples the fixed weight challenge from the output of SHAKE.
PREPAREDIGESTINPUT($\mathbf{G}, \widetilde{\mathbf{Q}}$): Calculates the RREF of $\mathbf{G}\widetilde{\mathbf{Q}}$ and returns the lexicographically sorted and minimized values of the non-pivot columns of the result as well as the corresponding entries of the monomial.
SEEDTREELEAVES($\mathsf{seed}, \mathsf{salt}$): Generates seed tree using SHAKE.
SEEDTREEPATHS($\mathsf{seed}, (x_0, \ldots, x_{t-1})$): Calculates nodes of path for the target leaves of the seed tree.
COMPRESSMONOMACTION($\mathbf{Q}\overline{\mathbf{Q}}$): Encodes the relevant permutation and coefficients of the monomial matrix.

$\mathbf{G}_0 \leftarrow \text{CSPRNG}(\mathsf{pk}[0], \mathbb{S}_{\mathsf{RREF}})$
$\mathsf{rootSeed} \xleftarrow{\$} \{0,1\}^\lambda$
$\mathsf{salt} \xleftarrow{\$} \{0,1\}^\lambda$
$(\mathsf{seed}[0], \ldots, \mathsf{seed}[t-1]) \leftarrow$ SEEDTREELEAVES($\mathsf{rootSeed}, \mathsf{salt}$)
for $i \leftarrow 0$ **to** $t-1$ **do**
  $\widetilde{\mathbf{Q}}_i \leftarrow \text{CSPRNG}(\mathsf{seed}[i], \mathsf{M}_n)$
  $(\mathbf{V}_i, \overline{\mathbf{Q}_i}) \leftarrow$ PREPAREDIGESTINPUT($\mathbf{G}_0, \widetilde{\mathbf{Q}}_i$)
$\mathbf{d} \leftarrow$ HASH($\mathbf{V}_0 || \ldots || \mathbf{V}_{t-1} || \mathsf{msg} || \mathsf{salt}$)
$(x_0, \ldots, x_{t-1}) \leftarrow \text{CSPRNG}(\mathbf{d}, \mathbb{S}_{t,\omega})$
$\mathsf{treepath} \leftarrow$ SEEDTREEPATHS($\mathsf{rootSeed}, (x_0, \ldots, x_{t-1})$)
$j \leftarrow 0$
for $i \leftarrow 0$ **to** $t-1$ **do**
  if $x_i \neq 0$ **then**
    $\mathbf{Q} \leftarrow \mathsf{sk}[x_i]$
    $\mathsf{rsp}_j \leftarrow$ COMPRESSMONOMACTION($\mathbf{Q}\overline{\mathbf{Q}_i}$)
    $j \leftarrow j + 1$
return $(\mathsf{salt}, \mathsf{treepath}, \mathsf{rsp}_0, \ldots, \mathsf{rsp}_{\omega-1}, \mathbf{d})$

five major submodules: the seed generator, monomial arithmetic unit, generator arithmetic unit, RREF unit, and challenge generator. The seed generator is responsible for expanding the input seeds into the seeds used for sampling of monomial and generator matrices. This includes the simple expansions of the

Algorithm 4: LESS-VERIFY($\mathsf{pk}, \sigma, \mathsf{msg}$) [8]

Input: $\mathsf{pk} = (\mathbf{G}_0, \ldots, \mathbf{G}_{s-1})$: public key, where $\mathbf{G}_i \in \mathbb{F}_q^{k \times n}$
$\sigma = (\mathsf{salt}, \mathsf{treepath}, \mathsf{rsp}_0, \ldots, \mathsf{rsp}_{\omega-1}, \mathbf{d})$: signature composed by a salt salt, *omega* ZKID protocol responses $\mathsf{rsp}_i, 0 \leq i < t$, the seed-tree path $\mathsf{treepath}$ and a digest $\mathbf{d}$
msg: message to be signed, as a sequence of bits

Output: Boolean value indicating whether the signature is valid

Data: REBUILDSEEDTREELEAVES($\mathsf{treepath}, (x_0, \ldots, x_{t-1}), \mathsf{salt}$): Regenerates the target leaves of the seed tree using the path nodes.
EXPANDTOMONOMACTION(rsp): Decodes the encoded coefficients and permutation values from the monomial.
LEXMIN($\mathbf{v}$): lexicographically minimizes the input vector.
LEXSORTCOLUMNS($\mathbf{V}$): lexicographically sorts the set of the input vectors.

$\mathbf{G}_0 \leftarrow \text{CSPRNG}(\mathsf{pk}[0], \mathbb{S}_{\mathsf{RREF}})$
$(x_0, \ldots, x_{t-1}) \leftarrow \text{CSPRNG}(\mathbf{d}, \mathbb{S}_{t,\omega})$
$(\mathsf{seed}[0], \ldots, \mathsf{seed}[t-1]) \leftarrow$ REBUILDSEEDTREELEAVES($\mathsf{treepath}, (x_0, \ldots, x_{t-1}), \mathsf{salt}$)
for $i \leftarrow 0$ **to** $t-1$ **do**
  if $x_i = 0$ **then**
    $\widetilde{\mathbf{Q}}_i \leftarrow \text{CSPRNG}(\mathsf{seed}[i], \mathsf{M}_n)$
    $(\mathbf{V}_i, \overline{\mathbf{Q}_i}) \leftarrow$ PREPAREDIGESTINPUT($\mathbf{G}_0, \widetilde{\mathbf{Q}}_i$)
  else
    $\overline{\mathbf{Q}}_i \leftarrow$ EXPANDTOMONOMACTION(rsp_i)
    $\mathbf{G}_i \leftarrow \mathsf{pk}[x_i]$
    $\overline{\mathbf{G}}_i \leftarrow \mathbf{G}_i \overline{\mathbf{Q}}_i$
    $[\mathbf{I}\ \mathbf{V}_i] \leftarrow \text{RREF}(\overline{\mathbf{G}}_i)$ // $\mathbf{V}_i = [\mathbf{v}_0\ \ \mathbf{v}_1\ \ \cdots\ \ \mathbf{v}_{n-k-1}]$
    for $j \leftarrow 0$ **to** $(n-k)-1$ **do**
      $\mathbf{v}_j \leftarrow$ LEXMIN($\mathbf{v}_j$)
    $\mathbf{V}_i \leftarrow$ LEXSORTCOLUMNS($\mathbf{V}_i$)
$\mathbf{d}' \leftarrow \text{HASH}(\mathbf{V}_0 || \ldots || \mathbf{V}_{t-1} || \mathsf{msg} || \mathsf{salt})$
if $(\mathbf{d} = \mathbf{d}')$ **then**
  return true
return false

secret key seed as well as all seed tree operations. The monomial arithmetic unit performs the sampling, inversion, multiplication, encoding, and decoding of monomial matrices. It receives input from the seed generator when sampling and transfers the monomial matrices to the generator module as needed. The generator module performs the generator-monomial multiplication, encoding, as well as lexicographic sorting. The RREF unit converts the matrix multiplication result into RREF. The challenge parser hashes the commitment matrices and uses the result to generate the challenge.

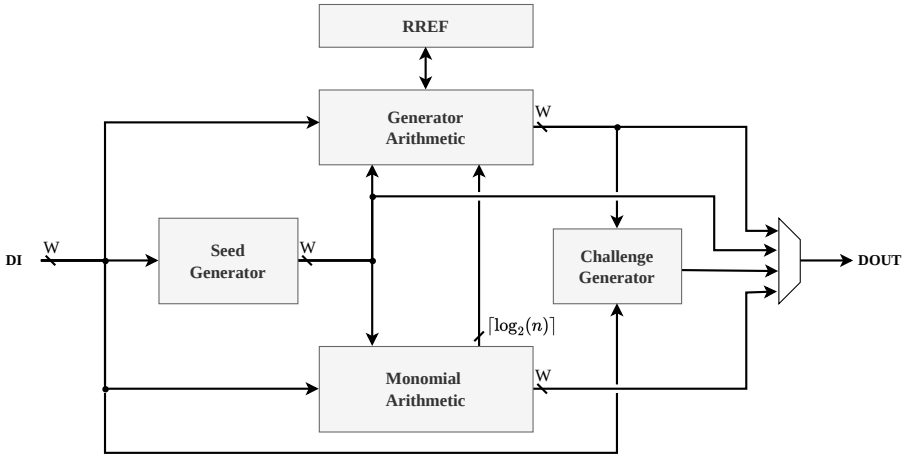

Fig. 1: Top-Level Block Diagram of LESS Hardware Architecture.

During most operations, only a single generator matrix needs to be stored throughout the entire operation. For example, during key generation and signing, only G_0 needs to be used multiple times. All other generator matrices are immediately hashed or unloaded from the accelerator. The exception is verification, where all public keys may be required to check the authenticity of the signature. For the balanced parameter sets, which only uses two generator matrices, this does not cause any issues. However, for the short parameter sets, there are eight matrices in the public key. Due to the large size of these matrices, this requires a significant amount of memory resources. To address this limitation, we assume that the system that is connected to the accelerator holds the full public key. The accelerator requests the generator matrices as they are needed during verification.

4.2 Submodule Design

Seed Generator. All operations of LESS require the generation of various seeds for the sampling of monomial and generator matrices. The architecture of this module can be seen in Fig. 2. The SHA-3 module used is a publicly available implementation [14]. During key generation, the generation of seeds is done by expanding the λ-bit seed into $s-1$ λ-bit seeds using the XOF. The SHA-3 module is configured to run the appropriate variant of SHAKE. The core ingests the input seed and produces the appropriate number of output bits which are stored back in memory. These seeds can then be used to initialize the XOF for sampling as needed.

During signature generation and verification, the seed tree operations are also utilized. This includes the generation of the seed tree by recursively hashing the root seed, generation of the path nodes needed for the signature, and recreation of

the relevant leaves using the path nodes. The seed tree operation is implemented in a straightforward manner using breadth-first traversal of the tree and hashing the current node into two child nodes until the required number of leaves is generated. The seed path generation is performed in two steps. During the first step, the tree is traversed from the leaves up, and a flag is set for each seed to indicate whether or not it is in the path of the target seeds. For the leaves themselves, they are considered target seeds if they are not part of the non-zero challenges. For the node seeds, they are included if both of their children are included. Once the flags of all seeds are set, the tree is traversed again, and seeds are included in the path if their flag is set but their parent node is not. The entire seed tree is kept in memory after the initial generation, so no hashing is required during this operation. During the regeneration of the leaves, the first step from path generation is repeated. For the second step, the tree is traversed in the same manner, but once a seed that is in the path is reached, it is hashed to generate its child nodes.

All of these operations can be performed using a very simple datapath shown in Fig. 2. The controller is responsible for tracking the current location during tree traversal and setting the $flgi$ signal, which is used to set the flag for each seed as needed during the seed tree operations.

All operations performed by the seed generator are constant time with respect to the sensitive data. Hashing requires a constant number of cycles. Thus seed expansion does not leak any information. The generation and usage of the seed tree path are non-constant time, but the variation of the latency depends on the public challenge, not the secret seeds.

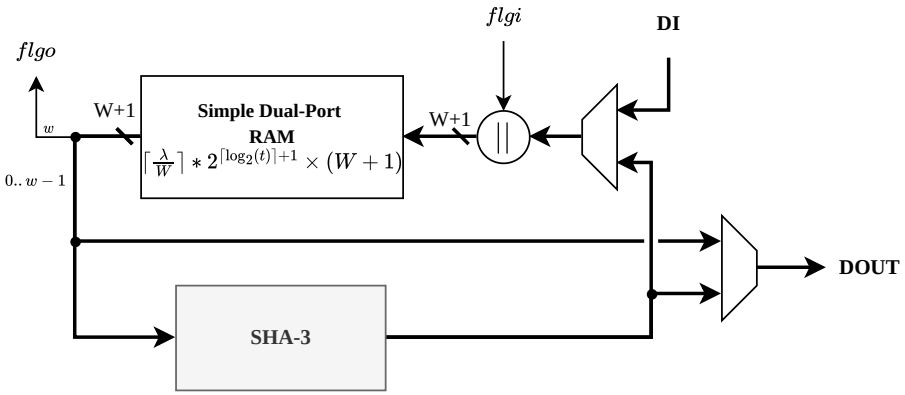

Fig. 2: Top-level Diagram of Seed Generator Submodule

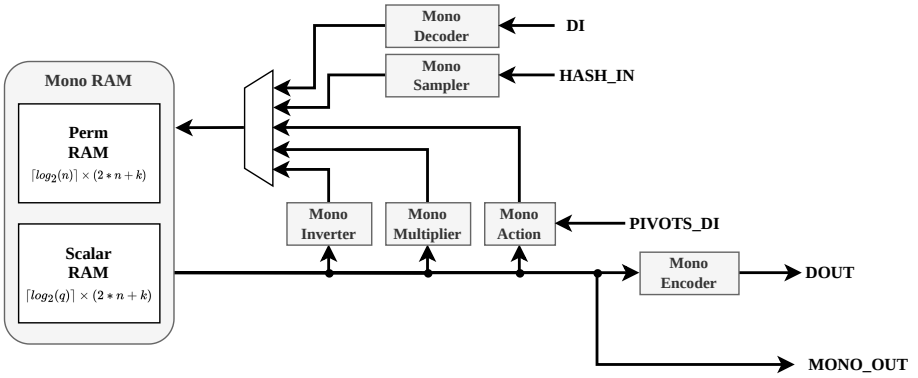

Fig. 3: Top-level Diagram of Monomial Submodule

Monomial Arithmetic. The top-level architecture of the monomial arithmetic unit is shown in Fig. 3. This section of the hardware consists of two memories and five submodules related to the monomial matrices.

Monomial sampling is required in key generation for the creation of the secret key and in signing for the generation of the ephemeral secrets used to create the commitments. The hardware architecture implementing monomial sampling is shown in Fig. 4. The monomial is represented as two lists, one representing the scalar values and one representing the permutation. The scalar values are generated using rejection sampling on $\lceil \log_q(q) \rceil$ bits of pseudorandom input at a time. Samples are accepted if they are in the range $[0, q-2]$ and then are incremented by one to shift them into the range $[1, q-1]$. Since the latency of monomial sampling is negligible in comparison to the latency of the RREF operation, only one sample is processed per cycle. The permutation coefficients are generated using a simple shuffling algorithm. The shuffler module contains a $N \times \lceil \log_2(n) \rceil$ RAM module, which is initialized to hold the array $[0, 1, ..., n-1]$. This array is then shuffled using n random samples in the range $[0, n-1]$.

Monomial encoding is a straightforward serialization of the permutation and scalar values. This is accomplished using a variable-rate bus width converter, which can receive input at a rate of $\lceil \log_2(n) \rceil$ or $\lceil \log_2(q) \rceil$ and produces an output of length W. Fig. 5 shows the architecture of this module. The decoding module follows a similar architecture with the modification that the input is W bits and the output can optionally be $N \times \lceil \log_2(n) \rceil$ or $N \times \lceil \log_2(q) \rceil$ bits.

Monomial inversion involves element-wise inversion of the scalar values of the monomial as well as calculation of the inverse permutation. The architecture implementing this operation is shown in Fig. 6. The inverse permutation b of permutation a can be calculated by $b_{a_i} = i$. This can be accomplished by using the input permutation as the address of a memory while writing values sequentially from $0, ..., n-1$. In the hardware module shown, the input permutation is used as the address input when writing the sequential values coming from a counter. After the entire permutation is loaded in, the RAM will contain the in-

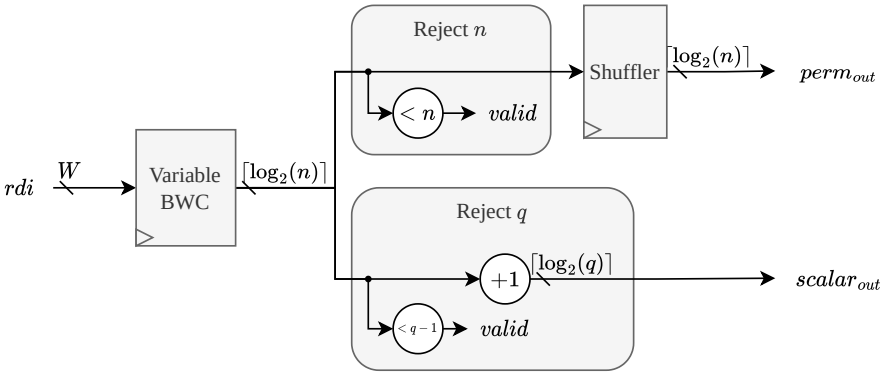

Fig. 4: Monomial Sampler Block Diagram

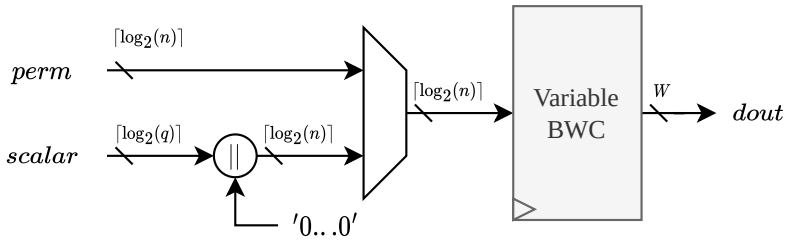

Fig. 5: Monomial Encoder Block Diagram

verse permutation, which can be read out sequentially using the counter to drive the address input. Since the modulus is small, inversion of the coefficient values can be done inexpensively using a Look-Up-Table one coefficient at a time. The ordering of the coefficients must also be adjusted to match the new permutation so the coefficients are written into RAM in the same manner as the permutation.

Monomial multiplication involves combinations of both the permutations and the scalar values of the two input monomials. The resulting permutation is calculated by reading the permutation of the left operand with the permutation of the right operand. In the hardware architecture shown in Fig. 7, this is accomplished by first writing the left operand's permutation into a RAM and then unloading using the right operand's permutation as the address. The scalar values are calculated by first applying the right operand's permutation to the scalar values of the left-operand and then multiplying coefficient-wise with the left operand's coefficients. This is accomplished in the hardware by writing the left operand's scalar values into memory and then using the right operand's permutation to drive the address when performing the multiplication.

The monomial operations are also constant time with respect to the sensitive data. Monomial inversion, multiplication, encoding, and decoding are all constant time operations, as their latency depends only on the dimension of the

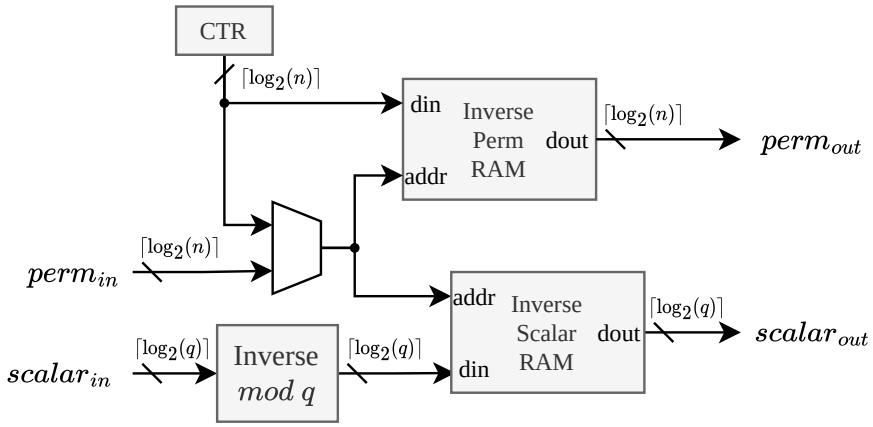

Fig. 6: Monomial Inverter Block Diagram

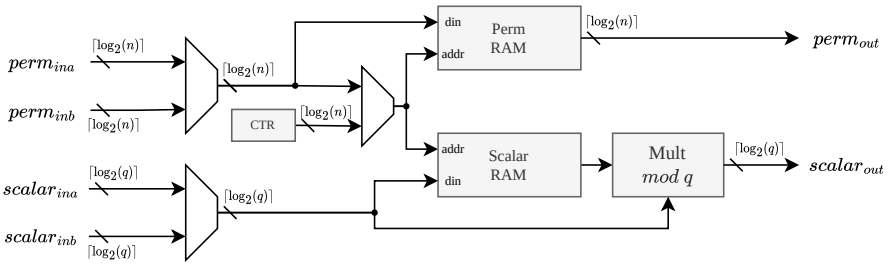

Fig. 7: Monomial Multiplier Block Diagram

matrices and not on the values within them. Monomial sampling is not strictly constant time as it uses rejection sampling. However, the difference in latency caused by rejection does not leak any information about the value of the accepted samples. Thus it is not vulnerable to timing attacks.

Generator Arithmetic. The generator arithmetic module has three primary functions: 1) preparing the generator for processing by decoding a matrix from the public key or sampling it from a seed, 2) performing the monomial multiplication while loading the generator into the RREF module, and 3) performing the post-processing of sorting and encoding the non-pivot columns after RREF.

Generator encoding and decoding is similar to monomial encoding in that it is a straightforward serialization of elements, one being a set of n bits representing whether each column is a pivot or not and the other being the $\lceil \log_2(q) \rceil$ bit coefficients. The list of pivot locations is serialized first at the beginning of the encoded string. Then the coefficients of the non-pivot columns are serialized row-wise.

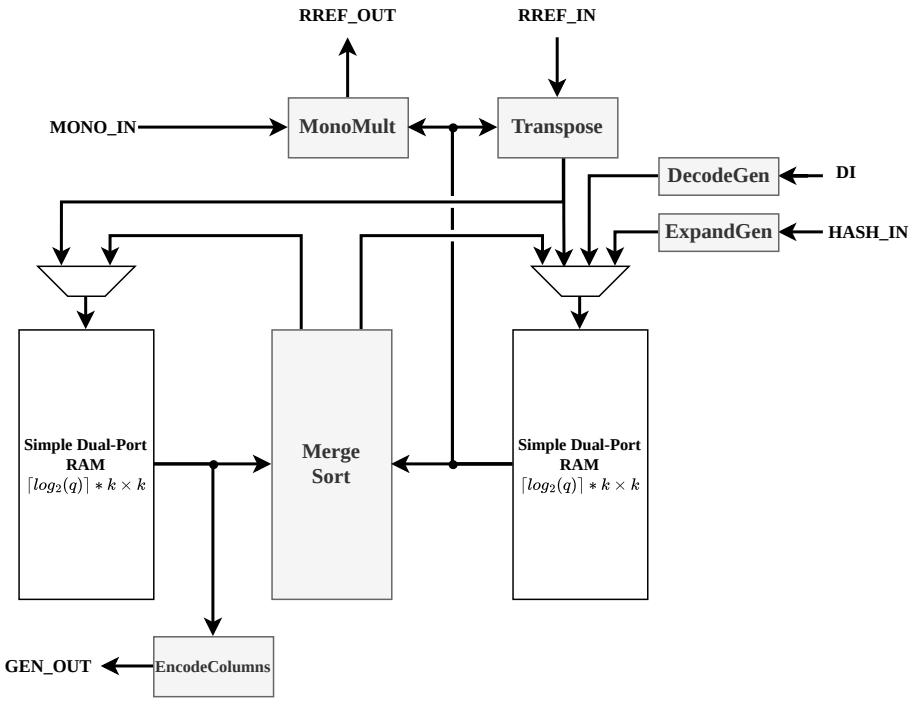

Fig. 8: Top-level Diagram of Generator Submodule

The monomial multiplication is performed when loading the matrix into the RREF module. An entire row can be accessed from memory at once. The permutation is applied using several $k \times 1$ multiplexers to read the row coefficients using the monomial permutation. The selected row coefficients are then scaled by the monomial coefficients before they are loaded into the RREF module.

During the lexicographic sorting operation, the matrix must be sorted column-wise. However, the operations of RREF and encoding are performed row-wise. Therefore we have a module which transposes the matrix when it is received from the RREF module. The columns are then sorted using an implementation of merge sort. During this operation, the entire column is accessible from the memories, so the comparison operation can be performed in a single cycle. Since merge sort requires additional memory overhead during processing, both memories are used during the sorting process. The sorted columns are then transposed back before encoding of the columns.

All operations performed by the generator arithmetic module are secure against timing attacks. The latencies of decoding, monomial multiplication, and transposition are determined by the dimension of the matrix and always take the exact same number of cycles. The expansion of the generator is a public operation. Merge sort was selected as the approach for sorting the columns because

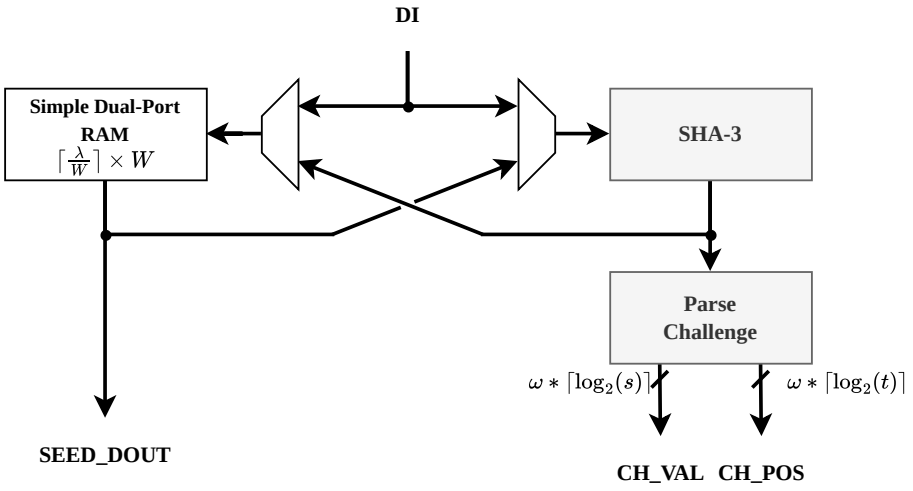

Fig. 9: Top-level Diagram of Challenge Generator Submodule

of its excellent performance in hardware and because it is very straightforward to implement in constant time. Each of the $\log_2(k)$ layer involves exactly k read, write, and comparison operations. Since these operations always require the same amount of time, the entire operation is constant time.

Challenge Generator. The challenge generator module is responsible for hashing the commitment matrices and parsing the signature element d into the fixed-weight challenge. The challenge is parsed by first sampling the values of the ω non-zero entries. When s is two, this stage can be skipped since 1 is the only possible value. These samples are written into the top ω entries of a t entry memory. They are then randomly permuted using a variant of the Fisher-Yates shuffle. A counter p is initialized to $t - \omega$, and then samples are repeatedly generated in the range $[0, p-1]$ using rejection sampling to determine where to shuffle the value at index p. This is repeated until all ω samples have been shuffled into the challenge. Since the parsing of the challenge is a public operation, it is not a target for timing attacks.

RREF. The RREF operation converts a matrix to row reduced echelon form. The typical complexity of this operation on a $k \times n$ matrix is $O(nk^2)$, where all k rows must be reduced, and each reduction requires all $k \times n$ elements to be operated on.

The reduction of a matrix to RREF is described in Algorithm 1. Four major steps of the algorithm can be identified. They are: (1) pivot search, (2) row swap, (3) rescaling a pivot row, and (4) reduce other rows. These steps are repeated

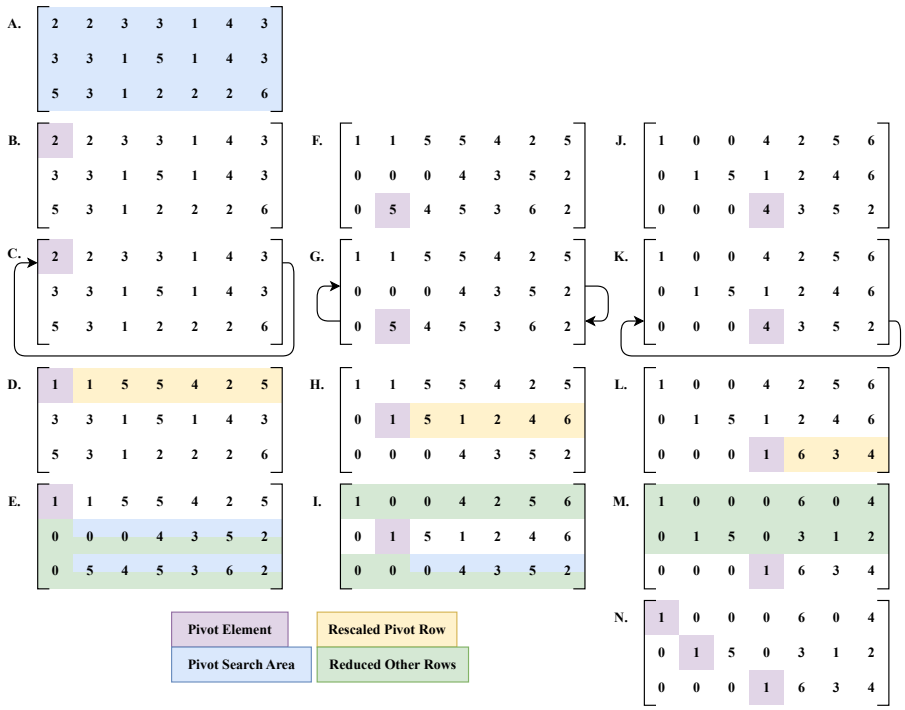

Fig. 10: RREF Example: $n = 7, k = 3, q = 7$

k times, once for every row, so that all rows of the matrix are fully reduced. The first step is identifying a pivot element. The pivot of a row is the leftmost non-zero element such that after the reduction of the matrix, the pivot will be 1, and all elements below it, in the same column, will be 0. The pivot search step is described in lines 2-8 of Algorithm 1. After finding the pivot (which is not guaranteed to be in the row to reduce), a row swap is performed so the row to reduce always contains the pivot. Next, the pivot row is rescaled so that the pivot element is 1. This is achieved by multiplying the entire row by a multiplier equal to the inverse of the pivot element modulo q. This operation is described in lines 11-12 of Algorithm 1. Finally, all other rows are reduced so that the elements in the same column as the pivot, above and below, are set to 0. This operation is described in lines 13-18 of the Algorithm 1.

There are several features of this algorithm that can be taken advantage of for optimization. When performing an arithmetic operation on a row, such as rescaling a pivot row or reducing a non-pivot row, there is no sequential dependence between elements. Arithmetic operations can be performed on all elements of a row at the same time. This parallelism reduces the time complexity of the algorithm to $O(k^2)$ and creates an $O(n)$ area cost in hardware. Additionally, the rows involved in the pivot search are always bounded by the row to reduce and k. This means that each time a pivot search is performed, the pivot row will

always be between the row to reduce and k, and the search will require less time, each iteration, at a constant rate. Also, the current iteration's reduce other row's results are the elements that are searched during the pivot search in the next row to reduce. Cycles can be saved by searching for the next pivot while performing reduce other rows operation of the current row to reduce. Finally, once the rescale pivot row step has been completed, all operations being performed in the reduce other rows step are row-independent. This means that any row can be reduced in any order, creating an independent series of operations that can be pipelined to increase the hardware frequency, while maintaining a high throughput. Combining these observations about the pivot search and reduce other rows, the reduce other rows operation can be performed on rows in the pivot search area so that the next pivot will always be found before the next iteration of row reducing begins, masking the time spent searching for a pivot.

The small scale example provided in Fig. 10 identifies the the key features of the RREF operation. Step A. starts with identifying the first pivot, where the search area is the entire matrix. After finding the pivot, the pivot row is swapped so that it is in the same position as the row to reduce, this swap takes place in step C. The pivot row is rescaled in step D, and in step E, all other rows are reduced. The pivot search area in step E includes one less column and row than the area in step A. The search also overlaps with the reduction of other

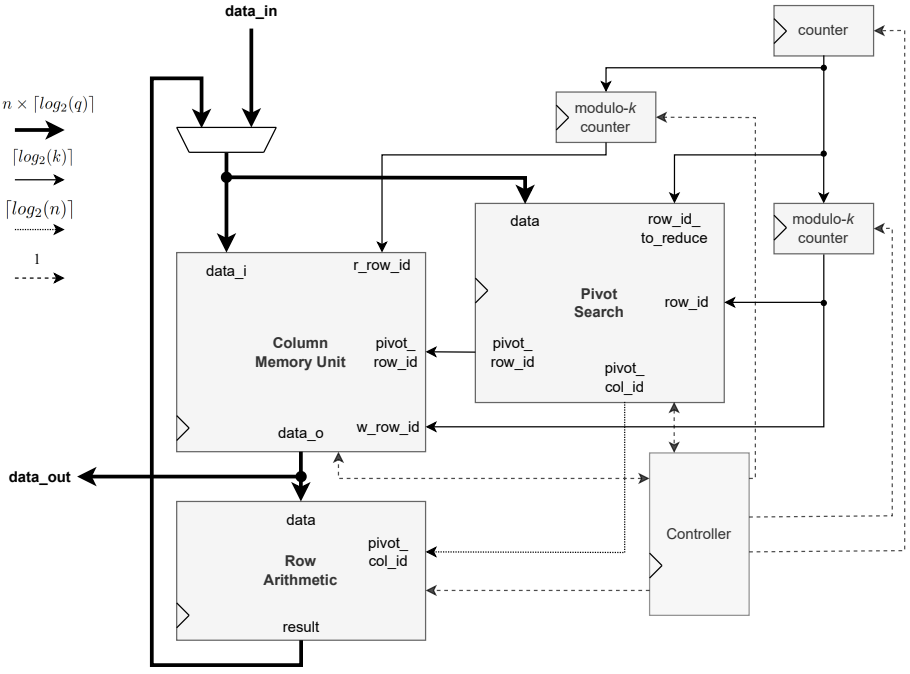

Fig. 11: RREF Top-Level Block Diagram

rows. Once all of the other rows are reduced, the search is also completed and the pivot is identified by step F. From here, the process is repeated until all k pivots have be identified and their rows reduced.

The hardware implementation of RREF aims to take advantage of each of the identified characteristics. The top-level module is split into three main parts: column memory unit, pivot search, and row arithmetic. The top level block diagram is presented in Fig. 11. Each part operates on an entire row of the input matrix at once. The pivot search unit is designed to search only rows that are within the search area for a specific row to reduce. This feature is in line with the row arithmetic unit's write back, so the search for the pivot of the next row to reduce occurs during the operations on the current row to reduce. The column memory unit provides separate read and write ports to enable pipelining of the rescale arithmetic which supports its highly parallel nature.

The column memory unit provides a wide interface to enable reading/writing to an entire row in a memory single access. The block diagram for the column memory unit is presented in Fig. 12. It is built up of n simple dual-port RAMs with synchronous read operating in parallel. Additionally, when addressing the memory, an address translation table is used. The table is built of a true dual-port RAM to enable the swapping of rows without needing to access the entire memory. A separate translation table is required for both the read and write ports of the column memory unit so the pipelined accesses do not need to target the same row. Any access to this memory unit will require two cycles, one for the address translation and another for the data access.

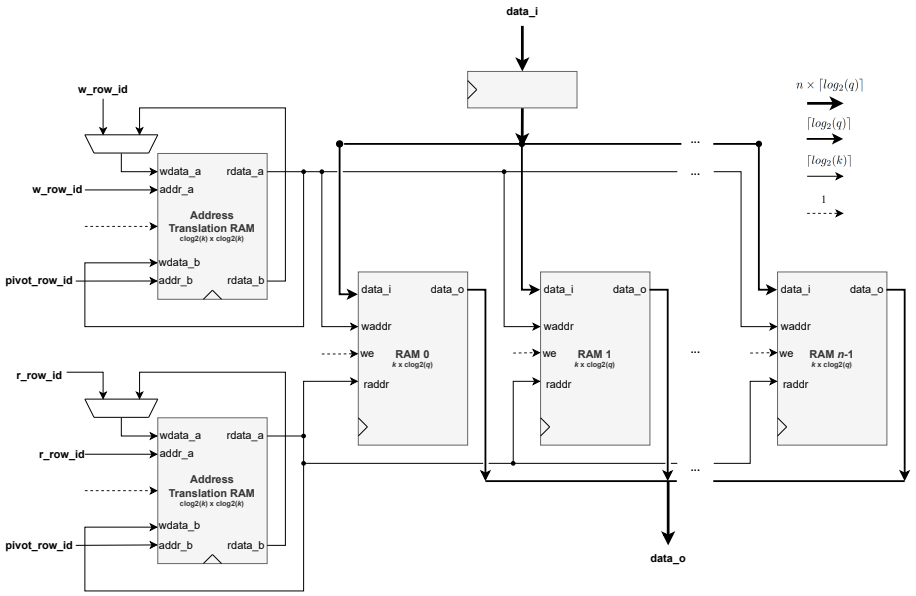

Fig. 12: RREF Column Memory Unit Block Diagram

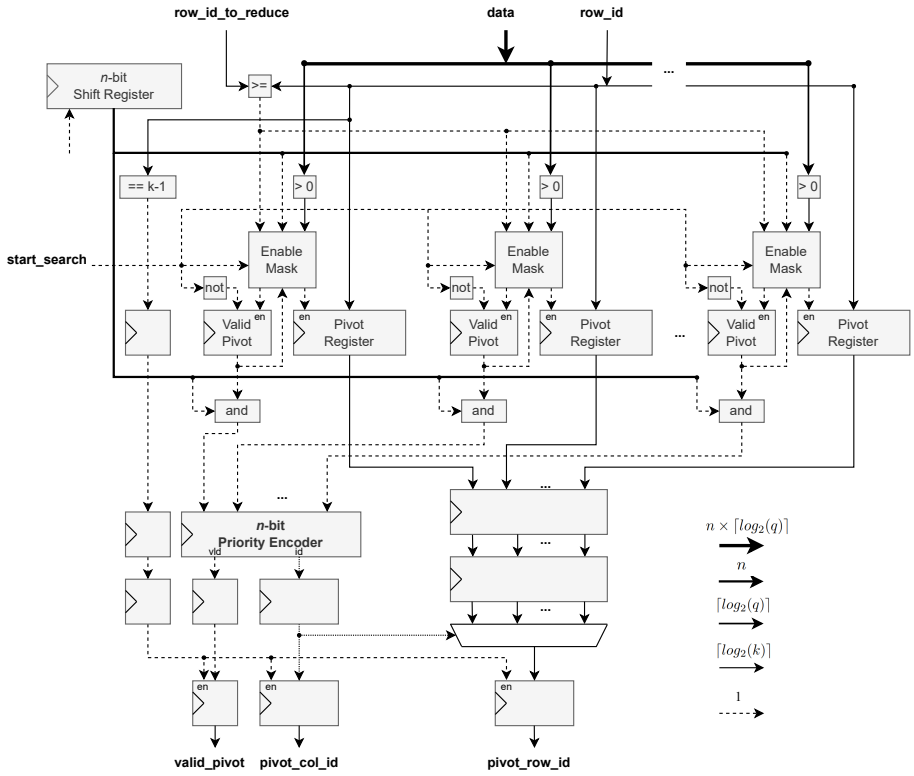

Fig. 13: RREF Pivot Search Block Diagram

The pivot search circuit, presented in Fig. 13, guarantees the results of the pivot search in a deterministic number of cycles, independent of the location of the pivot. This is achieved by requiring the entire matrix to be searched before revealing the result, guaranteeing a constant number of clock cycles spent searching, independent of any input data. The pivot of a specific row to reduce is the first non-zero element in a row, greater than or equal to the row to reduce, in the leftmost column, greater than or equal to the row to reduce. To identify the pivot, as the row id changes between the row to reduce and $k-1$, the corresponding data is checked to be non-zero using n comparators operating in parallel. If a comparator determines that its corresponding column element is non-zero, then it will set a flag and record the row id in its own register. At this point, that specific columns pivot has been determined for a row to reduce, meaning that there are now n registers holding the row id of the first non-zero element in a column and a flag to identify if the column contains a non-zero element. An n-bit priority encoder is used on all n flag bits to determine the leftmost column that contains a pivot. These flags are masked so that only a column id greater than or equal to the row to reduce is identified. The encoded

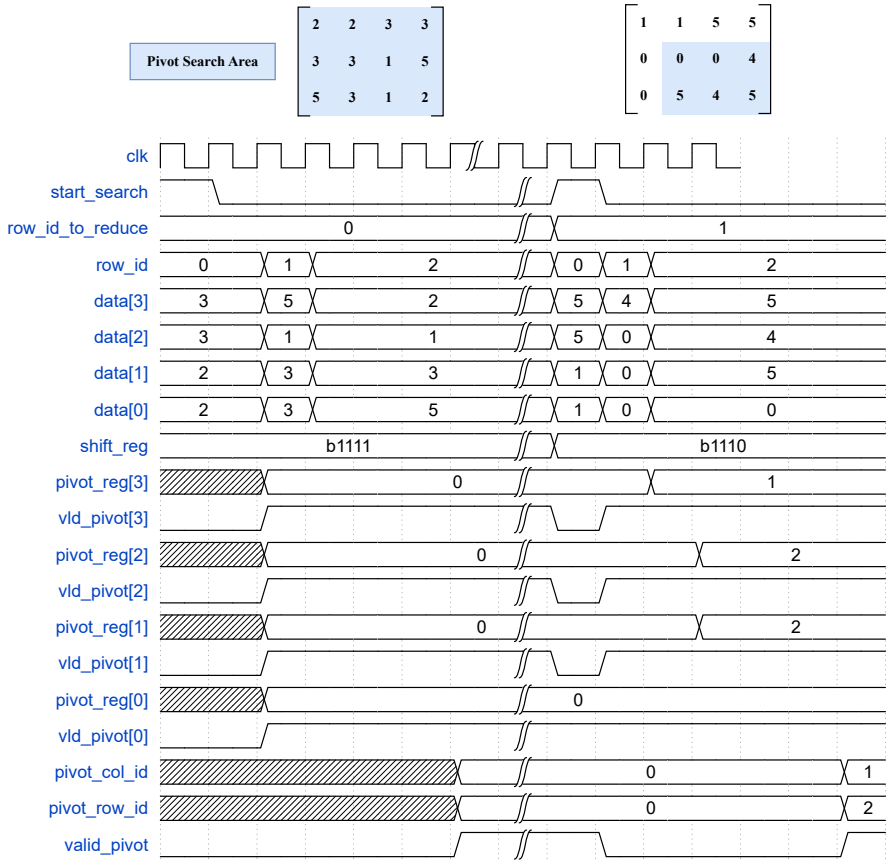

Fig. 14: RREF Pivot Search Example

column id is then used to read from the corresponding register to identify the row id of the pivot element in that column. By the time all k rows have been iterated over, the circuit will have determined the pivot row id and pivot column id for the corresponding row id to reduce. There is a three-cycle latency for the results of the pivot search to be accessible by the rest of the module to support a higher frequency. This latency will be masked away by the top-level pipeline.

An example of the pivot search circuit in operation, with $n = 4$, is provided in Fig. 14. After the start_search signal is set, the valid pivot flip-flops (vld_pivot[i]) are cleared. This action initializes the circuit to begin a search. If the data in a column within the search area is non-zero, then the corresponding pivot register will store the index of the first (lowest-index) row containing a non-zero element. Each valid pivot flip-flop will store a value indicating whether there exists at least one non-zero element in the search area of a given column. The priority encoder uses the flags of each column and a mask, driven by the shift register, to identify the lowest column id with a valid pivot. The mask, stored in shift_reg, makes it

possible to shrink the search area when iterating over subsequent values of the row_id_to_reduce. Once the pivot_column_id is determined, the row id from the corresponding pivot register is routed to the output register, pivot_row_id, of the pivot search circuit. The valid signal of the priority encoder is used to determine if a valid pivot was found in the current iteration of the algorithm. The search area of the next iteration of row_id_to_reduce is smaller than the previous one. It does not include row 0 and column 0. The start_search signal is asserted again to clear the valid pivot flip-flops corresponding to the columns located inside of the search area. In the iteration when row_id is 1, columns 1 and 2 contain zeros, this causes the corresponding pivot registers and valid pivot flip-flops to not be updated. When a non-zero value occurs in a column and row within the search area, and the valid pivot flag is not already set, then the row index is captured, and the valid pivot flag is raised. If the pivot search circuit cannot find a pivot, the RREF controller will halt operation and signal that an error has occurred. In the context of LESS key generation, signing, and verifying, all input matrices to the RREF operation are guaranteed to have an RREF.

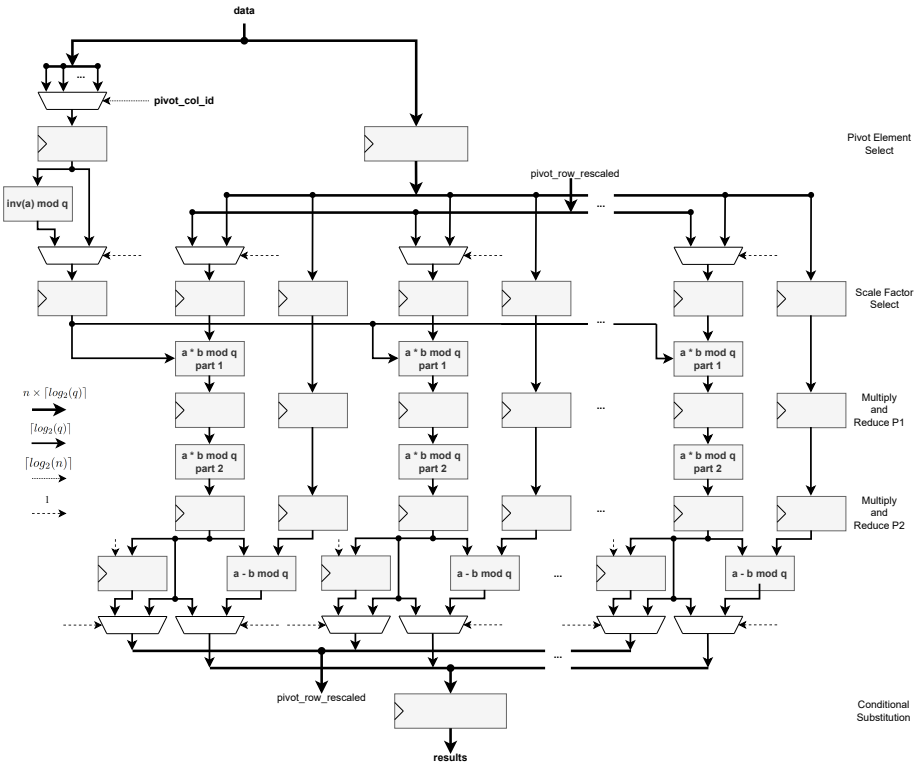

Fig. 15: RREF Row Arithmetic Block Diagram

The row arithmetic circuit, shown in Fig. 15, is used to perform multiplication, subtraction, and reduction modulo q of all elements in a row. It can be controlled to switch between rescaling a pivot row and reducing other rows. The circuit contains n arithmetic units to operate on an entire row in parallel. To support higher clock frequencies, the circuit takes advantage of the parallel nature of the rescaling operation by implementing several pipeline registers. The arithmetic pipeline has 5 stages: pivot element select select, scale factor select, two stages of multiply and reduce, and conditional subtract. The pivot row must be rescaled before the reduction of other rows can begin. The bypass of the last row of registers and the feedback loop allow starting the reduction of other rows a couple of cycles earlier than the when the rescaled pivot row is written back to memory. The pipeline bypass consistently occurs independent of input data, so this circuit will always complete its operation in a constant number of clock cycles.

The RREF order of operations is demonstrated in Fig. 16. While data is being loaded into the RREF internal memory, the first pivot search occurs. Once all rows of the matrix are loaded in, then the RREF pipeline begins by performing a swap, if no swap is required, the clock cycle is spent swapping in place. The rescale pivot row and reduce other rows operations follow, while performing the next pivot search during the reduce other rows. These operations repeat until the matrix is fully reduced.

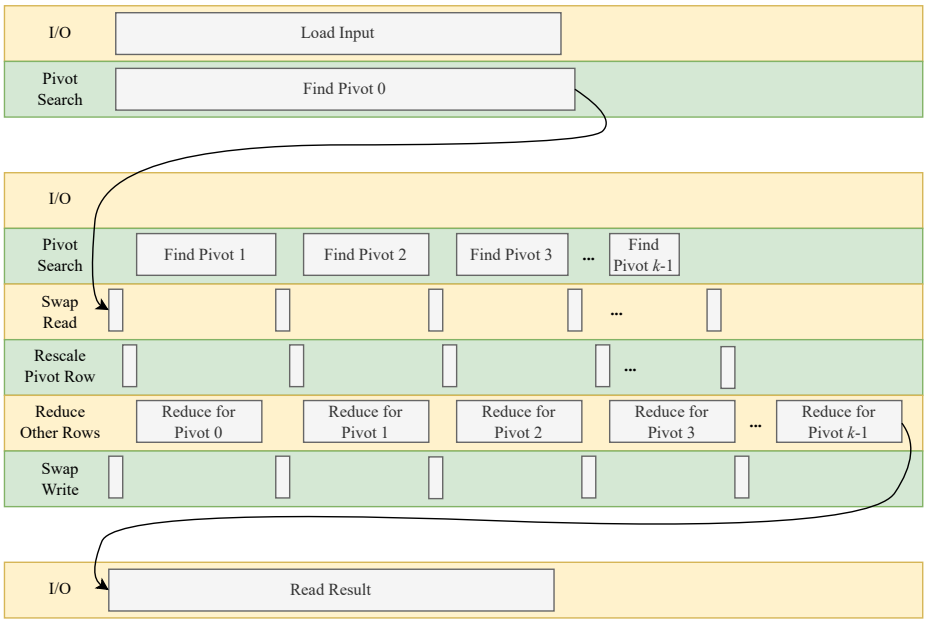

Fig. 16: RREF Operation Scheduling

All loops within the RREF algorithm described in Algorithm 1 are bounded by constants and do not exit early due to any results. This enables a fixed latency to perform all operations of RREF. Additionally, memory access also has a fixed latency, regardless of the data. Therefore the RREF operation is completed in constant time, regardless of the input matrix. The required cycles to perform the operation, for any parameter set, not including loading the input matrix and unloading the output matrix, can be represented by $k^2 + 3k + 58$.

4.3 Operation Scheduling

The schedule of operations used to perform the algorithms of LESS is described in Figs. 17, 18, and 19. The figures provide insight into the order of operations and which of them can be performed in parallel, but the duration of the operations is not to scale. In the key generation, the operation begins with the module receiving the secret key seed and the parameter generator matrix G_0. The seed is expanded into $s - 1$ seeds which are used to sample the secret key monomial matrices. The inversion of the first monomial matrix is performed in parallel with the sampling of the generator matrix. Then the generator matrix is multiplied by the monomial matrix before being written loading into the RREF module. The RREF operation is then started, and the next monomial is sampled and inverted in parallel with the operation. Once RREF completes, the resulting generator matrix is encoded and unloaded from the hardware. This loop of RREF is repeated in parallel with monomial sampling until all generator matrices for the public key are calculated.

The first stage of signing is similar to key generation, except that the resulting generator matrices are hashed instead of unloaded. Before the hashing, these non-pivot columns are transposed, sorted, and then encoded. The sorting, encoding, and hashing are performed in parallel with the RREF operation. Once t generator matrices are calculated and hashed, the message is ingested to the hash function to generate the challenge seed. The challenge seed is then used to parse the challenge itself. There are t monomial matrices sampled during the commitments. However, only ω are needed for the response. Since ω is much smaller than t and monomial sampling is a computationally inexpensive operation, it is more efficient to resample these needed matrices. The resampling is performed, and then the results are multiplied together and encoded as a part of the signature. After all the non-zero responses are calculated, the seed generator generates the path needed for the regeneration of the zero-response seeds.

Verification begins by first reconstructing the challenge from the signature and decoding the monomial matrices. The responses must be processed sequentially in order to successfully recreate the challenge seed. Thus, for each response, if the challenge value is zero, then the monomial is resampled from the response seed. If it is nonzero, the decoded monomial is used. Once the response monomial is prepared, the public key generator matrix corresponding to the challenge value is decoded and multiplied with the monomial before being loaded into the RREF module. The post-processing is performed in the same manner as in signing. The next monomial and generator matrix are prepared in parallel with

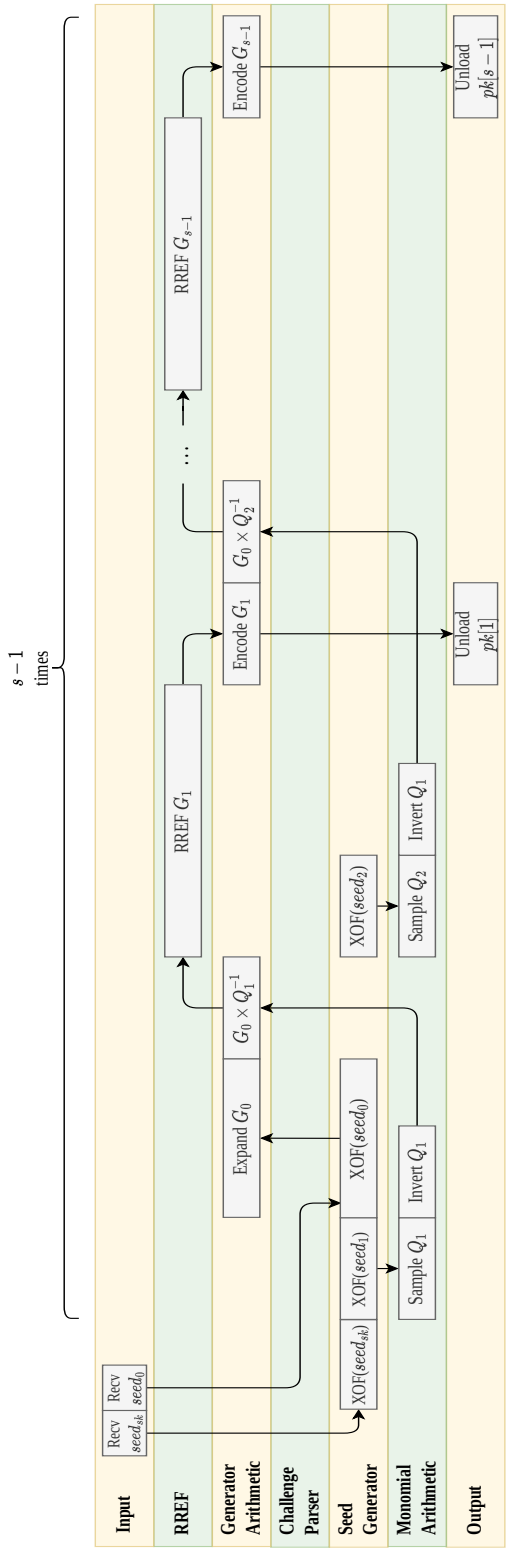

Fig. 17: Operation Scheduling for LESS Key Generation

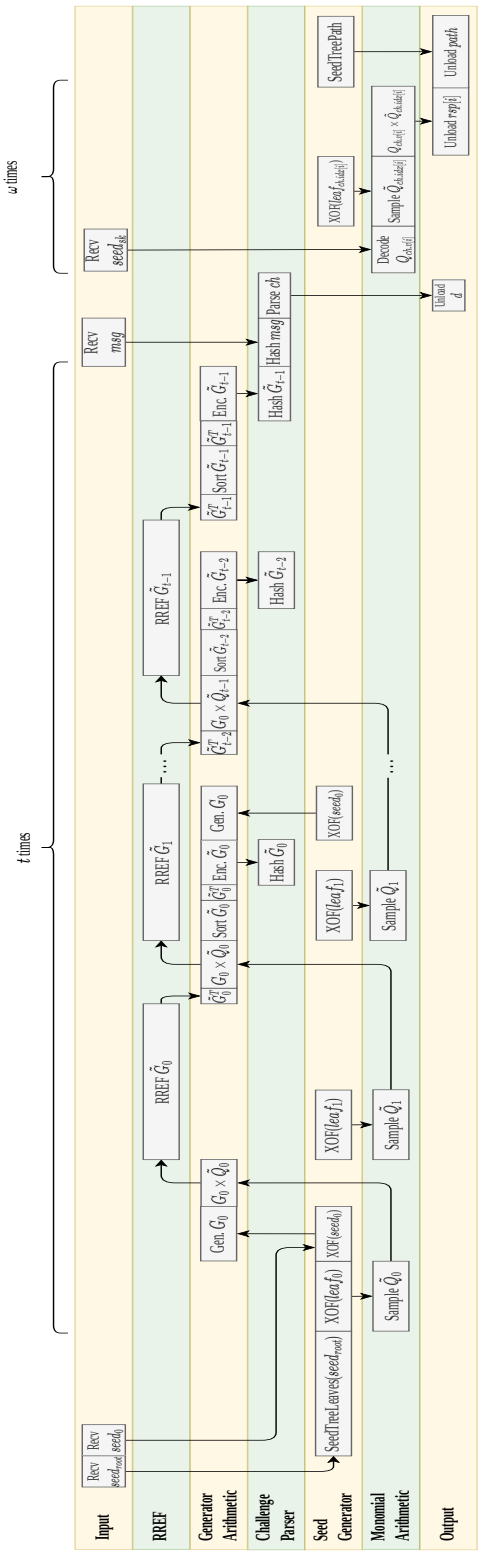

Fig. 18: Operation Scheduling for LESS Sign

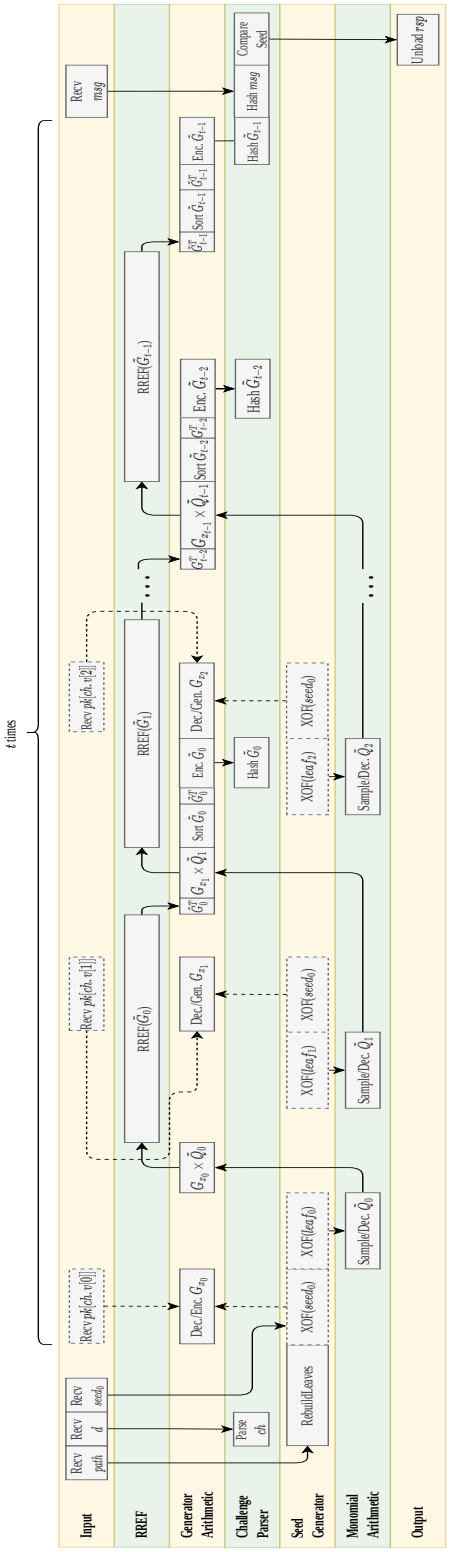

Fig. 19: Operation Scheduling for LESS Verify

Table 2: RREF Implementation Results on Artix-7

k	n	q	**Frequency** (MHz)	**LUTs** ($\times 10^3$)	**FFs** ($\times 10^3$)	**BRAM** (36 Kbit)	**Cycles** ($\times 10^3$)
126	252	127	200	27.1	26.8	26.5	16.3
200	400	127	167	40.3	43.5	41	40.7
274	548	127	143	58.8	61.2	56.5	75.9

the RREF operation. The result is encoded and ingested into the hash function. After all generator matrices and the message are hashed, the resulting hash is compared with the challenge seed. If they match, the signature is accepted. If not, it is rejected.

5 Results

Hardware performance and area results are reported for Artix-7 FPGA. The device used for generating timing and area results was XC7A200TFBG484-3. Xilinx Vivado 2022.2 was used for synthesis and implementation. Performance cycle counts were determined using simulation. All hardware implementations included for comparison also reported their area and timing for Artix-7 FPGA.

The implementation results for the RREF unit are listed in Table 2. The area and operating frequency results are dependent on n. A large n will result in a greater area, due to more parallel elements, along with a slower clock frequency, due to large multiplexers. The RREF operation in hardware is executed in constant time, where the number of clock cycles is uniquely dependent on the size of k.

The implementation results for the entire LESS scheme are provided in Tables 3 and 4. The maximum frequency of the LESS module is limited by the critical path of the RREF unit, which is dependent on the size of the generator matrix. Thus the lower parameter sets have a higher maximum frequency. The RREF module consumes the majority of resources and takes up the most significant portion of the latency. Approximately 50%-56% of the LUTs of the design are used in the RREF module, and approximately 80% of the cycles in sign and verification are spent in the RREF module. Due to the computational intensity of RREF, all other modules were able to be optimized for the low area.

With respect to area consumption, most of the change in resource consumption is related to the size of the generator matrices. In particular, the resources of RREF and generator modules scale linearly with the size of these matrices. This is because the RREF and sorting modules always perform their operations on an entire row or column in a single cycle, so the number of processing elements within these modules scales with n and k. The memory also scales directly with the dimension of the matrices. The LUT and FF resources of the remaining modules are mostly independent with respect to the size of the matrices, but the memories within these modules do increase for the larger matrices.

Table 3: Comparison of Hardware Area for Relevant PQC Implementations. *TW* refers to this work.

Algorithm	Implementer	Platform	Parameter Set	Frequency	LUTs ($\times 10^3$)	FFs ($\times 10^3$)	DSP	BRAM (36 Kbit)
LESS	TW	Artix-7	L1-{b,i,s}	200	54.8	39.9	0	59.5
			L3-{b,s}	167	76.7	57.9	0	102.5
			L5-{b,s}	143	104.3	76.7	0	167.5
FALCON	Beckwith et al	Artix-7	L1	142	14.5	7.3	4	2
			L5		13.9	6.7	4	2
Dilithium	Zhao et al	Artix-7	L2	96.9	30	10.4	10	11
			L3					
			L5					
SPHINCS+	Amiet et al	Artix-7	128s-simple	250 & 500	48.2	72.5	0	11.5
			128s-robust		49.1	73.1	0	15.5
			128f-simple		48.0	72.5	1	11.5
			128f-robust		48.9	73.0	1	15.5
			192s-simple		48.7	72.5	0	17
			192s-robust		50.1	74.5	0	22.5
			192f-simple		48.4	73.5	1	17
			192f-robust		47.2	74.3	1	22.5
			256s-simple		51.1	74.6	1	22.5
			256s-robust		50.1	75.7	1	30
			256f-simple		51.0	74.5	1	22.5
			256f-robust		50.3	75.7	1	30

Table 4: Performance Results for Relevant PQC Implementations on Artix-7. *TW* refers to this work.

Design		Algorithm Details				Performance Results					
						Keygen		Sign		Verify	
Algorithm	Implementer	Parameter Set	Public Key (KB)	Signature (KB)	Frequency	Cycles ($\times 10^3$)	Latency (μs)	Cycles ($\times 10^3$)	Latency (μs)	Cycles ($\times 10^3$)	Latency (μs)
LESS	TW	L1-b	13.7	8.4	200	29.1	145.3	5,204.6	26,023.0	5,156.2	25,780.9
		L1-i	41.1	6.1		77.5	387.7	5,126.4	25,631.8	5,093.2	25,465.8
		L1-s	95.9	5.2		174.5	872.5	4,166.1	20,830.6	4,137.2	20,685.9
		L3-b	34.5	18.4	167	72.1	432.8	39,237.4	235,424.4	39,146.0	234,875.7
		L3-s	68.9	14.1		132.8	796.7	46,216.7	277,300.0	46,142.8	276,856.9
		L5-b	64.6	32.5	143	134.4	941.1	129,885.6	909,199.5	129,726.1	908,082.6
		L5-s	129	26.1		247.9	1,735.5	87,161.5	610,130.3	87,013.8	609,096.4
FALCON	Beckwith et al	L1	0.897	0.666	142	N/A	N/A	N/A	N/A	2.4	16.8
		L5	1.79	1.28	142					4.7	32.8
Dilithium	Zhao et al	L2	1.31	2.4	96.9	4.1	41	28.1	281	4.4	44
		L3	1.95	3.3		5.9	59	44.7	447	6.2	62
		L5	2.59	4.6		8.8	88	49.0	490	9.0	90
SPHINCS+	Amiet et al	128s-simple	0.032	7.9	250 & 500	N/A	N/A	N/A	12,400	N/A	70
		128s-robust	0.032	7.9					21,100		110
		128f-simple	0.032	17.1					1,010		160
		128f-robust	0.032	17.1					1,640		230
		192s-simple	0.048	16.3					21,400		100
		192s-robust	0.048	16.3					38,300		150
		192f-simple	0.048	35.7					1,170		190
		192f-robust	0.048	35.7					2,120		310
		256s-simple	0.064	29.8					19,300		140
		256s-robust	0.064	29.8					36,100		200
		256f-simple	0.064	49.9					2,520		210
		256f-robust	0.064	49.9					4,680		340

Table 5: Performance Comparison with AVX2 Implementation

Parameter Set	Platform	Frequency	Keygen		Sign		Verify	
			Latency (μs)	HW Speedup	Latency (μs)	HW Speedup	Latency (μs)	HW Speedup
L1-b	**Artix-7**	**200 MHz**	145.3	×**1.5**	26,023.0	×**2.5**	25,780.9	×**2.5**
L1-i			387.7	×**1.4**	25,631.8	×**2.7**	25,465.8	×**2.7**
L1-s			872.5	×**1.4**	20,830.6	×**2.6**	20,685.9	×**2.6**
L3-b		**167 MHz**	432.8	×**1.4**	235,424.4	×**2.5**	234,875.7	×**2.5**
L3-s			796.7	×**1.4**	277,300.0	×**2.5**	276,856.9	×**2.5**
L5-b		**143 MHz**	941.1	×**1.4**	909,199.5	×**2.6**	908,082.6	×**2.6**
L5-s			1,735.5	×**1.3**	610,130.3	×**2.5**	609,096.4	×**2.5**
L1-b	**Software (AVX2)**	**3.9 GHz**	222.7		64,653.9		68,500.4	
L1-i			557.5		68,689.0		68,689.0	
L1-s			1,185.8		54,835.1		54,835.1	
L3-b			610.7		584,660.7		584,660.7	
L3-s			1,099.8		683,915.5		683,915.5	
L5-b			1,306.4		2,348,707.0		2,348,707.0	
L5-s			2,333.3		1,547,110.1		1,547,110.1	

5.1 Software Comparison

Table 5 provides a comparison between our hardware accelerator running on Artix-7 and the optimized AVX2 implementation running on an AMD Ryzen 5 5600G desktop CPU. The software was compiled with GCC 12.2 with `-O3 -march=native -mtune=native` optimization flags and the measurements were taken with hyperthreading and frequency-scaling (Turbo Core) disabled. The CPU was running at 3.9 GHz, which is $19.5 - 27.3\times$ faster than the hardware.

Despite this significant difference in clock frequency, the hardware outperforms the software for all parameter sets. The key generation operation is $1.4\times$ faster in hardware. Signing and verification are both $2.5\times$ faster. The performance could be increased further through the use of a higher-end FPGA, which can enable high clock frequencies, or through implementation as an ASIC.

5.2 Comparison with Other Digital Signature Schemes

In Tables 3 and 4, we provide comparisons with the best high-performance hardware architectures for CRYSTALS-Dilithium, FALCON, and SPHINCS$^+$.

When comparing both performance and area, the two lattice-based algorithms CRYSTALS-Dilithium and FALCON both outperform the implementation of LESS. Both algorithms require significantly fewer resources, with the exception of DSPs, and have significantly lower latency. The DSP usage is required in these algorithms for modular multiplication because the moduli are too large to effectively perform in LUTs without a significant increase in the critical path of the design, whereas the small modulus of LESS allows for multiplication to be implemented using LUTs.

A more relevant comparison is with SPHINCS$^+$, which is the only non-lattice-based signature scheme selected by NIST for standardization to date. Both algorithms provide parameter sets optimizing for different metrics. For SPHINCS$^+$,

the "s" and "f" notations correspond to "smaller signature but slower signing" and "faster signing but larger signature," respectively. The "simple" parameter set has higher performance but a less conservative security argument than the "robust" parameter set [5].

When comparing the parameter sets for LESS and SPHINCS$^+$, we can observe that the signature size of LESS varies from 30%-106%, 39%-112%, and 52%-109% of the size of SPHINCS$^+$'s signatures for security levels 1, 3, and 5, respectively. Both algorithms have multiple parameter sets with different tradeoffs for the signature size. The small signature parameter set of LESS always has a smaller signature size then that of SPHINCS$^+$, but the public key is significantly larger. The latency for SPHINCS$^+$ is lower for all levels except when comparing SPHINC$^+$ 128s-robust to LESS L1-s, in which LESS is slightly shorter.

The area of the LESS design is comparable to SPHINCS$^+$ for most parameter sets. LESS uses similar LUTs at level 1 to all levels of SPHINCS$^+$, slightly more at level 2, and substantially more at level 3. The flip-flop utilization of LESS is less or very similar to all parameter sets of SPHINCS+. The BRAM utilization is much larger for LESS than SPHINCS$^+$ due to the optimization of RREF operating on an entire row of the matrix.

6 Conclusions

This work presents a high-performance hardware implementation of LESS, a recently-proposed code-based digital signature scheme, which was submitted to the NIST post-quantum cryptography standardization process. A key component is a constant-time, highly parallel unit implementing conversion of an arbitrary $k \times n$ matrix over $GF(p)$ to the reduced row echelon form. This conversion is by far the most computationally intensive operation of LESS. The hardware implementation running on Artix-7 FPGA outperforms optimized software running on a modern desktop CPU by factors ranging between 1.3 and 2.7 depending on a variant and security level. The entire hardware implementation of LESS is resistant to timing attacks.

Acknowledgments. This work has been partially supported by the National Science Foundation under Grant No.: CNS-1801512 and by the US Department of Commerce (NIST) under Grant No.: 70NANB18H218.

References

1. Aikata, Mert, A.C., Jacquemin, D., Das, A., Matthews, D., Ghosh, S., Roy, S.S.: A Unified Cryptoprocessor for Lattice-based Signature and Key-exchange. IEEE Transactions on Computers pp. 1–13 (2022). https://doi.org/10.1109/TC.2022.3215064

2. Aikata, A., Mert, A.C., Imran, M., Pagliarini, S., Roy, S.S.: KaLi: A Crystal for Post-Quantum Security Using Kyber and Dilithium. IEEE Transactions on Circuits and Systems I: Regular Papers **70**(2), 747–758 (Feb 2023). https://doi.org/10.1109/TCSI.2022.3219555
3. Alagic, G., Apon, D., Cooper, D., Dang, Q., Dang, T., Kelsey, J., Lichtinger, J., Miller, C., Moody, D., Peralta, R., Perlner, R., Robinson, A., Smith-Tone, D., Liu, Y.K.: Status Report on the Third Round of the NIST Post-Quantum Cryptography Standardization Process. National Institute of Standards and Technology Interagency or Internal Report NIST IR 8413-upd1, National Institute of Standards and Technology (Jul 2022), https://doi.org/10.6028/NIST.IR.8413-upd1
4. Amiet, D., Leuenberger, L., Curiger, A., Zbinden, P.: FPGA-based SPHINCS+ Implementations: Mind the Glitch. In: 2020 23rd Euromicro Conference on Digital System Design (DSD). pp. 229–237. IEEE, Kranj, Slovenia (Aug 2020). https://doi.org/10.1109/DSD51259.2020.00046
5. Aumasson, J.P., Bernstein, D.J., Beullens, W., Dobraunig, C., Eichlseder, M., Fluhrer, S., Gazdag, S.L., Hülsing, A., Kampanakis, P., Kölbl, S., Lange, T., Lauridsen, M.M., Mendel, F., Niederhagen, R., Rechberger, C., Rijneveld, J., Schwabe, P., Westerbaan, B.: SPHINCS+ Specification v3.1 (Jun 2022), https://sphincs.org/data/sphincs+-r3.1-specification.pdf
6. Balasubramanian, S., Carter, H.W., Bogdanov, A., Rupp, A., Ding, J.: Fast Multivariate Signature Generation in Hardware: The Case of Rainbow. In: 16th International Symposium on Field-Programmable Custom Computing Machines, FCCM 2008. pp. 25–30 (Apr 2008)
7. Balasubramanian, S.R.: A Parallel Hardware Architecture for Fast Signature Generation of Rainbow. Master's thesis, University of Cincinnati, Cincinnati, OH (Oct 2007)
8. Baldi, M., Barenghi, A., Beckwith, L., Biasse, J.F., Esser, A., Gaj, K., Mohajerani, K., Pelosi, G., Persichetti, E., Saarinen, M.J.O., Santini, P., Wallace, R.: LESS: Linear Equivalence Signature Scheme, https://www.less-project.com/
9. Barenghi, A., Biasse, J.F., Persichetti, E., Santini, P.: LESS-FM: Fine-Tuning Signatures from the Code Equivalence Problem. In: Post-Quantum Cryptography, PQCrypto 2021. LNCS, vol. 12841, pp. 23–43. Springer International Publishing, Cham (2021), https://link.springer.com/10.1007/978-3-030-81293-5_2
10. Beckwith, L., Nguyen, D.T., Gaj, K.: High-Performance Hardware Implementation of CRYSTALS-Dilithium. In: 2021 International Conference on Field-Programmable Technology (ICFPT). pp. 1–10. IEEE, Auckland, New Zealand (Dec 2021). https://doi.org/10.1109/ICFPT52863.2021.9609917
11. Beckwith, L., Nguyen, D.T., Gaj, K.: High-Performance Hardware Implementation of Lattice-Based Digital Signatures. https://eprint.iacr.org/2022/217 (Feb 2022)
12. Biasse, J.F., Micheli, G., Persichetti, E., Santini, P.: LESS is More: Code-Based Signatures Without Syndromes. In: Progress in Cryptology - AFRICACRYPT 2020. LNCS, vol. 12174, pp. 45–65. Springer International Publishing, Cham (2020). https://doi.org/10.1007/978-3-030-51938-4_3
13. Bogdanov, A., Eisenbarth, T., Rupp, A., Wolf, C.: Time-Area Optimized Public-Key Engines: MQ-Cryptosystems as Replacement for Elliptic Curves? In: Cryptographic Hardware and Embedded Systems – CHES 2008. LNCS, vol. 5154, pp. 45–61. Springer, Washington, DC (Aug 2008)
14. CERG: SHAKE. https://github.com/GMUCERG/SHAKE

15. Ferozpuri, A., Gaj, K.: High-speed FPGA Implementation of the NIST Round 1 Rainbow Signature Scheme. In: 2018 International Conference on ReConFigurable Computing and FPGAs (ReConFig). pp. 1–8. IEEE, Cancun, Mexico (Dec 2018). https://doi.org/10/ggbsdm
16. Fiat, A., Shamir, A.: How To Prove Yourself: Practical Solutions to Identification and Signature Problems. In: Advances in Cryptology — CRYPTO'86. LNCS, vol. 263, pp. 186–194. Springer Berlin Heidelberg, Berlin, Heidelberg (1986). https://doi.org/10.1007/3-540-47721-7_12
17. Gupta, N., Jati, A., Chattopadhyay, A., Jha, G.: Lightweight Hardware Accelerator for Post-Quantum Digital Signature CRYSTALS-Dilithium. IEEE Transactions on Circuits and Systems I: Regular Papers pp. 1–10 (2023). https://doi.org/10.1109/TCSI.2023.3274599
18. Hochet, B., Quinton, P., Robert, Y.: Systolic solution of linear systems over GF(p) with partial pivoting. In: 1987 IEEE 8th Symposium on Computer Arithmetic (ARITH). pp. 161–168. IEEE, Como, Italy (May 1987). https://doi.org/10/ggbsf8
19. Hochet, B., Quinton, P., Robert, Y.: Systolic Gaussian Elimination over GF(p) with Partial Pivoting. IEEE Transactions on Computers **38**(9), 1321–1324 (Sep 1989). https://doi.org/10/ckc8nj
20. Karl, P., Schupp, J., Fritzmann, T., Sigl, G.: Post-Quantum Signatures on RISC-V with Hardware Acceleration. ACM Transactions on Embedded Computing Systems (Jan 2023). https://doi.org/10.1145/3579092
21. Land, G., Sasdrich, P., Güneysu, T.: A Hard Crystal - Implementing Dilithium on Reconfigurable Hardware. In: International Conference on Smart Card Research and Advanced Applications, CARDIS 2021. LNCS, vol. 13173, pp. 210–230. Springer International Publishing, Cham (Mar 2022). https://doi.org/10.1007/978-3-030-97348-3_12
22. Nannipieri, P., Di Matteo, S., Zulberti, L., Albicocchi, F., Saponara, S., Fanucci, L.: A RISC-V Post Quantum Cryptography Instruction Set Extension for Number Theoretic Transform to Speed-Up CRYSTALS Algorithms. IEEE Access **9**, 150798–150808 (2021). https://doi.org/10.1109/ACCESS.2021.3126208
23. NSA: Cybersecurity Advisory Announcing the Commercial National Security Algorithm Suite 2.0 (Sep 2022), https://media.defense.gov/2022/Sep/07/2003071834/-1/-1/0/CSA_CNSA_2.0_ALGORITHMS_.PDF
24. Persichetti, E.: LESS: Digital Signatures from Linear Code Equivalence. https://csrc.nist.gov/Projects/post-quantum-cryptography/workshops-and-timeline/pqc-seminars (Mar 2023)
25. Preucil, T.: Implementation of the Signature Scheme Rainbow on SoC FPGA. Master's thesis, Uppsala University, Uppsala, Sweden (2022), http://urn.kb.se/resolve?urn=urn:nbn:se:uu:diva-484811
26. Preucil, T., Socha, P., Novotny, M.: Implementation of the Rainbow signature scheme on SoC FPGA. In: 2022 25th Euromicro Conference on Digital System Design (DSD). pp. 513–519. IEEE, Maspalomas, Spain (Aug 2022). https://doi.org/10.1109/DSD57027.2022.00074
27. Rupp, A., Eisenbarth, T., Bogdanov, A., Grieb, O.: Hardware SLE solvers: Efficient building blocks for cryptographic and cryptanalytic applications. Integration **44**(4), 290–304 (Sep 2011). https://doi.org/10.1016/j.vlsi.2010.09.001
28. Shor, P.: Algorithms for quantum computation: Discrete logarithms and factoring. In: Proceedings 35th Annual Symposium on Foundations of Computer Science. pp. 124–134. IEEE Comput. Soc. Press, Santa Fe, NM, USA (1994). https://doi.org/10/czq562

29. Tang, S., Yi, H., Ding, J., Chen, H., Chen, G., Chen, G.: High-Speed Hardware Implementation of Rainbow Signature on FPGAs. In: Post-Quantum Cryptography, PQCrypto 2011. vol. 7071, pp. 228–243. Springer Berlin Heidelberg, Berlin, Heidelberg (2011). https://doi.org/10.1007/978-3-642-25405-5_15
30. Yi, H., Li, W.: Small FPGA implementations for solving systems of linear equations in finite fields. In: 2015 6th IEEE International Conference on Software Engineering and Service Science (ICSESS). pp. 561–564. IEEE, Beijing, China (Sep 2015). https://doi.org/10/ggbsgc
31. Yi, H., Nie, Z.: High-speed hardware architecture for implementations of multivariate signature generations on FPGAs. EURASIP Journal on Wireless Communications and Networking **2018**(1), 93 (Dec 2018). https://doi.org/10.1186/s13638-018-1117-2
32. Zhao, C., Zhang, N., Wang, H., Yang, B., Zhu, W., Li, Z., Zhu, M., Yin, S., Wei, S., Liu, L.: A Compact and High-Performance Hardware Architecture for CRYSTALS-Dilithium. IACR Transactions on Cryptographic Hardware and Embedded Systems **2022**(1), 270–295 (Nov 2021). https://doi.org/10.46586/tches.v2022.i1.270-295
33. Zhao, Y., Xie, R., Xin, G., Han, J.: A High-Performance Domain-Specific Processor With Matrix Extension of RISC-V for Module-LWE Applications. IEEE Transactions on Circuits and Systems I: Regular Papers **69**(7), 2871–2884 (Jul 2022). https://doi.org/10.1109/TCSI.2022.3162593
34. Zhou, Z., He, D., Liu, Z., Luo, M., Choo, K.K.R.: A Software/Hardware Co-Design of Crystals-Dilithium Signature Scheme. ACM Transactions on Reconfigurable Technology and Systems **14**(2), 11:1–11:21 (Jun 2021). https://doi.org/10.1145/3447812