High-Performance Hardware Implementation of CRYSTALS-Dilithium

Luke Beckwith, Duc Tri Nguyen, Kris Gaj
George Mason University, USA
{lbeckwit, dnguye69, kgaj}@gmu.edu

***Abstract*—Many currently deployed public-key cryptosystems are based on the difficulty of the discrete logarithm and integer factorization problems. However, given an adequately sized quantum computer, these problems can be solved in polynomial time as a function of the key size. Due to the future threat of quantum computing to current cryptographic standards, alternative algorithms that remain secure under quantum computing are being evaluated for future use. One such algorithm is CRYSTALS-Dilithium, a lattice-based digital signature scheme, which is a finalist in the NIST Post Quantum Cryptography (PQC) competition. As a part of this evaluation, high-performance implementations of these algorithms must be investigated. This work presents a high-performance implementation of CRYSTALS-Dilithium targeting FPGAs. In particular, we present a design that achieves the best latency for an FPGA implementation to date. We also compare our results with the most-relevant previous work on hardware implementations of NIST Round 3 post-quantum digital signature candidates.**

***Index Terms*—Post-Quantum Cryptography, Digital Signature, Number Theoretic Transform, FPGA**

I. Introduction

Current public key cryptographic standards, such as RSA and ECC, rely on the difficulty of the integer factorization and the discrete logarithm problem. However, with a sufficiently large quantum computer, Shor's algorithm [1] can be applied to solve these problems in superpolynomial run time [2]. While there is no known quantum computer capable of running Shor's algorithm to break current public-key standards, the process of selecting, standardizing, and deploying a new public-key cryptosystems may take a substantial amount of time. In 2016, NIST announced the Post-Quantum Cryptography (PQC) standardization process aimed at developing new public-key standards resistant against quantum computers. In July 2020, NIST selected seven third-round finalists, including two lattice-based digital signature schemes: CRYSTALS-Dilithium and FALCON. All remaining candidates are currently being evaluated in terms of their security, key and ciphertext/signature size, performance in both software and hardware, and several other criteria. While software implementations are relatively easy to develop and benchmark, hardware implementations require a lot of time and design effort to determine their efficiency in terms of speed, area, power, and energy. FPGA implementations of PQC submissions are necessary to provide insight into the cost and performance of these algorithms in hardware.

Contributions. In this work, we present a high-performance FPGA implementation of CRYSTALS-Dilithium designed at the Register Transfer Level (RTL) using Verilog. In particular, we make the following contributions:

- We present the first combined architecture capable of performing key generation, signature generation, and signature verification and selecting between security levels at runtime.
- Our design achieves the lowest latency for all operations while maintaining a smaller area than existing high-performance implementations.
- We present a new signature generation implementation approach based on splitting the signature generation rejection loop into a two-stage pipeline to balance latency and area utilization.

II. Previous Work

The existing hardware and software/hardware implementations of digital signatures schemes qualified to Round 3 of the NIST PQC standardization process are summarized in Table I. It is worth noting that GeMSS and FALCON are also candidates still under consideration, but they lack any reported hardware implementations and thus are not included in our comparisons. Hardware design may focus either on maximizing performance or minimizing the area and power the design consumes. These approaches lead to substantial differences in hardware architecture, and thus both types of implementations are important to measure an algorithm's performance in hardware. Therefore, the implementations in Table I are split into High-Speed or Lightweight. However, the dividing line is not always apparent as designs may seek to make area/performance trade offs. These labels are to help differentiate the results of various implementations, but should not be considered absolute.

There are currently two existing full hardware designs for Dilithium. In [3], a high-performance design for Dilithium version 2 [10] is described. The authors report area for individual modules instantiated for level 3 and performance for security levels 1-4. They utilized a 2×2 NTT butterfly arrangement to calculate two layers of the NTT at a time. This NTT module is duplicated several times within the key generation, sign, and verify modules to improve performance through parallelization. Another implementation, [6], describes a mid-range implementation for version 3.1 of Dilithium, which focuses on achieving the best performance possible with reasonable resource utilization. This design consists of three top-level modules each capable of performing all operations

TABLE I: Hardware implementations reported for Round 3 digital signature schemes

Algorithms	High-Speed	Lightweight
Lattice-based		
CRYSTALS-Dilithium	[3]	[4], [5]*, [6]
Multivariate		
Rainbow	[7]**	–
Symmetric-based		
Picnic	[8]	–
SPHINCS+	[9]	–

*extended version of [4]
**only for Round 1 and Round 2 parameter sets

at a single security level, as well as individual module which only perform a single operation at a single security level.

III. Background

A. Number Theoretic Transform

The Number Theoretic Transform (NTT) is a form of the Fast Fourier Transform (FFT) which can be efficiently performed over the ring $R_q = \mathbb{Z}_q[x]/(x^n + 1)$ in $O(n \log n)$, where n is power of 2 and $q \equiv 1 (\text{mod } 2n)$. Polynomial multiplication can be efficiently calculated using the NTT as follows: $a \times b = NTT^{-1}(NTT(a) \circ NTT(b))$, where $\circ$ is point-wise multiplication of the polynomial coefficients. This approach reduces the complexity of polynomial multiplication from $O(n^2)$ to $O(n \log n)$. By default, Dilithium uses Number Theoretic Transform (NTT) as a part of `MatrixVectorMul` (matrix-by-vector) $A \cdot s$ and `InnerProd` (vector-by-vector) $s \cdot s'$ multiplication. With $(s, s') \in R_q$ and $A \in R_q^{k \times l}$, where $k \times l$ denote the dimensions of matrix A.

B. Dilithium Overview

Dilithium is a member of the Cryptographic Suite for Algebraic Lattices (CRYSTALS) along with the Key Encapsulation Mechanism (KEM) Kyber. The core operations of Dilithium are the arithmetic of polynomial matrices and vectors. Unlike many other Module Learning with Errors (M-LWE) cryptosystems, all polynomials in Dilithium are uniformly sampled, which greatly simplifies polynomial generation. As described in [11], Dilithium is a Fiat-Shamir with Aborts [12], [13] style signature scheme, and bases its security upon the M-LWE and Shortest Integer Solution (SIS) problems. The M-LWE problem can be briefly described as follows: Let $A \in R_q^{k \times l}$ be uniformly chosen, $s_1 \in R_q^l$, and $s_2 \in R_q^k$. Then, the standard M-LWE problem is to distinguish $(A, A \cdot s_1 + s_2)$ from (A, u) where u is a uniformly chosen vector. The SIS problem can be briefly described as follows: Let A be a uniformly chosen $R_q^{k \times l}$ matrix, then find a non-zero vector $x \in R_q^l$ such that the norm of x is less than β for some β and $A \cdot x = 0$.

The three core algorithms of Dilithium are key generation, signature generation, and signature verification. Key generation creates a public and secret key pair composed of polynomials and byte-array seeds. These keys are used for

Algorithm 1: Dilithium Key Generation

```
Input: $\zeta \in \{0,1\}^{256}$
Output: $pk = (\rho, t_1)$, $sk = (\rho, K, tr, s_1, s_2, t_0)$
1 $(\rho, \sigma, K) \leftarrow H(\zeta)$
2 $A \leftarrow$ExpandA$(\rho)$    $s_1, s_2 \leftarrow$ExpandS$([(\sigma, 0), (\sigma, l)])$
3 $t \leftarrow A \cdot s_1 + s_2$
4 $(t_0, t_1) \leftarrow$ Power2Round$(t, d)$
5 $tr \leftarrow \{0,1\}^{256} := H(\rho || t_1)$
```

signature creation and verification. To generate the signature, the inputs are the private key and message of the sender. Then, the receiver can use the public key of the sender and the message to verify the signature.

As seen in Algorithm 1, key generation closely resembles the description of the M-LWE problem with (A, t) serving as the public key with two optimizations: the seed ρ is used instead of the actual **A** matrix, and only the upper bits of t are included. This second change nearly halves the size of the key, but requires that a small hint is added to the signature. The only modifications to the secret key are the inclusion of the lower bits of t and two byte-arrays K, tr used in sign.

Algorithm 2: Dilithium Signature Generation

```
Input: $sk = (\rho, K, tr, s_1, s_2, t_0), M \in \{0,1\}^*$
Output: $\sigma = (\hat{c}, z, h)$
1 $\mathbf{A} \leftarrow$ExpandA$(\rho)$    $\mu \leftarrow H(tr || M)$    $\rho' \leftarrow H(K || \mu)$
2 $k \leftarrow 0$    $abort \leftarrow 1$
3 while $abort$ do
4     $abort \leftarrow 0$
5     $y \leftarrow$ExpandMask$(\rho', k)$
6     $w \leftarrow \mathbf{A} \cdot y$
7     $w_1 \leftarrow$HighBits$_q(w, 2\gamma_2)$
8     $\hat{c} \leftarrow H(\mu || w_1)$
9     $c \leftarrow$SampleInBall$(\hat{c})$
10    $z \leftarrow y + c \cdot s_1$
11    $r_0 =$LowBits$(w - c \cdot s_2, 2\gamma_2)$
12    if $||z||_\infty \geq \gamma_1 - \beta$ or $||r_0||_\infty \geq \gamma_2 - \beta$ then
13        $abort \leftarrow 1$
14    else
15        $h \leftarrow$MakeHint$(-c \cdot t_0, w - c \cdot s_2 + c \cdot t_0, 2\gamma_2)$
16        if $||c \cdot t_0||_\infty \geq \gamma_2$ or $\sum h_i > \omega$ then
17            $abort \leftarrow 1$
18    $k = k + l$
```

In Algorithm 2, signature generation is described. This is the most complex and costly operation of Dilithium. The goal when signing is to generate a polynomial pair $(c, z = y + cs_1)$ using the secret key and message. However, since the z is closely related to the s_1, the algorithm must check that the signature does not leak information about this secret value. This is done by checking that the max norm of the polynomial is within an acceptable predefined range. If it is not, the signature must be rejected, and a new attempt is made. The max norm of a vector is represented as $||x||_\infty$ and is the

Algorithm 3: Dilithium Signature Verification

```
Input: $pk = (\rho, t_1)$, $M \in \{0,1\}^*, \sigma = (\hat{c}, z, h)$
Output: Valid or Invalid
1 $\mathbf{A} \leftarrow$ExpandA$(\rho)$
2 $\mu \leftarrow H(H(\rho \| t_1) \| M)$
3 $c \leftarrow$SampleInBall$(\hat{c})$
4 $(w_1, w_0) \leftarrow$UseHint$(h, A \cdot z - c \cdot t_1 \cdot 2^d)$
5 if $||z||_\infty < \gamma_1 - \beta$ & $\hat{c} = H(\mu || w_1)$ & $\sum h_i \leq \omega$ then
6     return Valid
7 return Invalid
```

coefficient in the polynomial vector with the largest absolute value. As discussed in key generation, a hint is also needed to ensure the signature can be verified. The hint specifies which coefficients of t_1 require a carry bit during the verification algorithm. This adds another rejection condition, since there is a maximum number of hints that the signature can support. Depending on the security level, between 3 and 5 attempts are required on average to generate a valid signature.

In Algorithm 3, verification attempts to recreate the $\hat{c}$ seed using the message, public key, and signature. Presuming all inputs are valid, w_1 can be recreated as follows:

$$\begin{aligned} & w = A \cdot z - c \cdot t = A \cdot (y + c \cdot s_1) - c \cdot (A \cdot s_1 + s_2) \\ \Longrightarrow\ & w = A \cdot y - c \cdot s_2 \end{aligned}$$

The higher order bits of this value can then be hashed with μ which is generated using the public key and message. Since s_2 and c have small coefficients, it will not contribute to the upper bits and the hash will match the value generated in sign. In practice, the polynomials are not sent directly in their polynomial form but are instead serialized into a byte array to more efficiently pack the keys and signatures.

C. Dilithium Parameters

First submitted to the PQC competition as [14], Dilithium has gone through several parameter modifications during the NIST PQC standardization process. The current version detailed in [11] was adjusted to better fit the NIST security levels than the previous round 2 submission [10], primarily by adjusting the k, l dimension parameters. The parameters for the current version can be seen in Table II. All versions of Dilithium make use of the SHAKE128 and SHAKE256 operation modes of the Keccak hash function, standardized in [15]. These functions are used for two applications: generation of pseudorandom data for all uniform sampling of polynomials and for hashing. SHAKE128 is used for sampling of the **A** matrix and SHAKE256 for all other sampling and hashing.

IV. Hardware Designs

We present architectures for key generation, signature generation, and signature verification. All operations are supported at security levels 2, 3, and 5, as defined in Bai et al. [11] and Table II. This section will discuss the main subcomponents used to implement the Dilithium algorithms and the high-level

TABLE II: Dilithium parameters for version 3.1 at all security levels 2, 3, 5.

Parameter	**Value**		
	2	**3**	**5**
q **[modulus]**		$2^{23} - 2^{13} + 1$	
d **[dropped bit from t]**		13	
τ **[# of +/- 1's in c]**	39	49	60
ω **[max # of 1's in hint]**	80	55	74
(k, l) **[Vector Dimensions]**	(4,4)	(6,5)	(8,7)
η **[secret range]**	2	4	2

implementation and scheduling of the design. The high-level block diagram is shown in Fig. 3.

Within Dilithium, there is a substantial amount of data dependence between operations, which allows little room for high-level parallelization. To improve performance: 1. We optimize our polynomial arithmetic units as much as possible to decrease the latency of any individual operation; 2. We split the rejection loop in the signature generation unit into a two-stage pipeline similar to the work performed in Beckwith et al. [16]. We also parallelize the other submodules as needed to minimize the stall and wait delays in polynomial arithmetic modules.

A. Polynomial Arithmetic

In our design, the polynomial arithmetic unit, PolyArith, is responsible for polynomial multiplication (including the NTT), addition, and subtraction. The NTT operation is used to accelerate the polynomial multiplication operation as appropriate. For multiplication, Barrett reduction is used as it can be implemented efficiently in hardware through the use of shifts and additions in place of constant multiplication. Our design utilizes four butterfly units, each capable of performing the basic arithmetic operation as well as the Cooley-Tukey and Gentlemen-Sande butterfly operations. This design choice was centered around the NTT as it is the most costly and complex polynomial operation. With four butterflies, our design can make use of a 2×2 butterfly arrangement, which reduces the cost of memory access in the NTT by processing two layers at once. Between every layer, the NTT pipeline must be stalled to wait for the write back to complete. Our 2×2 arrangement reduces the number of write-back conflicts and is further combined with coefficient rearrangement, reducing stall lengths. This is an optimal tradeoff, as it provides better performance than processing a single layer at once but does not utilize excessive resources. The next viable arrangement would be to use 16 butterflies in a 4×4 arrangement. However, this would increase the area by a factor of 4 and only provide minimal performance improvement as our 2×2 approach removes almost all stalls. Since the butterfly hardware is reused for all polynomial arithmetic, four coefficients are processed in parallel for all operations.

1) 2×2 NTT: Our design for the 2×2 NTT unit, shown in Fig. 1, builds upon the pipelined NTT design by Nguyen et al. [17], [18], which processed 4 coefficients per clock cycle and 2 NTT layers at once. The ideal cost of Forward and Inverse NTT is $\frac{n}{4} \cdot \frac{\log n}{2}$ clock cycles, where n is the degree

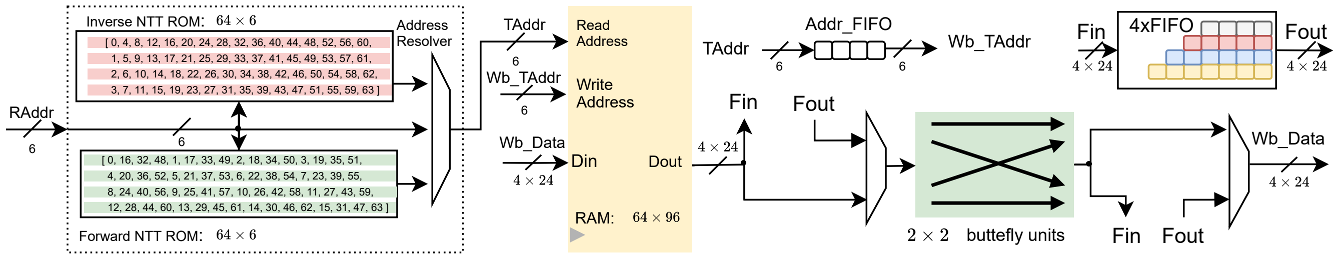

Fig. 1: 2×2 NTT with Address Resolver, $4\times$FIFO for data, Addr_FIFO for indices

of the polynomial. The BRAM stores 4 coefficients per row and uses $n/4$ rows. In case of Dilithium $n = 256$. Thus, the polynomial configuration is 64×4 coefficients, corresponding to the 64×96-bit RAM. Instead of multiplying with n^{-1} in the last stage of Inverse NTT, we incorporate the divide-by-2 technique [19] into the four 2×2 butterfly units at every level of the Inverse NTT. Each butterfly unit is deeply pipelined and uses DSP units to improve critical path. Please note that the Forward and the Inverse NTT use Decimation-In-Frequency (DIF) and Decimation-In-Time (DIT) variant of NTT, respectively. Hence, all butterfly units in our design must support both DIF and DIT, so that a single 2×2 NTT unit is capable of performing both Forward and Inverse NTT.

2) Address Resolver: The Address Resolver unit, shown in Fig. 1, is responsible for converting the Representation Address (`RAddr`) in the NTT algorithm to True Address (`TAddr`) at the first time read in RAM. It does so by using two 64×6 ROMs to map the address or by passing the input without any change when performing an operation which does not need any address resolving. Contents of the respective ROMs, Forward NTT ROM and Inverse NTT ROM, are shown Fig. 1. To determine the contents of ROMs, we examined the order of indices before and after the NTT transform, such that the conversion between `RAddr` and `TAddr` guarantees that the RAM words `RAddr` refers to are always correct. The construction of the mapping tables depends only on the parameter n and the writeback pattern of the FIFO buffer, which are fixed at runtime. We decided to use ROM-based approach since the entire ROM content is able to fit in a single 6-input LUT. It should be noted that the ROM content can also be computed on the fly using the Algorithm 4. With `Address Resolver` unit, we completely eliminate the execution time of a shuffle and re-ordering at the cost of the negligible amount of extra memory.

3) FIFO buffer: One challenge in DIT and DIF NTT is to satisfy the distance between indices for all NTT levels. Hence, we improved from the NTT design by Nguyen et al. [17], [18] by allocating five linear shift register-based FIFO units to prepare addresses and data for the next NTT levels. As shown in Fig. 1, the tiles inside FIFO units are 24-bit and 6-bit for FIFO data and address, respectively. The `Addr_FIFO` unit is responsible for delaying `TAddr` by 4 clock cycles (which is the depth of the pipeline). The remaining 4 registered-based FIFOs (sharing the same unit) are responsible for transposing

Algorithm 4: Address Resolver ROM calculation

```
Input: Representation address (RAddr) $i$
Output: True address (TAddr)
1 if mode = Forward NTT then
2   | return $(i \bmod 4) \times 16 + i/4$
3 else if mode = Inverse NTT then
4   | return $(i \bmod 16) \times 4 + i/16$
5 else
6   | return $i$
```

a 4×4 matrix of intermediate data. This transpose operation must be executed before data enters the butterfly unit in Forward NTT (DIF) and after it exits this unit in Inverse NTT (DIT). The same $4\times$FIFO unit is used for both of these operations. Internally, this unit is composed of registers and an output MUX. It is capable of storing data in rows and reading them in columns, and vice versa.

B. BRAM Configuration

As discussed in previous sections, the polynomial arithmetic modules process four coefficients per clock cycle. Thus, the bandwidth requirement is $96 = 4 \times 24$ bits. The smallest true dual port BRAM configuration that can accommodate this width in today's Xilinx FPGAs is composed of three 36-kbit BRAMs, each configured as 1024x36 memory [20]. This configuration can efficiently store two vectors of polynomials at the security level 5, with the $4 \cdot 24/3 \cdot 36 = 89\%$ utilization of each memory row. This structure allows us to efficiently utilize BRAM for polynomial storage, leading to lower BRAM utilization than previously reported implementations.

C. Polynomial Samplers

In Dilithium, the polynomials composing the vectors and matrices are independently sampled using a constant seed value and an appended incriminating nonce value as the input to either `SHAKE128` or `SHAKE256`. This allows parallel sampling of polynomials if multiple Keccak cores are used. A single Keccak core consumes a substantial amount of resources as shown in Table III. However, Dilithium requires a large amount of pseudorandom data to perform polynomial sampling. For example, at security level 5, the matrix $\mathbf{A}$ has dimension 8×7, with each sample requiring 24 bits of pseudorandom data, which amounts to at least

$8 \times 7 \times n \times 24 = 56 \times 256 \times 24 = 344$-kbit in total, not taking into account the rejection rate of each sampling. Producing that amount of pseudorandomness quickly becomes the performance bottleneck in high-performance designs. To improve the performance of sampling, we propose to use three Keccak cores. Two of them are primarily used for sampling of the matrix $\mathbf{A}$ and the third is used for the remaining hashing and sampling. The Keccak cores are taken from the existing implementation [21], which has a 64-bit datapath. In addition, we add multiple rejection lanes for vectors and matrices to process the pseudorandom data in parallel. The number of rejection lanes is chosen to be the minimum number that maximizes utilization of the corresponding Keccak core. For example, sampling a coefficient of $\mathbf{A}$ requires 24 bits, while Keccak is capable of producing 64-bits per cycle. Therefore three rejection lanes are used, so that their combined throughput (*72-bits/cc*) is higher than the Keccak core's throughput. This prevents the Keccak core from having to stall during sampling. Either 2 or 3 coefficients are processed per clock cycle, allowing the Keccak core to operate at maximum throughput. A similar approach is used for the y vector. The only exception to this approach is the c polynomial which must be sampled using the Fisher-Yates shuffle [11].

V. Operation Scheduling

Thoroughly designed and optimized scheduling of operations is crucial for efficient hardware implementations. Creating a high-performance design with an acceptable design area is only possible with high utilization of components and constant progress on the critical path of the operation. As such, we have highly optimized our operation scheduling to maximize utilization of the polynomial arithmetic units, which are the core of our design and are responsible for the majority of the operations in Dilithium.

A. Key Generation

The schedule of operations for key generation can be seen in Fig. 2. The longest path for key generation is the computation, packing, and hashing of the polynomial vector t. As such, our schedule aims to minimize any delays in the computation of t by immediately sampling s_1 and matrix A so the `NTT` transform of vector $\hat{s}_1$ and $\hat{t} = \hat{A} \cdot \hat{s}_1$ can be performed as soon as possible. t is then encoded and hashed in parallel with the addition operation so no additional time is needed to calculate *tr*. `Pack_t1` and `Pack_t0` correspond to the function `Power2Round` called in line 4 of Algorithm 1.

B. Signature Generation

The schedule for the signature generation is split into two sections: the precomputation stage in Fig. 4 and the rejection loop stage, where multiple signature attempts are run in parallel, in Fig. 5. Our design can be thought of as executing three unique stages: 1. *Precomputation* is the stage where the secret key values are unpacked and transformed into the NTT-domain 2. *Stage0* computes the y and w vectors, and 3. *Stage1* uses the results of *Stage0* to generate the z vector and the hint. The purpose of this design is to reduce the latency between signature attempts. Initially, *Precomputation* and *Stage0* are run in parallel as shown in Fig. 4. After they complete their respective calculation, *Stage0* and *Stage1* run in parallel acting as a 2-stage pipeline. Whenever *Stage1* completes, it checks if any component of the signature has violated a rejection condition and, if so, it restarts with the new results from *Stage0*. If not, *Stage1* signals *Stage0* to halt and unloads the signature. The computation of the $\hat{y}$ and w vectors was determined to be the best place to split the pipeline because it minimizes the number of polynomials that must be handed off to the next stage of the pipeline while maintaining a reasonably well balance pipeline for all security levels. For security level 3 and 5 the vector NTT and matrix multiplication operations become the dominant operations and thus splitting the number of these operations evenly between the two stages keeps the pipeline balanced.

We found that splitting computations into two stages gives better average-case performance for signature generation. The straightforward approach would be to linearly accelerate operations in the rejection loop by improving the performance of the individual operations. For example, this acceleration could be done by using multiple polynomial arithmetic units to perform computations on different elements of a single vector to achieve lower latency. However, there are two drawbacks to the linear acceleration approach. First, it substantially complicates the BRAM utilization since more parallel access to vector coefficients is required. A single vector would need to be split between multiple BRAM modules, making it more difficult to efficiently utilize BRAM. Second, some operations are not possible to parallelize, such as the sampling of c. In the linear approach, both the polynomial arithmetic units would have to stall during this operation. The linear approach does have a better best-case performance where the signature is accepted on the first attempt. However, the average number of repetitions required for the rejection loop is 3.85-5.1 depending on the security level [11]. Therefore, the shorter latency for the rejection loop in the pipelined approach gives better average-case and worst-case performance.

C. Signature Verification

The verification schedule is shown in Fig. 6. The longest path for verification is the calculation and hashing of w. Similarly to key generation, z and t_1 are immediately unpacked in parallel with the generation of $\mathbf{A}$ so that calculation of w' and w'' can be performed as soon at the polynomial arithmetic unit completes the `NTT` operations. Once w is calculated, the hint is applied to the higher-order bits so that it can be hashed with μ and compared with the challenge seed c' to determine if the signature is valid.

VI. Results

All results were generated using Xilinx Vivado 2020.2. Maximum clock frequencies were determined using the Minerva hardware optimization tool [22]. The critical path of the

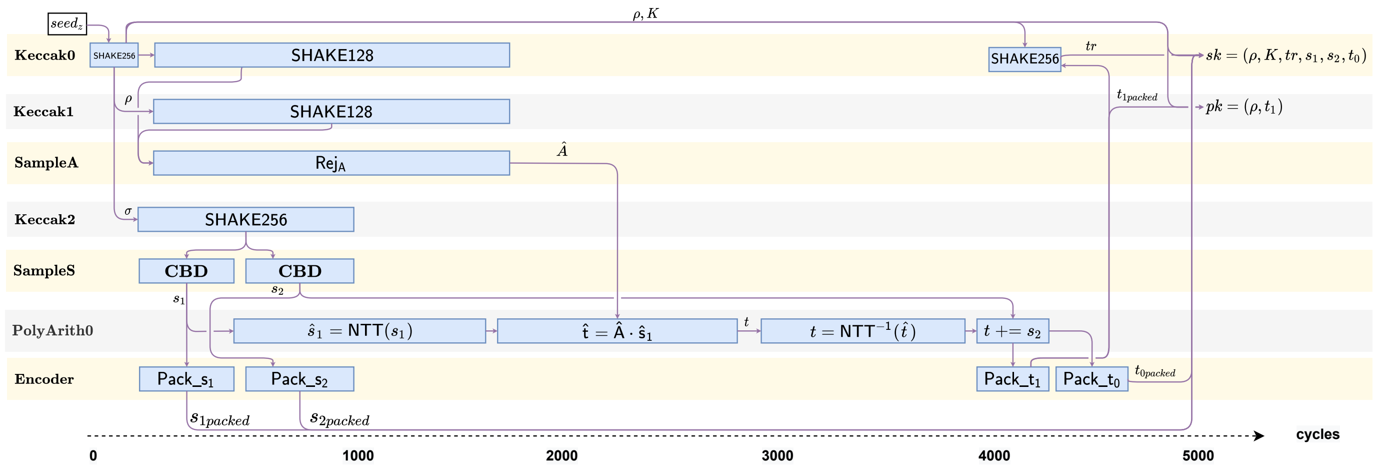

Fig. 2: Schedule of Key Generation for security level 2

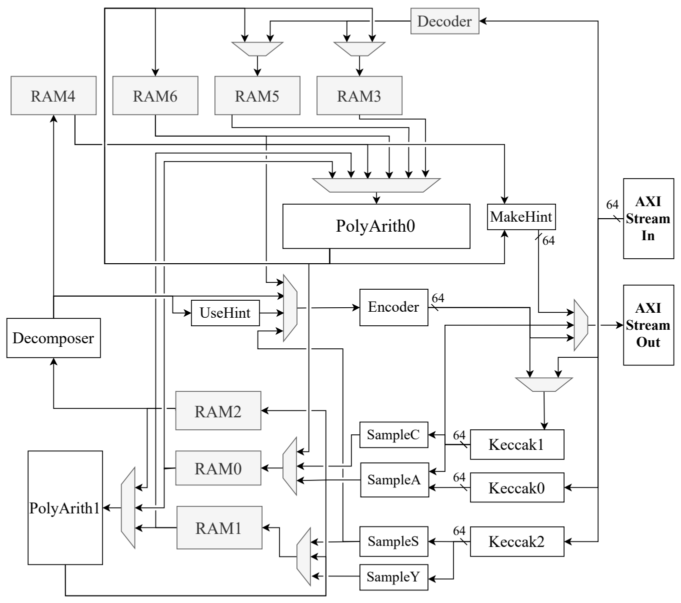

Fig. 3: Block diagram for the combined architecture of CRYSTALS-Dilithium. Bus widths are 96 bits unless shown otherwise.

TABLE III: Resource utilization of submodules in the combined architecture

Submodule	Resource Utilization				% of Total
	LUT	**FF**	**DSP**	**BRAM**	**(LUT)**
96×1024 **RAM**	0	0	0	3×6	0
96×4096 **RAM**	0	0	0	11	0
MakeHint	2,389	740	0	0	4.5
UseHint	6,453	2,808	0	0	12.1
Encoder	1,626	461	0	0	3.1
Decoder	2,189	239	0	0	4.1
Decomposer	1,437	680	0	0	2.7
NTT/PolyArith	$4,509 \times 2$	$3,146 \times 2$	16	0	16.9
SampleA	1,793	619	0	0	3.4
SampleS	1,755	396	0	0	3.2
SampleY	2,220	630	0	0	4.2
SampleC	1,856	868	0	0	3.5
Keccak	$5,483 \times 3$	$4,451 \times 3$	0	0	30.1
Other	6,002	1,231	0	0	11.3
Combined Architecture	53,187	28,318	16	29	100.0

TABLE IV: Resource utilization of top-level modules implementing major operations of CRYSTALS-Dilithium for all security levels

Module	LUT	FF	DSP	BRAM
Keygen	29,021	18,952	8	18.5
Sign	42,440	24,419	16	29
Verify	39,341	22,743	8	20
Combined	53,187	28,318	16	29

design is within the interconnect for the shared Keccak modules. Since our design targets high performance, we primarily report our results for Virtex Ultrascale+. However, we also include selected results for the Artix-7 and Kintex-7 FPGAs to perform fair comparison with previous work.

The detailed resource utilization of our implementations is summarized in Tables III and IV. Table III reports the area breakdown of the submodules used in our design and the percentage of the total LUTs they consume in the combined architecture. The entry "Other" represents the entire control logic of the top-level module and minor components of the datapath not listed explicitly in the table. Table IV shows the results for both our combined architecture and the individual modules that only perform one major operation.

There is a substantial amount of resource sharing possible between the implementations of three major operations, with the combined architecture only consuming 48% of the sum of the LUTs of the individual modules. The limited area increase over the most complex signature generation module is due to specific units only being required for certain operations, such as the secret sample, SampleS, which is only utilized by key generation. However, many modules such as the Keccak cores, polynomial arithmetic modules, and RAMs can be fully shared between the different operations.

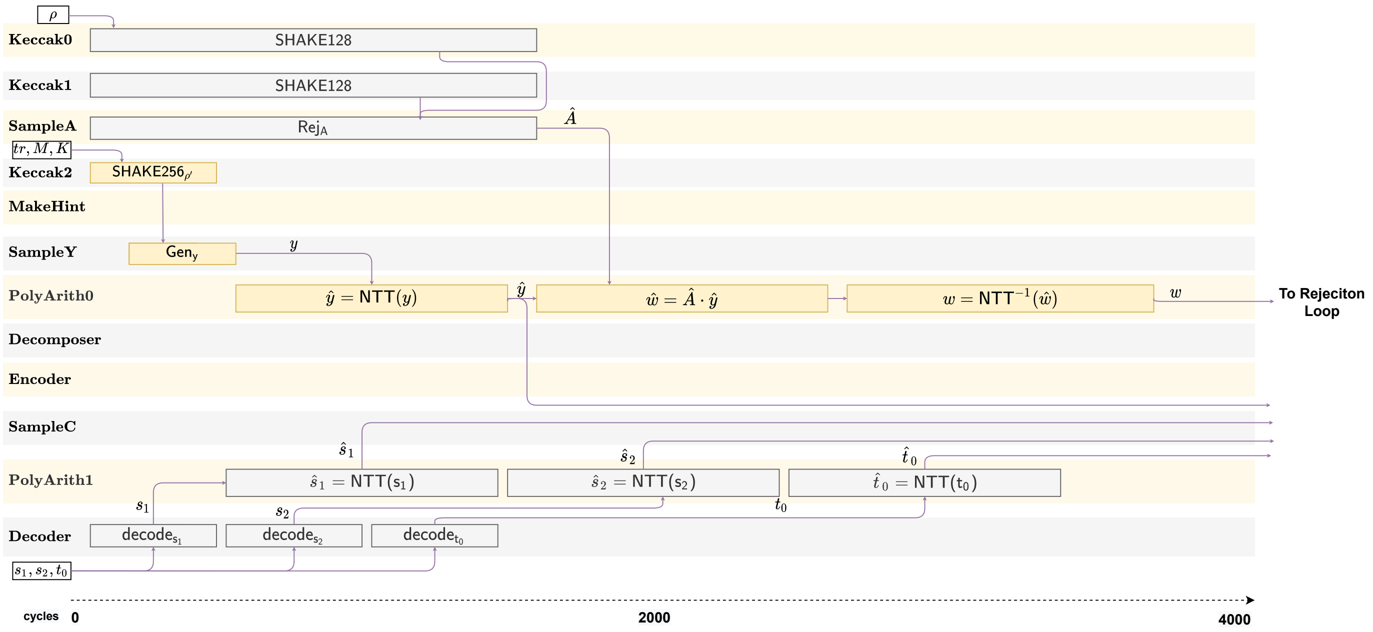

Fig. 4: Schedule of the precomputation stage of Signature Generation at security level 2

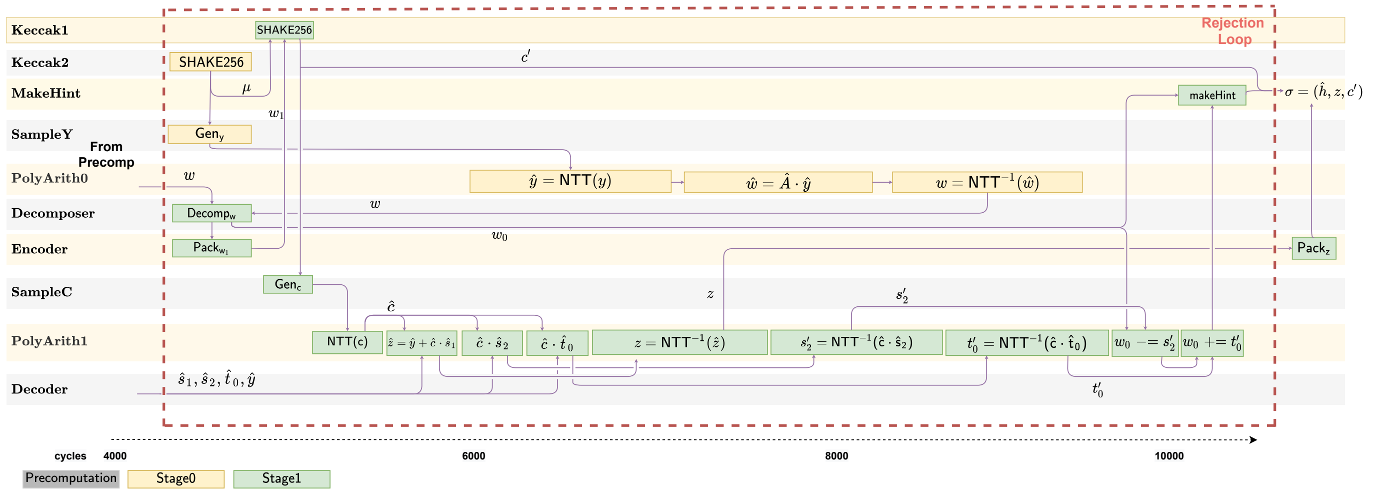

Fig. 5: Schedule of the signature rejection loop for Signature Generation at security level 2

The performance results and comparison with existing implementations is detailed in Table V. For this work (TW) and the paper by Land et al. [6], we list the best and average execution time for each of the major operation. The grouping by security level is based on the NIST-defined PQC security levels. In Dilithium, the clock cycle cost of each operation generally increases by 50% as the security level increases due to the larger vector dimensions. The one exception to this rule is the average case for signature generation, which largely depends on the average number of attempts needed to generate a valid signature. According to the specification [11], on average, level 2 requires 4.25 attempts, level 3 5.1 attempts, and level 5 3.85 attempts. In our pipelined design, each additional attempt requires 5.8K cycles for security level 2, 8.1K cycles for level 3, and 10.8K cycles for level 5.

Ricci et al. [3] report the number of clock cycles for all security levels but the maximum clock frequency and area only for security level 2. The area is reported individually for each operation, so we will compare against the area results for our individual modules. Compared to this high-performance implementation, our implementation achieves performance improvements with a lower utilization in all metrics except for BRAM in key generation and verification. In terms of latency, we achieve 1.9-3.7$\times$ improvement in clock cycles and 1.5-3.7$\times$ better results in terms of latency in microseconds. Our designs utilize 38%-46% fewer LUTs, 25%-72% fewer FFs,

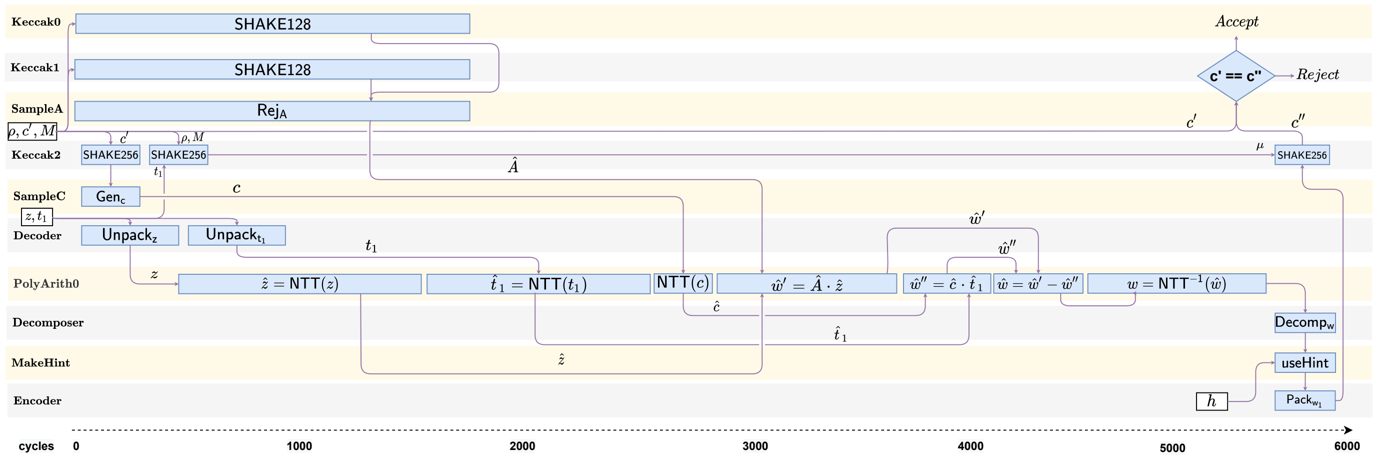

Fig. 6: Schedule of Signature Verification for security level 2

TABLE V: Full hardware implementations of digital signature schemes qualified as finalists to Round 3 of the NIST PQC standardization process. **TW** denotes This Work. Notation for FPGA families - A7: Artix-7, K7: Kintex-7, VUS+: Virtex UltraScale+

Design	Algorithm	Max Freq. (MHz)	LUT	FF	DSP	BRAM	Keygen cycles	Keygen μs	Verify cycles	Verify μs	Sign cycles	Sign μs	Family
						Security Level 1							
[8]	Picnic-L1-FS	91	90,535	23,516	0	52.5	-	-	29,600	326	31,300	344	A7
[3]	Dilithium-II[1]	-	-	-	-	-	12,600	-	10,546	-	18,338	-	VUS+
[9]	SPHINCS$^+$-128f-simple	250 & 500[3]	47,991	72,505	1	11.5	-	-	-	16	-	1,010	A7
						Security Level 2							
[6]	Dilithium-II	163	27,433	10,681	45	15	18,761	115	19,687	121	29,057/76,613	178/470	A7
[3]	Dilithium-III[1,2]	350/333/158	54,183/68,461/61,738	25,236/86,295/34,963	182/965/316	15/145/18	18,193	52	15,032	97	21,033/-	63/-	VUS+
TW	Dilithium-II	256	53,907	28,435	16	29	4,875	19	6,582	26	10,945/29,876	43/117	VUS+
						Security Level 3							
[6]	Dilithium-III	145	30,900	11,372	45	21	33,102	228	32,050	221	45,068/123,218	310/850	A7
[9]	SPHINCS$^+$-192f-simple	250 & 500[3]	48,398	73,476	1	17	-	-	-	19	-	1,170	A7
[3]	Dilithium-IV[1]	-	-	-	-	-	22,981	-	20,221	-	22,362/-	-	VUS+
TW	Dilithium-III	256	53,907	28,435	16	29	8,291	32	9,724	39	16,178/49,437	63/193	VUS+
						Security Level 5							
[6]	Dilithium-V	140	44,653	13,814	45	31	50,982	363	52,712	377	70,376/145,912	503/1,042	A7
[9]	SPHINCS$^+$-256f-simple	250 & 500[3]	51,009	74,539	1	22.5		-	-	21	-	2,520	A7
TW	Dilithium-V	116	53,187	28,318	16	29	14,037	121	14,642	126	24,358/55,070	210/475	A7
[8]	Picnic-L5-FS	125	167,530	33,164	0	99	-	-	146,600	1,173	154,500	1,236	K7
TW	Dilithium-V	173	54,468	28,639	16	29	14,037	81	13,642	85	23,358/55,070	141/318	K7
TW	Dilithium-V	256	53,907	28,435	16	29	14,037	55	13,642	57	23,358/55,070	95/215	VUS+

[1]Uses Round 2 parameter set [2]Area reported separately for Key Generation, Sign, and Verify [3] Split frequency domain: Keccak at 500 MHz, other units at 250 MHz

and 96%-98% fewer DSP. Our implementation of signature generation also uses 80% fewer BRAMs.

Our area and performance improvements are enabled by our efficient NTT design and optimized operation scheduling. The NTT design reported in [3] requires 48 DSP units for the forward NTT and 84 for the inverse NTT, while our design utilizes only 8. This allows our design to use multiple instances without drastically increasing the area. Our splitting of the rejection loop in signing also leads to a much lower area. Ricci et al. [3] duplicate modules so that there are 18 parts running in parallel, including 11 NTT instances. This requires a large amount of BRAM to buffer data between modules and leads to a much larger implementation of signature generation. One benefit of this level of parallelization and DSP usage is that it does allow a high clock frequency, however our lower clock cycle count still leads to better latency in time units.

Compared to the mid-range implementation by Land et al. [6], our design achieves substantially better performance at the cost of moderate increases in LUTs and FFs. Since this design includes results reported for modules that perform all operations at a single security level, we will compare with our combined module implemented for Artix-7. As our combined module is capable of selecting between all operations at all security levels at runtime, we will compare with with their level 5 implementation.

The design by Land et al. [6] employs some parallelization, such as the use of multiple butterfly units in their NTT, but does not use as much parallelization as our design. They also utilize different operation modes of FPGA DSPs in their design, such as pre-addition and Single Instruction Multiple Data (SIMD) addition. This allows their NTT to achieve an impressive maximum frequency of 311 MHz, but also results in very high DSP usage. However, this high frequency NTT core is not able to improve overall performance since it reside in the same clock domain as the rest of the design, forcing it to run at a lower clock rate. Our polynomial arithmetic unit is optimized so that it is not the critical path of the design, but also seeks to minimize DSP usage and cycle latency. This allows our design to achieve higher NTT performance in the application of Dilithium with a much lower DSP count. In particular, their NTT requires $533/536$ clock cycles for the forward and inverse operation while our design is able to complete the same operations in $300/294$ cycles. The additional cycles on top of the ideal of 256 cycles come from the pipeline depth. Further, their approach does not reorder the coefficients during the forward and inverse NTT. This means that they must split their coefficients across multiple true dual port BRAM to have sufficient memory bandwidth. Our reordering process means we only need a single simple dual port BRAM, which greatly simplifies the mapping of coefficients to memory leading to lower BRAM usage.

We achieve between $2.6-3.6\times$ improvement in terms of clock cycles and $2.2-3\times$ lower latency in microseconds. These improvements come at the cost of 19% more LUTs and 100.5% more FFs, but our design uses 64.6% fewer DSP units and 6.5% fewer BRAMs. Our DSP unit utilization is lower because we chose to use LUTs to implement simple arithmetic operations like addition and subtraction. Our number of LUTs also increases due to certain decisions supporting parallelization, such as the use of multiple Keccak cores. However, these design choices enable much more parallelization, which has allowed our design to obtain much lower latency with only a moderate area increase.

Software implementations of Dilithium for embedded devices are substantially slower than hardware. For example, a recent optimized implementation on Cortex-M4 takes 1.35M/3M/1.26M clock cycles for key Generation, Signing, and Verifying at security level 2 [23]. Their implementation was tested on an STM32F407 with a 24 MHz clock, so our hardware implementation has $2961\times/1071\times/2042\times$ lower latency.

In comparison with high-end CPUs, the highly optimized NEON implementation on ARMv8 Apple M1 *Firestorm* core at 3.2 GHz, reported by Becker et al. [24] for Dilithium-III takes $51/36\mu$s for Key generation and Verifying. Our hardware implementation is $1.60\times/0.92\times$ faster in two mentioned above operations. Due to rejected loop in Singing operation, it takes 149μs for average case in Apple M1, while our hardware is $0.77\times$ and $2.36\times$ faster in average and best case, respectively.

Table V also provides some basic insights into the comparison between Dilithium and other digital signature schemes in terms of performance in hardware. In particular, the Picnic implementation at security level 5 [8] has substantially lower performance than the Dilithium algorithms. While it consumes no DSP units, it requires approximately $3\times$ more LUTs and BRAMs.

Multiple SPHINCS$^+$ implementations for different parameter sets are reported in [9]. We selected the parameters with the best performance to compare to our design since we target high performance. While this implementation uses more FFs than our design, its LUTs utilization is comparable, and its use of BRAMs and DSPs is substantially smaller. The verification algorithm is also very efficient. However, signature generation is substantially slower than Dilithium which may cause issues in certain applications.

These results suggest that the lattice-based Dilithium cryptosystem is more efficient in hardware than Picnic and SPHINCS+. However, more investigation is required for a definitive ranking.

VII. Conclusions

This paper presents a high-performance implementation of CRYSTALS-Dilithium, which achieves the best-known latency and a smaller area than the best previously reported high-performance design. The implementation includes both a combined module capable of performing three major operations at all security levels and individual modules supporting one operation each. In all modules, each of the three NIST security levels can be selected at run-time. The rejection loop latency in the signature generation is optimized by splitting it into a two-stage pipeline, which minimizes the average-case signature generation time.

References

[1] P. Shor, “Algorithms for quantum computation: Discrete logarithms and factoring,” in *Proceedings 35th Annual Symposium on Foundations of Computer Science*, Santa Fe, NM, USA: IEEE Comput. Soc. Press, 1994, pp. 124–134.

[2] L. Chen, S. Jordan, Y.-K. Liu, D. Moody, R. Peralta, R. Perlner, and D. Smith-Tone, “Report on Post-Quantum Cryptography,” National Institute of Standards and Technology, Tech. Rep. NIST IR 8105, Apr. 2016, NIST IR 8105.

[3] S. Ricci, L. Malina, P. Jedlicka, D. Smekal, J. Hajny, P. Cibik, and P. Dobias, “Implementing CRYSTALS-Dilithium Signature Scheme on FPGAs,” Cryptology ePrint Archive 2021/108, Jan. 2021.

[4] U. Banerjee, T. S. Ukyab, and A. P. Chandrakasan, “Sapphire: A Configurable Crypto-Processor for Post-Quantum Lattice-based Protocols,” *IACR Transactions on Cryptographic Hardware and Embedded Systems*, vol. 2019, no. 4, Aug. 2019.

[5] ——, “Sapphire: A Configurable Crypto-Processor for Post-Quantum Lattice-based Protocols (Extended Version),” Cryptology ePrint Archive 2019/1140, Sep. 2020.

[6] G. Land, P. Sasdrich, and T. Guneysu, “A Hard Crystal - Implementing Dilithium on Reconfigurable Hardware,” Cryptology ePrint Archive 2021/355, Mar. 2021.

[7] A. Ferozpuri and K. Gaj, “High-speed FPGA Implementation of the NIST Round 1 Rainbow Signature Scheme,” in *2018 International Conference on ReConFigurable Computing and FPGAs (ReConFig)*, Cancun, Mexico: IEEE, Dec. 2018, pp. 1–8.

[8] D. Kales, S. Ramacher, C. Rechberger, R. Walch, and M. Werner, "Efficient FPGA Implementations of LowMC and Picnic," in *The Cryptographers' Track at the RSA Conference 2020, CT-RSA 2020*, San Francisco: Springer, Feb. 2020.

[9] D. Amiet, L. Leuenberger, A. Curiger, and P. Zbinden, "FPGA-based SPHINCS+ Implementations: Mind the Glitch," en, in *2020 23rd Euromicro Conference on Digital System Design (DSD)*, Kranj, Slovenia: IEEE, Aug. 2020, pp. 229–237.

[10] L. Ducas, E. Kiltz, T. Lepoint, V. Lyubashevsky, P. Schwabe, G. Seiler, and D. Stehlé, "CRYSTALS-Dilithium: Algorithm Specifications and Supporting Documentation," NIST Round 2, Mar. 2019.

[11] S. Bai, L. Ducas, E. Kiltz, T. Lepoint, V. Lyubashevsky, P. Schwabe, G. Seiler, and D. Stehlé, "CRYSTALS-Dilithium: Algorithm Specifications and Supporting Documentation (Version 3.1)," Tech. Rep., Feb. 2021.

[12] Ö. Dagdelen, M. Fischlin, and T. Gagliardoni, "The Fiat–Shamir Transformation in a Quantum World," in *Advances in Cryptology - ASIACRYPT 2013*, ser. LNCS, vol. 8270, Berlin, Heidelberg: Springer Berlin Heidelberg, 2013, pp. 62–81.

[13] V. Lyubashevsky, "Fiat-Shamir with Aborts: Applications to Lattice and Factoring-Based Signatures," in *Advances in Cryptology – ASIACRYPT 2009*, vol. 5912, Berlin, Heidelberg: Springer Berlin Heidelberg, 2009, pp. 598–616.

[14] L. Ducas, T. Lepoint, V. Lyubashevsky, P. Schwabe, G. Seiler, and D. Stehlé, "CRYSTALS – Dilithium: Digital Signatures from Module Lattices," Cryptology ePrint Archive 2017/633, 2017, p. 17.

[15] National Institute of Standards and Technology, *FIPS PUB 202: SHA-3 Standard: Permutation-Based Hash and Extendable-Output Functions*. Aug. 2015.

[16] L. Beckwith and W. Diehl, "New Directions for NewHope: Improving Performance of Post-Quantum Cryptography through Algorithm-level Pipelining," in *2020 International Conference on Field-Programmable Technology (ICFPT)*, Maui, HI, USA: IEEE, Dec. 2020, pp. 120–128.

[17] D. T. Nguyen, V. B. Dang, and K. Gaj, "A High-Level Synthesis Approach to the Software/Hardware Codesign of NTT-Based Post-Quantum Cryptography Algorithms," in *2019 International Conference on Field-Programmable Technology (ICFPT)*, Tianjin, China: IEEE, Dec. 2019, pp. 371–374.

[18] ——, "High-Level Synthesis in Implementing and Benchmarking Number Theoretic Transform in Lattice-based Post-Quantum Cryptography using Software/Hardware Codesign," in *16th International Symposium on Applied Reconfigurable Computing, ARC 2020*, Apr. 2020.

[19] N. Zhang, B. Yang, C. Chen, S. Yin, S. Wei, and L. Liu, "Highly Efficient Architecture of NewHope-NIST on FPGA using Low-Complexity NTT/INTT," *IACR Transactions on Cryptographic Hardware and Embedded Systems*, pp. 49–72, Mar. 2020.

[20] "7 Series FPGAs Memory Resources User Guide," 2019.

[21] CERG, *SHAKE*, https://github.com/GMUCERG/SHAKE, 2021.

[22] F. Farahmand, A. Ferozpuri, W. Diehl, and K. Gaj, "Minerva: Automated hardware optimization tool," in *2017 International Conference on ReConFigurable Computing and FPGAs, ReConFig 2017*, Cancun: IEEE, Dec. 2017, pp. 1–8.

[23] D. O. C. Greconici, M. J. Kannwischer, and D. Sprenkels, "Compact Dilithium Implementations on Cortex-M3 and Cortex-M4," en, *IACR Transactions on Cryptographic Hardware and Embedded Systems*, pp. 1–24, Dec. 2020. [Online]. Available: https://tches.iacr.org/index.php/TCHES/article/view/8725 (visited on 08/01/2021).

[24] H. Becker, V. Hwang, M. J. Kannwischer, B.-Y. Yang, and S.-Y. Yang, "Neon NTT: Faster Dilithium, Kyber, and Saber on Cortex-A72 and Apple M1," Cryptology ePrint Archive 2021/986, Jul. 2021.