






A Hard Crystal - Implementing Dilithium on Reconfigurable Hardware

Georg Land^{1,2} , Pascal Sasdrich¹ , and Tim Güneysu^{1,2} 

¹ Ruhr University Bochum, Horst Görtz Institute for IT Security, Bochum, Germany
{georg.land,pascal.sasdrich,tim.gueysu}@rub.de

² DFKI GmbH, Cyber-Physical Systems, Bremen, Germany

Abstract. CRYSTALS-Dilithium as a lattice-based digital signature scheme has been selected as a finalist in the Post-Quantum Cryptography (PQC) standardization process of NIST. As part of this selection, a variety of software implementations have been evaluated regarding their performance and memory requirements for platforms like x86 or ARM Cortex-M4. In this work, we present a first set of Field-Programmable Gate Array (FPGA) implementations for the low-end Xilinx Artix-7 platform, evaluating the peculiarities of the scheme in hardware, reflecting all available round-3 parameter sets. As a key component in our analysis, we present results for a specifically adapted Number-Theoretic Transform (NTT) core for the Dilithium cryptosystem, optimizing this component for an optimal Look-Up Table (LUT) and Flip-Flop (FF) utilization by efficient use of special purpose Digital Signal Processors (DSPs). Presenting our results, we aim to shed further light on the performance of lattice-based cryptography in low-cost and high-throughput configurations and their respective potential use-cases in practice.

Keywords: FPGA · Dilithium · PQC

1 Introduction

In the light of continuous progress and advancement on the development of quantum computers, security of existing public-key cryptographic schemes starts to crumble [12]. While most existing and currently deployed schemes rely on the hardness of *integer factorization* or computing *discrete logarithms*, broken by Shor's quantum algorithm [15], given that an attacker has access to a large-scale quantum computer, a call for the design, proposal, and standardization of new post-quantum secure schemes for Key Encapsulation Mechanism (KEM) and digital signatures has been initiated by the United States National Institute for Standards and Technology (NIST) in 2017 [10].

After two competitive rounds of thorough scrutiny and examination, NIST announced the seven finalists from the initial field of 69 candidates in 2020 which still have to undergo further evaluation in a third and final round. Moreover, the

seven finalists can be categorized into the four key establishment schemes, Classic McEliece, Kyber, NTRU, and Saber as well as the three digital signature schemes Dilithium, Falcon, and Rainbow.

Interestingly, five out of the seven remaining finalists are using hard lattice problems as fundamental security assumption. Along with Falcon [7], Dilithium [5] is one of the two remaining lattice-based digital signature schemes, while Rainbow is based on multivariate cryptography instead. Further, Dilithium and Kyber are part of the Cryptographic Suite for Algebraic Lattices (CRYSTALS) using structured lattices to allow fast arithmetic and enable compact key, ciphertext and signature sizes. More precisely, the underlying polynomial ring enables efficient polynomial multiplication leveraging the Number-Theoretic Transform (NTT).

While literature is rich in efficient and optimized implementations on lattice-based KEMs, to date, lattice-based digital signature schemes are mostly neglected. In particular, efficient implementation of lattice-based signature schemes in reconfigurable hardware urgently needs to be investigated in order to guide and support the selection of the future post-quantum cryptography standards. In this regard, we are only aware of a two existing hardware implementations of Dilithium [13, 16], while several optimized software implementations, e.g., targeting AVX2 [4] or Cortex-M4 [8] architectures, have been presented recently. Further, even though the design in [13] has been implemented on a high-performance Virtex-7 Field-Programmable Gate Array (FPGA), it does not exploit important features of modern reconfigurable hardware architectures efficiently. For this, we present a novel set of efficient and compact FPGA implementations specifically targeting a low-end Xilinx Artix-7 series through evaluating the peculiarities of the Dilithium digital signature scheme for efficient and clever mapping into modern FPGA features and components¹.

Contribution. For this, our contribution can be summarized as follows:

- An optimized NTT component making extensive use of Digital Signal Processors (DSPs) is presented to exploit peculiarities and features of modern low-end FPGAs. We were able to synthesize our NTT implementation for a frequency of 311 MHz, resulting in a latency of 1.7 μ s, which is, to the best of our knowledge, the fastest NTT implementation for comparable parameters in Artix-7 FPGAs.
- Our Dilithium core is compact and self-contained, providing functionalities for key generation, signature generation, signature verification, precomputation, arbitrary-length message digesting, and packing and unpacking keys and signatures.
- For Dilithium-III, our core uses 30k Look-Up Tables (LUTs), 11k Flip-Flops (FFs), 45 DSPs and 23 Block-RAMs (BRAMs) with $f_{max} = 142$ MHz. For key generation, our core is capable of performing 4290 OP/s, for signature generation 1351 OP/s and for signature verification 11751 OP/s.

¹ Our implementation is publicly available at <https://github.com/Chair-for-Security-Engineering/dilithium-artix7>.

- Additionally, we report area and speed results for individual stand-alone cores, supporting either only key generation, signature generation, or verification. These smaller cores still support the necessary unpacking, packing, digesting, and precomputation operations.
- For Dilithium-III, our keygen-only core is capable of performing 7250 OP/s. The sign-only core performs 1560 OP/s and the verify-only core 16137 OP/s.

Related Work. Many lattice-based schemes have been proposed in recent years and there is a wide variety of implementations in hardware. The first implementation of a lattice-based signature scheme was proposed by Güneysu et al. [9] in 2012. Pöppelmann et al. extend this work in [11]. Soni et al. present an implementation of the second-round parameter set of Dilithium targeting Artix-7 FPGAs [16]. However, they use a High Level Synthesis (HLS) approach resulting in a rather large design. Another implementation of the second-round parameter set of Dilithium is provided by Ricci et al., which targets the high-end Virtex-7 platform [13]. Most notably, their design achieves a high throughput for signature generation. Other post-quantum secure signature schemes that have been implemented in reconfigurable hardware include Rainbow [6], SPHINCS [1] and XMSS [17]. Furthermore, efficient implementation of the NTT in hardware has been researched very well. Roy et al. presented an efficient design that uses two merged NTT layers [14]. Banerjee et al. presented an Application-Specific Integrated Circuit (ASIC) design of the NTT that can be used to accelerate multiple schemes [2]. Finally, Zhang et al. present a way to integrate the post-processing of the inverse transformation into the main computation resulting in a low-complexity implementation [19].

2 Preliminaries

2.1 Notation

Throughout this work, we will use and assume the following notation. Let n and q be two integers, such that $n = 256$ and $q = 2^{23} - 2^{13} + 1$. Further, let \mathcal{R}_q be a polynomial ring with $\mathcal{R}_q = \mathbb{Z}_q[X]/(X^n + 1)$. In addition, let us denote vectors in bold lower-case letters, e.g., \mathbf{v} , while matrices are denoted in bold upper-case letters, e.g., \mathbf{A} . Polynomials in NTT domain are indicated by a hat.

Additionally, for an integer s , we denote $s[a : b]$, where $a > b$, as the bit slice of s bounded by the offsets a, b counting from LSB to MSB, for example for $s = 6$ we have $s[2 : 1] = 11_2 = 3$.

2.2 Number-Theoretic Transform

The NTT, as used in Dilithium, can be seen as a discrete Fourier transform over polynomials in \mathcal{R}_q , where the complex arithmetic is replaced by the modular arithmetic of the polynomial coefficients. Since the ring structure enables negative wrapped convolution, we can use an n -point NTT for fast polynomial

multiplication by transforming both factor polynomials to the NTT domain, multiplying coefficient-wise in NTT domain, and then applying the inverse transform to the result to obtain the final product polynomial.

2.3 CRYSTALS-Dilithium

In July 2020, NIST announced the 7 finalist and 8 alternate candidates for the Post-Quantum Cryptography (PQC) standardization competition, with both schemes of the CRYSTALS suite being selected as finalist for their respective categories. In particular the digital signature scheme Dilithium has undergone a thorough scrutiny during the competition process and since then reached version 3.1 [5], while most recently some major changes and updates for the various security parameter sets have been presented.

In general, the Dilithium digital signature scheme has been designed to adopt simple and secure design principles, in particular substituting discrete Gaussian sampling in favor of uniform sampling. In addition, all remaining fundamental operations have been carefully chosen such that they easily can be performed in constant time. Aiming at long-term security, the different security levels and parameters have been chosen conservatively while endeavoring to minimize the combined size of public key and signatures. Eventually, the modular construction of Dilithium favors efficient and highly optimized implementations across all security levels and parameter sets as the main operations rely on SHAKE-128 or SHAKE-256 and the multiplication in the polynomial ring \mathcal{R}_q , regardless of the security level. Instead, higher or lower security is only achieved through addition or reduction in the number of operations performed in \mathcal{R}_q .

Further, as a digital signature scheme, Dilithium provides the following three core methods for *key generation*, *signature generation*, and *signature verification*.

Key Generation. For key generation, the respective algorithm generates a $k \times l$ matrix \mathbf{A} such that each entry in the matrix is a polynomial of the ring \mathcal{R}_q . Using randomly sampled vectors \mathbf{s}_1 and \mathbf{s}_2 , with polynomials in \mathcal{R}_q where each coefficient is in $[-\eta, \eta]$, the second part of the public key is generated as $\mathbf{t} = \mathbf{A}\mathbf{s}_1 + \mathbf{s}_2$, performing all algebraic operations over \mathcal{R}_q . To keep the public key size small, the matrix \mathbf{A} is replaced by a seed ρ which generates \mathbf{A} deterministically, which is a widespread technique in lattice-based cryptography. Additionally, to further decrease the size of the public key, the lower d bits of each coefficient in \mathbf{t} are placed in the secret key rather than the public key.

Signature Generation. The fundamental operation of Dilithium is the generation of digital signatures. For this, the signing algorithm chooses a masking vector \mathbf{y} with coefficients from $[-\gamma_1, \gamma_1)$ in order to compute $\mathbf{w} = \mathbf{A}\mathbf{y}$ and rounds the result such that $\mathbf{w} = \mathbf{w}_1 \cdot 2\gamma_2 + \mathbf{w}_0$, where each coefficient in \mathbf{w}_0 is less than or equal to γ_2 . The challenge c , a polynomial in \mathcal{R}_q with coefficients from $\{-1, 1\}$ at τ random positions and all other coefficients being 0, is sampled by hashing the message and \mathbf{w}_1 and is used to generate the potential signature $\mathbf{z} = \mathbf{y} + c\mathbf{s}_1$.

Using rejection sampling, leakage of the secret key is prevented, at the penalty of repeating the signature generation process if the signature fails the security and correctness checks. Additionally, since for the verification \mathbf{t} is needed but only the upper 10 bits of \mathbf{t} are contained in the public key, the signer needs to compute the vector of carry bits (“hints”) \mathbf{h} that result from the unknown part in \mathbf{t} during the verification computation. Finally, if a \mathbf{z} is found that passes the checks, the signature is returned as $(c, \mathbf{z}, \mathbf{h})$.

Signature Verification. For signature verification, $\mathbf{A}\mathbf{z} - c\mathbf{t}$ is rounded analogously to the signing procedure and the resulting *higher-order* bits are set to be \mathbf{w}'_1 . Since the lower bits of each coefficient in \mathbf{t} are not contained in the public key, the verifier makes use of the hints \mathbf{h} to perform this operation. Following this, the challenge c is recomputed from the message and \mathbf{w}'_1 and compared to the one provided in the signature. Also, \mathbf{z} is checked to have a valid norm (i.e., whether each coefficient has the maximum value as checked during signature generation).

Parameter Sets. With introduction of version 3.1 of the Dilithium algorithm specification, the list of supported security parameter sets has been adjusted for the three NIST security levels II, III, and V. Since the operations in \mathcal{R}_q do not change for the different parameter sets, the performance-critical dimensions of \mathbf{A} are adjusted, resulting in an increased or reduced number of operations, depending on the targeted security level.

Compared to round 2, the following adjustments have been proposed:

- d is decreased from 14 to 13.
- τ is now different for each parameter set rather than 60 for all, resulting in a slight speed-up for the lower parameter sets.
- γ_1 is now a power of two, which simplifies sampling \mathbf{y} significantly.
- $\eta = 2$ for security levels II and V and $\eta = 4$ for security level III, rather than different η s for each parameter set.
- $\gamma_2 = (q - 1)/88$ for security level II. Both other security levels keep $\gamma_2 = (q - 1)/32$.

3 Design Considerations

Modern FPGA generations are equipped with a multitude of general purpose logic. However, for certain applications, highly optimized special purpose components such as very compact and optimized DSP cores are provided, offering efficient and fast integer arithmetic operations, or BRAMs, offering compact true dual-port memory banks for easy storage of larger amounts of data. Given this, our primary design goal was to reduce the footprint of our architecture in terms of general purpose components such as LUTs and FFs, as these components usually are the limiting factor in larger systems. Additionally, we design all operations such that there is no timing dependency on secret values.

3.1 Arithmetic

As a first step, we opted to implement the basic arithmetic using DSP modules for fast and efficient coefficient-level computations. DSP blocks are abundantly available on latest FPGA devices but in general applications rarely used. More precisely, we exploit several special features of modern Xilinx DSP blocks, including:

Runtime Reconfiguration. During design and synthesis time, the DSP can be configured to provide different functionalities during runtime. Based on this, we configured some of our instantiated DSP modules to provide multiple different arithmetic operations, allowing to re-use the same DSP for different operations, hence resulting in a highly integrated and optimized design with respect to area and utilization.

Pre-addition. Besides fast integer multiplication, each DSP unit is equipped with a pre-adder stage, allowing to merge multiple arithmetic operations within a single DSP.

Single Instruction Multiple Data. Although each DSP unit can perform up to 48-bit wide additions, we opted to use DSP cores in a Single Instruction Multiple Data (SIMD) fashion, allowing to perform two 24-bit additions or subtractions instead, perfectly fitting the constraints of underlying arithmetic operations in the polynomial ring.

Number-Theoretic Transform. On a high level, we follow the design ideas from [19], especially including the inverse NTT without post-processing. However, by applying the aforementioned DSP features to our NTT design, our implementation achieves a low latency despite processing relatively big coefficients. In contrast to known NTT architectures, which usually utilize DSPs only for a low-latency multiplication and perform any other arithmetic with general-purpose logic, our novel approach of leveraging the full capabilities of DSPs results in a low latency for any involved arithmetic. This approach fits the requirements for implementing Butterfly Units (BFUs) particularly well, as during the forward NTT, e.g., $a + bw$ is computed, which can be mapped to DSP functionality *without* additional arithmetic logic. Also, even though this operation is not useful for our inverse NTT, we still can re-use the exact same DSPs by reconfiguring them at runtime at the cost of additional control logic.

3.2 Memory

Besides efficient arithmetic, a specific memory architecture and layout is required to store and load coefficients and polynomials efficiently during arithmetic operations. Given the design considerations for our arithmetic modules including the NTT unit, we identified the following two design constraints for our memory architecture:

1. Given the NTT architecture, the design would benefit from reading and writing up to four coefficients simultaneously. For this, we decided to use four simple dual port BRAMs to store polynomials. More precisely, we use four parallel 18K BRAM instances for this, each of them holding up to 512 coefficients. This means, since for a single polynomial only 64 coefficients are stored per BRAM, we can fill the four 18K BRAM units with up to eight full polynomials.
2. The memory layout has to be adjusted such that the number of read and write conflicts are minimized. In particular, the layout has to ensure that the coefficients of the polynomials are distributed among the BRAMs such that we always can read or write data during the arithmetic operations without stalling due to memory access conflicts. This can be achieved as follows: For a polynomial's coefficient aX^i , the coefficient is placed in memory *bankaddr* (Eq. 1) at address *addr* (Eq. 2) [19, Sec. 3.1].

$$\text{bankaddr} = i[7 : 6] + i[5 : 4] + i[3 : 2] + i[1 : 0] \bmod 4 \quad (1)$$

$$\text{addr} = i[7 : 2] \quad (2)$$

Our design needs to hold $k \cdot l + 2l + 6k + 1$ polynomials in total. It is possible to reduce this memory footprint significantly by sampling single polynomials of $\hat{\mathbf{A}}$ just in time. However, we opt to expand $\hat{\mathbf{A}}$ once and store it for further computations. This has the advantage that introducing a pre-processing operation enables signing multiple messages (or verifying multiple signatures) under the same key without the necessity of re-sampling $\hat{\mathbf{A}}$.

Since \mathbf{z} and \mathbf{y} are never accessed simultaneously, we only plan with l polynomials for both together. Additionally, \mathbf{s}_1 takes storage for l polynomials. c occupies storage for one polynomial. Four of the six polynomial vectors of size k are $\mathbf{s}_2, \mathbf{t}_0, \mathbf{t}_1, \mathbf{w}$. The remaining $2k$ polynomials are used as temporary storage, for example during *MakeHint*.

Given that we can store up to 8 polynomials using four BRAM units, the total number of BRAM instances is governed by the security level. In particular, we need storage for 49 polynomials for level II, 77 polynomials for level III, and 119 polynomials for level V. We were able to identify efficient memory mappings for each parameter set, such that it only requires $\lceil 4(kl + 2l + 6k + 1)/8 \rceil$ 18K BRAM primitives. We did so by iteratively searching through possible memory mappings in a randomized way and checking whether the requirements are met. The memory mapping enables the following operations in a pipelined or parallel fashion:

- During matrix-vector multiplication, the vector elements are transformed sequentially to NTT domain. Upon completion of the transformation, the multiply-accumulate module updates the resulting vector elements through coefficient-wise multiplication with the $\hat{\mathbf{A}}$ polynomials.

- In pre-computations for signature generation and verification, the matrix $\hat{\mathbf{A}}$ is expanded and in parallel, NTTs of \mathbf{s}_1 , \mathbf{s}_2 , \mathbf{t}_0 and \mathbf{t}_1 can be performed.
- During verification, the norm check of \mathbf{z} can be performed in parallel to sampling c .
- Since, at the end of key generation, \mathbf{s}_1 is part of the secret key, during matrix-vector multiplication, \mathbf{s}_1 is transformed to NTT domain for fast multiplication. To avoid the necessity of performing an inverse NTT, \mathbf{s}_1 is stored in two locations simultaneously during sampling, after which one can be transformed and the other location is used as result.

3.3 Functionality

In order to provide an integrated and self-contained core for generation and verification of digital signatures based on the Dilithium scheme, our architecture needs to support the full set of the following operations:

KeyGen Generation of a key from a given seed.

Sign_{pre} Expansion of $\hat{\mathbf{A}}$ and pre-computation of $\hat{\mathbf{s}}_1$, $\hat{\mathbf{s}}_2$, and $\hat{\mathbf{t}}_0$.

Sign Signature computation.

Verify_{pre} Expansion of $\hat{\mathbf{A}}$ and pre-computation of $\hat{\mathbf{t}}_1$.

Verify Signature verification.

Digest_{msg} Hashing of arbitrary-length messages along with tr (of the public key).

Store Storing and unpacking public keys, secret keys, signatures, or seeds.

Load Packing and sending public keys, secret keys, or signatures.

Additionally, we provide individual cores which only support either key generation, signature generation, or verification. Besides featuring only a subset of the operations, these smaller cores also come with a lower BRAM usage since some polynomials are only required for a subset of operations. An overview which operation is supported by each single-task core can be found in Table 1.

Table 1. Operation support matrix for single-task cores

	KeyGen	Sign _{pre}	Sign	Verify _{pre}	Verify	Digest _{msg}	Store	Load
KeyGen-only	✓	✗	✗	✗	✗	✗	Seed	Keys
Sign-only	✗	✓	✓	✗	✗	✓	k_{priv}	Sign.
Verify-only	✗	✗	✗	✓	✓	✓	k_{pub}	

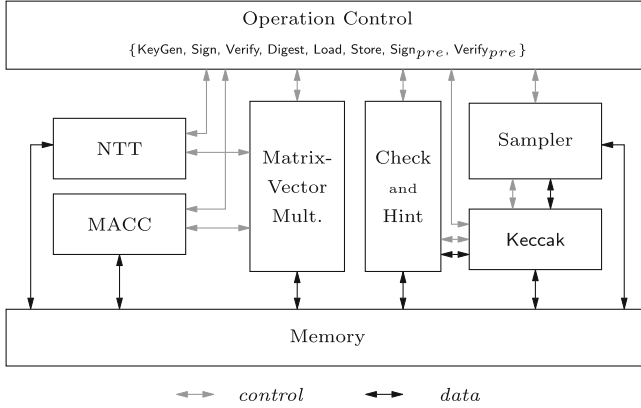


Fig. 1. Dilithium high-level architecture

4 Implementation on Reconfigurable Hardware

In this section, we outline the basic architecture of our comprehensive Dilithium architecture. In particular, our construction exploits special purpose units and features of an Artix-7 FPGA (XC7A100T).

4.1 Architectural Details

The high-level architecture of our implementation is shown in Fig. 1. All basic arithmetic operations are performed by the *NTT*, *Multiply-Accumulate (MACC)*, and *Matrix-Vector Multiplication* units. However, even though the matrix-vector multiplication serves as master and control unit for the *NTT* and *MACC* cores, both sub-cores must be accessible from the global operation control unit as well to provide auxiliary support for additional arithmetic operations. Besides, the check units directly access polynomials in the memory for norm checking and provide the check result to the operation control module. The *Sampler* module controls and accesses the *Keccak* hash core in order to buffer the hash output before writing the uniformly generated random samples to memory. However, the *Keccak*-based hash core is also accessible from the operation control unit, mostly required for random seed expansion. Finally, the *hint* modules control read and write access to the hint registers in the memory unit. Further, as already highlighted in Sect. 3, the memory unit consists of several BRAMs for the intermediate polynomials, two 512-bit registers to store ρ' and μ as well as some additional 256-bit registers for ρ , \tilde{c} , tr , K , and the seed for the key generation.

Number-Theoretic Transform. As already mentioned in Sect. 3, our *NTT* implementation follows the design principles of [19]. However, we pre-multiply the stored twiddle factors for the inverse transform by a factor of 2^{-1} in order to avoid the additional logic for multiplying one coefficient by 2^{-1} in the BFU.

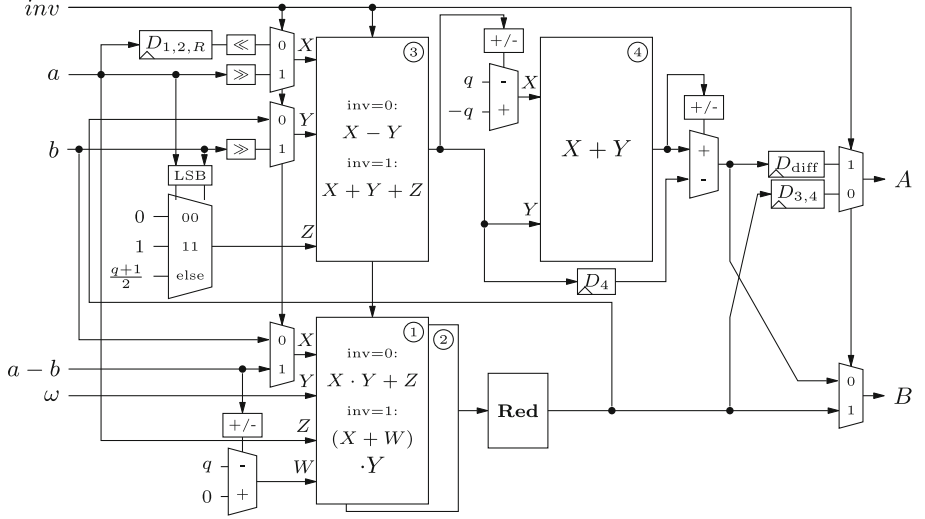


Fig. 2. Architecture of the BFU. DSPs are numbered, D_i are shift registers that compensate for the DSPs or the reduction as given in their respective index. D_{diff} compensates for the difference of cycle counts between $D_{1,2,R}$ and $D_{3,4}$

From an implementation perspective, we modify several details: First, as already mentioned we utilize DSPs for all arithmetic operations in order to achieve a low area footprint and a high frequency. Second, we make use of the *true* dual-port capabilities of the BRAM modules, enabling our design to read two twiddle factors simultaneously in the lowest NTT layer and thus still allowing processing four coefficients at the same time.

At the core of the NTT, we instantiate two independent BFUs, as depicted in Fig. 2. More precisely, each BFU receives two unsigned 23-bit coefficients, an unsigned 23-bit twiddle factor, and the *signed* 24-bit value $a - b$. Note, however, that this value can be computed for both BFUs simultaneously using a single DSP in SIMD mode.

Forward Number-Theoretic Transform. In general, the butterfly configuration for the forward NTT computes two values A and B such that $(A, B) := (a + b \cdot \omega, a - b \cdot \omega)$, given that ω denotes the pre-computed twiddle factor. For this, we use the DSPs 1 and 2 to compute $a + b\omega$. More precisely, we need to combine two DSPs for this operation since each DSP itself can only perform signed 25×18 -bit multiplications. However, when combining DSPs for larger multiplications, we can leverage a dedicated low-latency cascade path. After multiplication, the resulting product is reduced to a representative in $[0, q)$ and already provides the first part of the forward NTT computation. Further, subtracting the first part from $2a$ and adding or subtracting q (depending on the sign of the subtraction result), we obtain the second part of the forward NTT output.

In addition, for increased throughput, the BFUs have been pipelined, using shift register instances to delay the input a of the third DSP. More specifically, the first part of the result is also delayed through a shift register in order to return both parts of the forward NTT computation simultaneously.

Inverse Number-Theoretic Transform. Similar to the forward NTT, the inverse NTT computes two values A and B , such that $(A, B) := (2^{-1}(a + b), (a - b)\omega)$. However, as already mentioned before, this time the operand ω for the inverse NTT is already pre-processed to incorporate the factor 2^{-1} .

Here, ω and the pre-computed, signed value $a - b$ are used as input for the multiplication DSPs 1 and 2. Further, depending on the sign bit of the value $a - b$, we choose between adding q or 0 using the pre-adder stage of the multiplication DSPs to obtain a positive multiplication result. Finally, the multiplication result is then reduced and serves as output. Besides, the second part of the output is designed to be $2^{-1}(a + b)$. For this, we use DSP 3 as 3-input adder with inputs $\lfloor a/2 \rfloor$, $\lfloor b/2 \rfloor$ and either 1 (if both least signification bits (LSBs) of a and b are 1), or $(q + 1)/2$ (if the LSB of either a or b is 1), or 0 otherwise. Since the result of this operation might be greater or equal to q , we use the fourth DSP to subtract q from the result. The second part of the BFU output is then chosen between the output of DSPs 3 and 4.

Multiply-Accumulate. The second arithmetic core is used to perform *multiply-accumulate* operations. More specifically, this core is designed to perform four computations per clock cycle in parallel in order to make full use of the available memory bandwidth. It consists of eight DSPs and four reduction modules. Each two DSPs perform one of the following operation, while the result then is fed into the reduction module.

- $a \cdot b + c$: The first DSP performs the multiplication of a with the lower 17 bits of b and the addition. The second DSP multiplies a with the remaining upper bits of b and updates the first result to the final 46 bit value that is then fed into the reduction module.
- $a + b$: The first DSP computes the sum, while the second one subtracts q . Eventually, the result of the second DSP is selected if it is non-negative, else the result of the first DSP is selected.
- $b - a$: The first DSP computes the subtraction, while the second one adds q . Eventually, the result of the first DSP is selected as output if it is positive, else the output of the second DSP is selected.

Note that for operations without multiplication, the reduction module can be bypassed, resulting in a lower latency. Again, this module is fully pipelined, allowing to process an entire polynomial within 64 cycles (in addition to the initial pipeline length).

Matrix-Vector Multiplication. This module controls both the NTT module and the MACC module to (1) transform the polynomials in the input vector

into NTT domain and (2) perform a matrix-vector multiplication with $\hat{\mathbf{A}}$. The resulting polynomial vector is then in NTT representation as well.

First, the first input polynomial is transformed by the NTT module. Then, while the second input polynomial is transformed, the point-wise multiplication between each polynomial from the first column in $\hat{\mathbf{A}}$ and the first, already transformed input polynomial is carried out consecutively using the MACC module. The resulting polynomials are stored in the result vector polynomial storage. Note that the point-wise multiplications take $k \cdot 64 + 14$ cycles², while one NTT takes 533 cycles. When both operations are finished, the NTT module transforms the third input polynomial and in parallel, the second, already transformed input polynomial is multiplied point-wise with each polynomial from the second column in $\hat{\mathbf{A}}$ and added to the intermediate result from the first k MACC operations. The resulting polynomials again are stored back to the result vector polynomial storage.

This procedure is repeated until all l input polynomials are transformed. Afterwards, the resulting k polynomials are updated to the final result using the MACC module. With this, a whole matrix-vector multiplication is carried out in $k \cdot 512 + 23 + k \cdot 64 + 14$ cycles³.

Modular Reduction. In our implementation, we need a total of six reduction module instantiations: While each BFU module contains a single reduction module, the MACC module contains four reduction modules. For the modular reduction of a 46-bit value s , we recursively exploit the relation $2^{23} \equiv 2^{13} - 1 \pmod{q}$ in a similar way as in [19].

$$\begin{aligned}
s[45 : 0] &\equiv 2^{23}s[45 : 23] + s[22 : 0] \equiv 2^{13}s[45 : 23] - s[45 : 23] + s[22 : 0] \\
&\equiv 2^{23}s[45 : 33] + 2^{13}s[32 : 23] - s[45 : 23] + z \\
&\equiv 2^{13}(s[45 : 33] + s[32 : 23]) - (s[45 : 33] + s[45 : 23]) + z \\
&\equiv 2^{23}s[45 : 43] + 2^{13}(s[42 : 33] + s[32 : 23]) - (s[45 : 33] + s[45 : 23]) + z \\
&\equiv 2^{13}(s[45 : 43] + s[42 : 33] + s[32 : 23]) - (s[45 : 43] + s[45 : 33] + s[45 : 23]) + z \\
&\equiv 2^{13}x - y + z \equiv 2^{23}x[11 : 10] + 2^{13}x[9 : 0] - y + z \\
&\equiv 2^{13}(x[11 : 10] + x[9 : 0]) - (y + x[11 : 10]) + z \pmod{q}
\end{aligned}$$

The result of our reduction can still be greater than 2^{23} so that we could repeat the substitution once again at the expense of additional depth and delay in the arithmetic computation. However, we observe that the result of the reduction at this point is already within the interval $(-\mathbf{q}, 2\mathbf{q})$ ⁴. For this, we can simply

² 14 is the initial pipeline length.

³ The NTTs can be pipelined as well and thus, 23 is the initial pipeline length.

⁴ Since in our implementation all coefficients are stored in the standard representation $[0, q)$, this reduction also works for results of computations $ab + c$, since $(\mathbf{q} - 1)^2 + (\mathbf{q} - 1) < 2^{46}$.

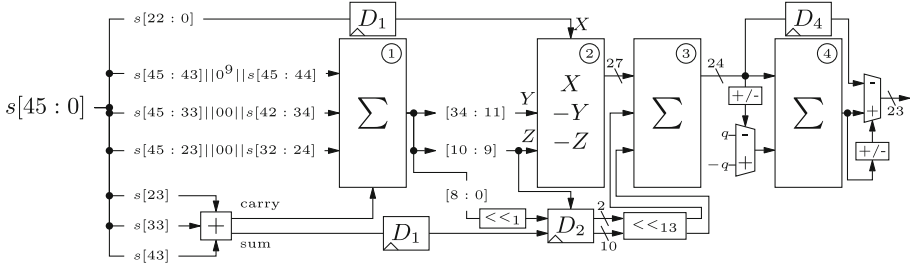


Fig. 3. Architecture of the modular reduction. DSPs are numbered, D_i are shift registers that compensate for the DSPs as given in their respective index. At the output multiplexer, the sign bit is discarded, which decreases the bit width to 23.

add \mathbf{q} to a negative result or subtract \mathbf{q} if the result is positive. Eventually, delaying the reduced result, as well as given the sum or subtraction with \mathbf{q} , the final result is determined by selecting the non-negative value out of both.

In practice, we use four DSPs and one small addition implemented in general-purpose logic to perform the modular reduction. The first DSP computes x and y by using a Kronecker substitution-like approach: The lower bits compute x and the higher bits compute y . However, as the computation does not fit entirely into the pre-adder stage, we need to add the least-significant bit of x using general-purpose logic outside the DSP and delay the resulting bit, while the carry is fed into the DSP as well. Thus, DSP 1 is used as a fully pipelined 4-input adder with a latency of four clock cycles. Note, however, that for recent Ultrascale FPGAs, the width of pre-adder stage within the DSPs increased, which would allow to improve this reduction and give up the general-purpose addition.

Further, the second DSP computes $z + x[11:10] - y$. The result is fed into DSP 3 via a low-latency path, where $(x[11:10] + x[9:0]) \cdot 2^{13}$ are added to the previous result. Note that $x[11:10]$ corresponds to the output bits 10 and 9, and $x[9:1]$ correspond to the output bits 8 to 0 from DSP 1. $x[0]$ has been computed separately before DSP 1 and is delayed accordingly. The fourth and final DSP is connected to the third one via a low-latency path and adds \mathbf{q} if the result of the third DSP is negative or subtracts \mathbf{q} otherwise. Eventually, only the positive result is selected as output.

Keccak. A fundamental part of Dilithium is the application of SHAKE-128 and SHAKE-256, both as hash function or as Extendable-Output Function (XOF). More precisely, both functions use the same Keccak permutation with the same state size of 1500 bits but a different rate r , which either is 1344 bits for SHAKE-128 or 1088 bits for SHAKE-256. Thus, our implementation features a single Keccak core that performs the permutation in 24 cycles (i.e., using a single cycle per round).

For data input and output we decided to implement 32-bit buses. During I/O operations, the Keccak module rotates the internal state for $r = 1344$ on a 32-bit basis, while simultaneously the input is added (exclusive-or) to the rotation

feedback. Note that this behavior can also be used to compute SHAKE-256, i.e., by just using an unaltered feedback for the last $8 = (1344 - 1088)/32$ words.

Sampling. Dilithium requires several sampling algorithms that use the output of SHAKE. Unfortunately, none of the sampling algorithms is aligned to work on 32-bit words. We solved this problem using buffers with a length of the *least common multiple* of 32 and the desired output bit width. This enables converting a stream of 32-bit words to a stream of words with the desired output bit width.

Sampling the challenge c involves the Fisher-Yates shuffle. We implement this using a shift register with runtime-variable depth that contains all offsets of the non-zero coefficients and their sign bit. Once a random offset is found in rejection sampling, we *rotate* through the shift register and compare the stored offsets with the newly sampled one. If they are equal, we replace the old one with the current rejection threshold (keeping the sign bit), which essentially performs the swap. Then we increase the register depth and shift in the newly sampled offset with the corresponding sign bit. Finally, the polynomial is written to the BRAM.

Rounding. Implementing the Power2Round operation in hardware is very efficient, since during the computation of \mathbf{t} , we simply split the result into the upper 10 bits and the lower 13 bits, stored into different polynomial memories. However, since the \mathbf{t}_0 coefficients are interpreted as signed integers and our main paradigm is to store coefficients always as standard representatives, we need to add \mathbf{q} if the most signification bit (MSB) is 1. Due to the structure of the operation, this is efficient with a LUT-based adder, which allows to avoid the additional usage of a DSP.

We implement the HighBits operation as a simple behavioral description of a range look-up depending on the input coefficient, which is efficient since for $\gamma_2 = (q-1)/32$, since there are only 16 different possible output values and only the 15 MSB of a coefficient contribute to the result. For $\gamma_2 = (q-1)/88$, there are only 44 different outputs and only the 13 MSB of a coefficient contribute to the result.

Checking the low bits of $\mathbf{w} - \mathbf{cs}_2$, however, involves the MACC module in subtraction mode. Again, we implement a simple look-up that returns HighBits times $2\gamma_2$ – which is efficient for the same reasons as explained above – and we subtract the result from the coefficient to obtain the low bits and check their norm without storing them.

Hint. We store the hint in two registers, i.e., one storing the 1's offsets and the other one storing the k polynomial boundaries in the same format as specified for the packed signatures. For the MakeHint operation, we have $\mathbf{w} - \mathbf{cs}_2$ and $\mathbf{w} - \mathbf{cs}_2 + \mathbf{ct}_0$ stored separately such that both can be read simultaneously. Eventually, we look up both HighBits and if differing, a new offset is shifted in. Further, for the UseHint operation, the hint module looks up the HighBits for

each coefficient, i.e., both for $h=0$ and $h=1$. Then, selecting the correct one, the value is shifted into a buffer register for sampling (as described before) and absorbed to compute the value \tilde{c} , which ultimately is compared to the value of the signature during verification.

Memory Access. In-place NTTs as deployed in our implementation usually require polynomials to be re-ordered according to a bit-reversal permutation. Our NTT with two BFUs requires reading and writing four coefficients simultaneously, which is ensured by distributing the coefficients according to Eqs. 1 and 2. However, this also ensures that four coefficients with position $br(i), br(i+1), br(i+2), br(i+3)$ (with $4|i$) are placed in different memories. As a consequence, we can access polynomials either in bit-reversed order or in normal order, which eliminates the necessity of an explicit re-ordering operation. Our implementation makes use of this either by sampling polynomials in bit-reversed order or by accessing polynomials in bit-reversed order during NTT.

Single-Task Cores. In order to instantiate single-task cores that only support a subset of operations which are sufficient to perform either key generation, signing, or verification (see Table 1), we adjust the opcode decoder in our top level module and delete all unnecessary module instantiations. Furthermore, we adjust the load module such that only values that are generated by the respective single-task core can be packed and loaded. Similarly, we adjust the store module such that only values can be unpacked and stored that are necessary for the respective single-task core. Thus, by applying slight changes to three files, a set of single-task cores is generated. Note that in order to adjust the security level, a single-line change is sufficient. Additionally, we generate memory mappings for each single-task core that exclude all polynomials which are not used in the respective core, resulting in a lower BRAM usage.

4.2 Utilization and Performance Results

This section provides area utilization and performance results obtained after Place-and-Route (PnR) on a Xilinx XC7A100T Artix-7 FPGA using the Vivado 2020.1 tool suite.

Utilization. Table 2 lists the results for resource utilization as well as the maximum frequency f_{max} obtained after synthesis and implementation. As expected, the LUT, FF, and BRAM utilization increases with the parameter sets, while the DSP utilization, governed by the NTT and MACC modules, is independent of the parameter sets.

Table 2. Resource utilization and performance on a XC7A100T FPGA

Param. Set	Core	Utilization				f_{\max}	KeyGen	Sign _{pre}	Sign	Verify _{pre}	Verify
		LUT	FF	DSP	BRAM	MHz	OP/s	OP/s	OP/s	OP/s	OP/s
II	Full	27433	10681	45	15	163	8692	16905	2435	14938	18595
	Keygen	11064	7209	45	11	221	11772	—	—	—	—
	Sign	18028	9166	45	15	179	—	18557	2673	—	—
	Verify	12118	7551	45	11	200	—	—	—	18331	22819
	Cycles:						18 761	9 647	66 966	10 917	8 770
III	Full	30 900	11 372	45	21	145	4 368	7 993	1 375	7 242	11 966
	Keygen	14 285	8 588	45	17	205	6 203	—	—	—	—
	Sign	21 832	10 245	45	21	174	—	9 603	1 659	—	—
	Verify	14 911	8 209	45	15	200	—	—	—	10 017	16 551
	Cycles:						33 102	18 089	105 129	19 966	12 084
V	Full	44 653	13 814	45	31	140	2 750	4 152	1 250	3 868	8 517
	Keygen	19 319	10 138	45	25	202	3 954	—	—	—	—
	Sign	29 331	12 867	45	31	158	—	4 691	1 412	—	—
	Verify	17 527	9 984	45	23	197	—	—	—	5 424	11 944
	Cycles:						50 982	33 767	112 145	36 250	16 462

Performance. Table 2 shows performance results for our implementations as the average over 1000 executions on random inputs. For signature verification, we report cycle counts for valid signatures only. More precisely, since the norm check of \mathbf{z} , taking less than 100 cycles, is performed at the beginning, an invalid signature is processed substantially faster. Besides, for signature generation, the cycle count spreads widely due to the nature of Dilithium. For the best-case scenario, in which a signature candidate is accepted after the first iteration, signing takes 19423, 26979, and 36609 cycles for Dilithium-II, III, and V, respectively.

Components. Table 3 shows the area consumption obtained after PnR for selected components. Additionally, cycle counts for the single operations are given.

4.3 Comparison to Existing Work

In out-of-context synthesis, we achieved a frequency of 311 *MHz* with a utilization of 524 LUTs, 759 FFs, 17 DSPs and 1 BRAM for our NTT. For NTT/iNTT, our implementation takes 533/536 cycles. In Table 4, we compare our NTT design to others. To the best of our knowledge, we are the first to report detailed performance numbers including latency for the Dilithium NTT as Ricci et al. [13] do not report cycle counts. Thus, we also include NTT implementations for different moduli and polynomial degrees. For a fair comparison, it is worth noting that the polynomial degree n mainly impacts the latency since an NTT has complexity $\mathcal{O}(n \log n)$, while the modulus size $\lceil \log_2 q \rceil$ defines the area of the arithmetic circuit which dominates the overall size.

The implementation from [13] achieves a very high frequency since it operates on the high-end Virtex-7 platform. Other implementations that target the same

Table 3. Area consumption and performance of selected components

Param. Set	Component	Operation	Utilization				Clock cycles
			LUT	FF	DSP	BRAM	
All	NTT	<i>Forward</i> <i>Inverse</i>	444	421	17	1	533 ^P 536 ^P
All	MACC	<i>MACC</i> <i>Add/sub</i>	641	751	24	–	85 ^P 75 ^P
All	Keccak	<i>Permute</i> <i>Absorb</i> <i>Squeeze</i>	3708	1623	–	–	24 ≥42 ^M 1 per 4B
II	Matrix-vector multiplication		2129	59	–	–	2370
III			2774	49	–	–	3019
V			4591	46	–	–	4434
II	Expansion	<i>Expand</i> $\hat{\mathbf{A}}$	198	142	–	–	9647
III			1021	144	–	–	18089
V			1316	144	–	–	33767
II	Sampler	<i>Sample</i> c	312	458	–	–	946
III			411	547	–	–	1417
V			384	662	–	–	2050
II	Sampler	<i>Sample</i> $\mathbf{s}_1, \mathbf{s}_2$	143	44	–	–	3176
III			114	48	–	–	6750
V			163	45	–	–	5953
II	Sampler	<i>Sample</i> \mathbf{y}	244	43	–	–	1654
III			112	42	–	–	2147
V			469	48	–	–	3006

^PMultiple consecutive operations are pipelineable ^MDepending on the master module

polynomial ring size $n = 256$ and target Artix-7 FPGAs are presented in [3, 18]. The first one offers a similar latency like our implementation, but due to the modulus supporting only 7 layers of NTT instead of 8, the gap is larger in practice. Note that regarding LUTs and FFs, our implementation has a similar area usage despite the 10 bit larger modulus. The reason for this is our heavy usage of DSPs. Finally, we compare to three NTT implementations with a smaller gap for the modulus size, but a higher polynomial degree. As expected, these implementations have a higher latency due to the bigger n . However, we expect that our implementation would have a latency of about $3.8 \mu\text{s}$ for $n = 512$ and about $8.4 \mu\text{s}$ for $n = 1024$ at the cost of a minor increase in area usage⁵. Overall, our NTT implementation features a low LUT and FF usage and at the same time, the f_{\max} is, to the best of our knowledge, significantly higher than for any other known design on Artix-7.

It is worth noting, however, that the comparison has several limitations: Our implementation results have been achieved with out-of-context synthesis and subsequent PnR, without connection to the memory that contains the polynomials. For other implementations, like [19], the authors do not report what exactly is contained in the NTT-only implementations and how the utilization and performance numbers are found, although the numbers indicate that some-

⁵ Doubling the polynomial degree can be achieved by increasing the size of an internal counter by 1 bit.

Table 4. Comparison of hardware designs for NTT implementations

(n, q)	Platform	Utilization				f	t	Ref.
		LUT	FF	DSP	BRAM	MHz	μs	
256, 8380417	XC7A100T	524	759	17	1	311	1.7	This
256, 8380417	XCVU7P	1798	2532	48	3.5	637	—	[13]
256, 3329	XC7A35T	609	640	2	4	257	1.9	[18]
256, 7681	XC7A200T	533	514	1	3	—	17.1	[3]
256, 7681	XC7A200T	479	472	1	2	—	16.7	[3]
1024, 12289	XC7Z020	847	375	2	6	244	10.5	[19]
512, 12289	XC7Z020	741	330	2	5	245	5.3	[19]
512, 12289	V6LX75T	994	944	1	3	278	14.8	[14] ^E

^EExcluding area usage for sampler and random number generator.

how a polynomial memory is connected. The advantage of our approach, to use an out-of-context synthesis without memory connection, is that a good approximation of the real f_{max} of the *arithmetic* is given. The operational frequency for a design that features our NTT then obviously depends *additionally* on the exact memory layout of the overall design, which however is not depending on the NTT itself.

In Table 5, we compare our implementation of Dilithium-III with other relevant implementations of post-quantum signature schemes on reconfigurable hardware. In contrast to existing implementations of Dilithium for Artix-7 [16] and Virtex-7 [13] which report area utilization, frequency, and latency individually *per operation*, we would like to emphasize that in addition to our single-task cores, our full core combines and embeds all operations in a single architecture. Additionally, since our cores feature precomputation operations for signing and verification, performing these for multiple messages under the same key can be speeded up significantly. In particular, for signing, 104 μs are spent on precomputations for the single-task core at security level III. For verification, precomputations take 100 μs at security level III, so the actual verification latency is about 60 μs in that case.

Notably, our architecture outperforms existing solutions either in terms of resource utilization or throughput thus provides a compact, self-contained, and efficient solution for post-quantum secure digital signatures. In general, our design focuses on a reasonable trade-off between area consumption and performance degradation, in order to provide a modestly large and fast architecture.

Table 5. Comparison of hardware design for PQC signature schemes

Oper.	Scheme	Platform	Utilization				f	t	Ref.
			LUT	FF	DSP	BRAM	MHz	μs	
KeyGen	Dilithium-III ^F	XC7A100T	30900	11372	45	21	145	229	This
	Dilithium-III	XC7A100T	14285	8588	45	17	205	161	This
	Dilithium-III ^R	XC7VU7P	54183	25236	182	15	350	52	[13]
	Dilithium-III ^{R,H}	Artix-7	86646	17674	–	–	119	1955	[16]
	qTesla-3 ^H	Artix-7	111122	23398	–	–	79	45650	[16]
Sign	Dilithium-III ^F	XC7A100T	30900	11372	45	21	145	852	This
	Dilithium-III	XC7A100T	21832	10245	45	21	174	709	This
	Dilithium-III ^R	XC7VU7P	81530	83926	965	145	333	63	[13]
	Dilithium-III ^{R,H}	Artix-7	90567	21160	–	–	114	14140	[16]
	qTesla-3 ^H	Artix-7	126008	25984	–	–	79	7441	[16]
	GLP	Spartan-6	7465	8993	28	29.5	–	1074	[11]
	Rainbow-Ia ^C	Kintex-7	27712	27679	0	59	111	18	[6]
	Rainbow-Ic ^C	Kintex-7	52895	32476	0	67	90	11	[6]
	SPHINCS-256	Kintex-7	19067	38132	3	36	525	1530	[1]
Verify	Dilithium-III ^F	XC7A100T	30900	11372	45	21	145	222	This
	Dilithium-III	XC7A100T	14911	8209	45	15	200	160	This
	Dilithium-III ^R	XC7VU7P	61738	34963	316	18	158	95	[13]
	Dilithium-III ^{R,H}	Artix-7	65274	15169	–	–	114	2491	[16]
	qTesla-3 ^H	Artix-7	84834	17604	–	–	79	1926	[16]
	GLP	Spartan-6	6225	6663	8	15	–	1002	[11]

^FFull core ^RRound-2 parameters ^HHigh level synthesis ^CCore enabling signing and verification

5 Conclusion

In this work, we present the first set of FPGA implementations for all three round-3 parameter sets of Dilithium for the low-end Artix-7 platform. Our design follows a universal design goal, featuring low latency compared to implementations of other post-quantum secure signature algorithms on the one hand, but still having a low area footprint on the other hand, making the usage of Dilithium feasible for many low-cost and constrained scenarios. As a highlight, our implementations can be used as full-service processors for Dilithium, being capable of performing key generation, precomputations, signature generation, verification, arbitrary-length message digesting as well as key and signature packing and unpacking.

Acknowledgments. The work described in this paper has been supported in part by the Deutsche Forschungsgemeinschaft (DFG, German Research Foundation) under Germany’s Excellence Strategy - EXC 2092 CASA - 390781972, by the H2020 project PROMETHEUS (grant agreement ID 780701), and by the Federal Ministry of Education and Research of Germany through the QuantumRISC (16KIS1038) and PQC4Med (16KIS1044) projects.

References

1. Amiet, D., Curiger, A., Zbinden, P.: FPGA-based accelerator for post-quantum signature scheme SPHINCS-256. *IACR Trans. Cryptogr. Hardw. Embed. Syst.* (2018)
2. Banerjee, U., Ukyab, T.S., Chandrakasan, A.P.: Sapphire: a configurable crypto-processor for post-quantum lattice-based protocols. *IACR Trans. Cryptogr. Hardw. Embed. Syst.* (2019)
3. Chen, Z., Ma, Y., Chen, T., Lin, J., Jing, J.: High-performance area-efficient polynomial ring processor for crystals-kyber on FPGAS. *Integr.* (2021)
4. Ducas, L., et al.: A lattice-based digital signature scheme. *IACR Trans. Cryptogr. Hardw. Embed. Syst.*, CRYSTALS-Dilithium (2018)
5. Ducas, L., et al.: CRYSTALS-Dilithium - Algorithm Specifications and Supporting Documentation (Version 3.1). Technical Report (2021). <https://pq-crystals.org/dilithium/data/dilithium-specification-round3-20210208.pdf>
6. Ferozpur, 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 2018 (2018)
7. Fouque, P.-A., et al.: Falcon: Fast-Fourier Lattice-based Compact Signatures over NTRU - (Specification v1.2 - 01/10/2020). Technical Report (2020). <https://falcon-sign.info/falcon.pdf>
8. Greconici, D.O.C., Kannwischer, M.J., Sprenkels, D.: Compact dilithium implementations on Cortex-M3 and Cortex-M4. *IACR Trans. Cryptogr. Hardw. Embed. Syst.* (2021)
9. Güneysu, T., Lyubashevsky, V., Pöppelmann, T.: Practical lattice-based cryptography: a signature scheme for embedded systems. In: Prouff, E., Schaumont, P. (eds.) CHES 2012. LNCS, vol. 7428, pp. 530–547. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-33027-8_31
10. NIST. Call for Proposals - Post-Quantum Cryptography — CSRC. Technical Report, NIST (2017). <https://csrc.nist.gov/Projects/Post-Quantum-Cryptography/Post-Quantum-Cryptography-Standardization/Call-for-Proposals>
11. Pöppelmann, T., Ducas, L., Güneysu, T.: Enhanced lattice-based signatures on reconfigurable hardware. In: Batina, L., Robshaw, M. (eds.) CHES 2014. LNCS, vol. 8731, pp. 353–370. Springer, Heidelberg (2014). https://doi.org/10.1007/978-3-662-44709-3_20
12. Post Quantum Cryptography Team. Post-Quantum Cryptography: NIST’s Plan for the Future. Technical Report, NIST (2016). <https://csrc.nist.gov/csrc/media/projects/post-quantum-cryptography/documents/pqcrypto-2016-presentation.pdf>
13. Ricci, S., et al.: Implementing crystals-dilithium signature scheme on FPGAS. In: Reinhardt, D., Müller, T. (eds.) ARES 2021: The 16th International Conference on Availability, Reliability and Security, Vienna, 17–20 August 2021, pp. 1:1–1:11. ACM (2021)
14. Roy, S.S., Vercauteren, F., Mentens, N., Chen, D.D., Verbaauwhede, I.: Compact ring-LWE cryptoprocessor. In: Batina, L., Robshaw, M. (eds.) CHES 2014. LNCS, vol. 8731, pp. 371–391. Springer, Heidelberg (2014). https://doi.org/10.1007/978-3-662-44709-3_21
15. Shor, P.W.: Algorithms for quantum computation: discrete logarithms and factoring. In: 35th Annual Symposium on Foundations of Computer Science (1994)

16. Soni, D., Basu, K., Nabeel, M., Karri, R.: A hardware evaluation study of NIST post-quantum cryptographic signature schemes. In: Second PQC Standardization Conference (2019)
17. Thoma, J.P., Güneysu, T.: A configurable hardware implementation of XMSS. IACR Cryptol. ePrint Arch. (2021)
18. Zhang, C., et al.: Towards efficient hardware implementation of NTT for Kyber on FPGAS. In: 2021 IEEE International Symposium: On Circuits and Systems (ISCAS) (2021)
19. Zhang, N., Yang, B., Chen, C., Yin, S., Wei, S., Liu, L.: Highly efficient architecture of NewHope-NIST on FPGA using low-complexity NTT/INTT. IACR Trans. Cryptogr. Hardw. Embed. Syst. (2020)