Implementing CRYSTALS-Dilithium Signature Scheme on FPGAs *

Sara Ricci$^{[0000-0003-0842-4951]}$, Lukas Malina$^{[0000-0002-7208-2514]}$, Petr Jedlicka$^{[0000-0003-0833-8068]}$, David Smekal$^{[0000-0002-1996-5334]}$, Jan Hajny$^{[0000-0003-2831-1073]}$, Petr Cibik$^{[0000-0003-0780-6288]}$, and Patrik Dobias

Department of Telecommunications, Brno University of Technology, Brno, Czech Republic
Tel.: +420 541 146 990
{ricci,malina,hajny}@feec.vutbr.cz, xjedli23@stud.feec.vutbr.cz, cibik@netcope.com, xdobia13@vutbr.cz

Abstract. In July 2020, the lattice-based CRYSTALS-Dilithium digital signature scheme has been chosen as one of the three third-round finalists in the post-quantum cryptography standardization process by the National Institute of Standards and Technology (NIST). In this work, we present the first Very High Speed Integrated Circuit Hardware Description Language (VHDL) implementation of the CRYSTALS-Dilithium signature scheme for Field-Programmable Gate Arrays (FPGAs). Due to our parallelization-based design requiring only low numbers of cycles, running at high frequency and using reasonable amount of hardware resources on FPGA, our implementation is able to sign 15832 messages per second and verify 10524 signatures per second. In particular, the signing algorithm requires 68461 Look-Up Tables (LUTs), 86295 Flip-Flops (FFs), and the verification algorithm takes 61738 LUTs and 34963 FFs on Virtex 7 UltraScale+ FPGAs. In this article, experimental results for each Dilithium security level are provided and our VHDL-based implementation is compared with related High-Level Synthesis (HLS)-based implementations. Our solution is ca 114 times faster (in the signing algorithm) and requires less hardware resources.

Keywords: Post-quantum cryptography · Post-quantum cryptography · Digital signatures · Number-theoretic transform · FPGA · VHDL implementation · Parallelization · Optimization.

1 Introduction

Nowadays, security of most well-established public key cryptosystems relies on Non-Polynomial (NP) time complexity problems, namely integer factorization, discrete logarithm, and elliptic curve discrete logarithm problems. Unfortunately, these closely related NP-problems are vulnerable to quantum computer attacks.

* This work is supported by Ministry of the Interior of the Czech Republic under grant VI20192022126.

Table 1. VHDL and HLS implementations of NIST PQC finalists.

Digital Signature			
Scheme	Type	HLS method	VHDL implementation
Dilithium	lattice	✓	✗
Falcon	lattice	✓	✗
Rainbow	multivariate	✓	✓
Encryption/KEM			
Scheme	Type	HLS method	VHDL implementation
Kyber	lattice	✓	✗
McEliece	code	✓	✓
NTRU	lattice	✓	✓
SABER	lattice	✓	✓

Note: ✓– algorithm is fully implemented, ✗– algorithm is not implemented.

The main threat arrives from the Shor's algorithm [5] which allows attackers to solve discrete logarithm and integer factorization problems, and therefore attacks the asymmetric cryptosystems based on them. Furthermore, symmetric key cryptography does not remain untouched either. In fact, the Grover's algorithm [5] simplifies the collision and symmetric key brute force search to sub-linear complexity which results in an increase of keys and parameter sizes in algorithms [7]. It is an interesting fact that both algorithms need to run on a quantum computer with a minimal-required number of logic qubits which has not been physically reached yet. For instance, Shor's algorithm requires 4000 logical qubits to break 2048-bit RSA keys [14], and current quantum computers capable to run Shor's algorithm only have about 20 logical qubits [13]. However, a significant number of experts and practitioners believe that such a quantum computer can be built in the next decade and concretely pose a danger to current cryptographic primitives [4, 17]. In 2016, the NIST initiated a process to solicit, evaluate, and standardize one or more Post-Quantum Cryptography (PQC) schemes [1], i.e. quantum-resistant digital signatures and Key Encapsulation Mechanisms (KEMs). In the 3rd round, 3 signature and 4 KEM finalists were selected as potential future standards from the competing 64 candidates. In particular, two lattice-based, CRYSTALS-Dilithium and Falcon, and one multivariate, Rainbow, signature schemes were selected. The schemes were chosen based on their security strength and software/hardware performance.

Lattice-Based Cryptography (LBC) is one of the families of primitives relying on hard problems which are believed to be secure against quantum-computing attacks. LBC has gained significant attention for its performance among the PQC families for both KEMs and digital signatures. Especially the CRYSTALS-Dilithium (shortly Dilithium), which is a lattice-based digital signature and a NIST finalist, provides a reasonable parameter size and promising performance [3, 12]. Several implementations of this scheme on different devices are currently accessible, e.g., C implementations [8], and HLS-based implementations [3, 20]. Nevertheless, a pure VHSIC-based implementation of Dilithium still does not exist.

To the best of our knowledge, we introduce a first pure VHDL-based implementation of the CRYSTALS-Dilithium digital signature for FPGAs. The target FPGA platform for our implementation is a widely-used chip from Xilinx, namely the Virtex 7 UltraScale+. This chip can be found in many computer components, in network cards and cryptographic accelerators in particular. Especially, FPGA network cards are widely used as cryptographic accelerators to speed up security and cryptography functions in high-performance communication systems, and therefore they could be suitable platforms for evaluating pre-standardized quantum-resistant schemes such as the Dilithium scheme. In this work, we present our original design and optimized VHDL implementation of the Dilithium scheme and our results which are indicating significant performance primacy over software-based and HLS-based related implementations.

This paper is organized as follows: the rest of this section contains related work and our contribution. Section 2 introduces the Dilithium digital signature. Section 3 presents details of our hardware implementation and explains our design decisions. Section 4 discusses the results of our implementation and provides the comparison with other related Dilithium implementations. In the last section, we conclude this work.

1.1 Related Work

Table 1 shows the current state of NIST PQC finalists VHDL-based and HLS-based implementations on the FPGA platform. All protocols have HLS-based implementations which have been published in different articles. HLS provides an automatic conversion from existed C, C ++ or Matlab implementations into HDL. Nevertheless, the outputs from HLS are often less efficient than native VHDL-based implementations, and some outputs could have functional errors or security bugs. In detail, Soni *et al.* [19] investigate NIST 2nd round PQC signatures by synthesizing the C-based code by using HLS on FPGA. They focus on Power-Performance-Area-Security (PPAS) trade-offs, design flows and implementation-resilience to a variety of side-channel attacks. In another article, Soni *et al.* [20] compare HLS implementations of two 2nd round NIST PQC semi-finalists, namely qTESLA and Dilithium schemes. The authors use the Xilinx Vivado HLS method and present the results of both schemes on FPGA (Xilinx Artix-7). They show that at lower security levels Dilithium has slightly lower hardware requirements than qTESLA. Basu *et al.* [3] employ the HLS method to implement and compare 11 2nd round NIST PQC semifinalists on Xilinx Virtex-7 FPGA platform. They cover all KEM finalists and the Dilithium signature scheme. They state several key points evaluation results as conclusions. For instance, they show that Dilithium is superior for the signing algorithm, qTesla is superior for the verification algorithm and SPHINCS+ is the costliest in terms of latency and latency-area product. In case of KEMs, CRYSTALS-Kyber is the fastest scheme for security level 1.

We are not aware of any hardware implementations (i.e., VHDL-based) of Dilithium, while Rainbow signature and McEliece, NTRU, and SABER schemes have been already implemented. Ferozpuri and Gaj [9] present the design and hardware implementation of Rainbow on Xilinx Virtex 7 (XC7VX1140) and Kintex-7 (XC7K480) FPGA platforms. In case of KEMs, Wang *et al.* [21] provide the full Niederreiter cryptosystem implementation on the Virtex-6 XC6VLX240T. This cryptosystem is the dual variant of the Classic McEliece scheme. At last, Roy and Basso [18] deal with the hardware implementation of SABER using the FPGA Xilinx ZCU102 board. In their study, they compare their results with existing implementations of SABER and few other PQC NIST candidates.

1.2 Contributions

To the best of our knowledge, this is the first VHDL-based implementation of the CRYSTALS-Dilithium signature scheme on FPGA that is created natively, without using High-Level Synthesis. In particular, we make the following contributions:

– We design and implement in VHDL all underlying functions used in Dilithium such as SHAKE-128, SHAKE-256, $\texttt{ExpandA}_q$, $\texttt{ExpandMask}_q$, $\texttt{PowerToRound}_q$, $\texttt{MakeHint}_q$. All mentioned functions are implemented in VHDL from scratch in order to get better performance at the FPGA platform (high frequency, low number of cycles).
– We design and implement essential Number-Theoretic Transform (NTT) functions in VHDL and optimize these functions for a hardware environment.
– We integrate the functions into the main Dilithium algorithms: `key generation`, `signing` and `verification`. For instance, a loop (while cycle) in the signing algorithm is designed and implemented in order to be efficient (high frequency, low number of cycles). The parallelization approach has been applied during the design of all algorithms and significant blocks.
– All algorithms are measured, tested and verified based on the reference implementation of Dilithium in the C programming language [8]. The results are also compared with related C-based and HLS-based implementations, and indicate a significant performance improvement in all algorithms.

2 Preliminaries

In this section, we discuss the mathematical background that is crucial for the understanding of our implementation. In Section 2.2, we revise the number-theoretic transformation. The Dilithium signature is described in Section 2.3.

2.1 Notation

In this section, we introduce the notation used throughout the paper. Let n and q be two integers, i.e. $n = 256$ and $q = 8380417 = 2^{23} - 2^{13} + 1$. We denote by $\mathcal{R}_q$ the polynomial ring $\mathbb{Z}[x]_q/(x^n + 1)$ where $x^n + 1$ is the modulus. Bold lower-case letters ($\mathbf{v}$) are used for column vectors in $\mathcal{R}$ or $\mathcal{R}_q$, while regular font letters (v) for elements in $\mathcal{R}$ or $\mathcal{R}_q$. Matrices are represented by bold upper-case letters ($\mathbf{A}$).

2.2 Number-Theoretic Transform (NTT)

The NTT is a generalization of the discrete Fourier transform over a finite field [15]. An interesting property of the discrete Fourier transform is the reduction of the overall complexity of (polynomial) multiplication to $O(n \log n)$. This is due to the usage of the point value representation of a polynomial instead of the coefficient representation.

In NTT, a polynomial becomes a multi-point evaluation at powers of a root of unity. Therefore, the polynomial multiplication consists in applying NTT in $O(n \log n)$, then performing point-wise multiplication in $O(n)$ and finally converting the result to a coefficient representation in $O(n \log n)$. This process can be synthesized by the following formula,

$$f(x) \times g(x) = NTT^{-1}(NTT(f) \odot NTT(g)),$$

where $\odot$ is the point-wise multiplication of the coefficients.

In order to allow efficient computation of the NTT the coefficient ring has to contain primitive roots of unity. Dilithium's modulus q is chosen such that there exists a 512-th root of unity r modulo q, where $r = 1753$. Since $\mathcal{R}_q$ is isomorphic to $\prod_i \mathbb{Z}_q/(X - r^i)$ a polynomial $f(x)$ can be represented as $(f(r), f(r^3), \dots f(r^{511}))$, which are Dilithium's NTT output vectors with coefficients in the order $f(r), f(r^3), \dots f(r^{511})$.

There are many ways to compute the number-theoretic transform. Dilithium requires the use of Cooley-Tukey butterflies in NTT, Gentleman-Sande butterflies in NTT^{-1}, and the Montgomery algorithm for modular reductions after multiplying with a precomputed root of unity [8]. Note that roots of unity are in modulo arithmetic, therefore the Montgomery reduction is required, more details in [8].

2.3 CRYSTALS-Dilithium Signature

CRYSTALS-Dilithium signature [8] is part of the Cryptographic Suite for Algebraic Lattices (CRYSTALS), which counts a KEM, namely Kyber, and a signature, namely Dilithium. Both protocols' security relies on the hardness of the Module variant of the Learning With Error (MLWE) problem [6, 11]. For Dilithium's MLWE problem, $\mathbf{A}$ is a $k \times l$ matrix of polynomials, whereas $\mathbf{s}$ and $\mathbf{e}$ become l-dimensional and k-dimensional vectors, respectively. Informally, the MLWE problem can be viewed as the Ring-LWE problem where the single ring elements ($\mathbf{a}$ and $\mathbf{s}$) are replaced with module elements over the same ring. Note that the MLWE problem has been introduced since it might be able to offer a better level of security than the Ring-LWE, while offering advantages in performance with respect to plain LWE [2].

Dilithium uses the MLWE problem with n and q fixed. The security level of this signature changes by simply changing the dimension of the matrix $\mathbf{A}$, i.e. by changing k and l. Therefore, since $\mathcal{R}_q$ is the same for all security levels it is possible to optimize all Dilithium security levels by optimizing the operations in $\mathcal{R}_q$. This makes it easy to vary security. Dilithium specifies four sets of parameters: weak, medium, recommended and very high which use $\mathbf{A}$ dimensions $(k, l) = (3, 2), (4, 2), (5, 4)$ and $(6, 5)$, respectively. For the recommended security level, the scheme has 2.7KB signatures and 1.5KB public keys.

Algorithm 1 Key Generation `KeyGen()`

```
1: $\rho, K \leftarrow \{0,1\}^{256}$
2: $(\mathbf{s}_1, \mathbf{s}_2) \in S_\eta^\ell \times S_\eta^k := \texttt{ExpandA}(K)$
3: $\mathbf{A} \in R_q^{k \times \ell} := \texttt{ExpandA}(\rho)$
4: $\mathbf{t} := \mathbf{A}\mathbf{s}_1 + \mathbf{s}_2$
5: $(\mathbf{t}_1, \mathbf{t}_0) := \texttt{Power2Round}(\mathbf{t}, d)$
6: $tr \in \{0,1\}^{384} := \texttt{CRH}(\rho || \mathbf{t}_1)$
7: return $pk = (\rho, \mathbf{t}_1),\ sk = (\rho, K, tr, \mathbf{s}_1, \mathbf{s}_2, \mathbf{t}_0)$
```

In this section, we briefly describe the Dilithium scheme and we refer to the original article [8] for further information. Dilithium is composed by three algorithms: `Key Generation`, `Signing` and `Verification` as shown in Algorithms 1, 2 and 3. In particular, each algorithm employs few internal functions which are used more times and/or in more phases. Table 2 maps these basic functions.

Algorithm 2 Signing `Sign(`sk, M`)`

```
1: $\mathbf{A} \in R_q^{k \times \ell} := \texttt{ExpandA}(\rho)$
2: $\mu \in \{0,1\}^{384} := \texttt{CRH}(tr || M)$
3: $\kappa := 0$, $(\mathbf{z}, \mathbf{h}) := \perp$
4: while $(\mathbf{z}, \mathbf{h}) := \perp$ do
5:     $y \in S_{\gamma_1 - 1}^\ell := \texttt{ExpandMask}(K || \mu || \kappa)$
6:     $\mathbf{w} := \mathbf{A}\mathbf{y}$
7:     $\mathbf{w}_1 := \texttt{HighBits}_q(\mathbf{w}, 2\gamma_2)$
8:     $c \in B_{60} := \texttt{H}(\mu || \mathbf{w}_1)$
9:     $\mathbf{z} := \mathbf{y} + c\mathbf{s}_1$
10:    $(\mathbf{r}_0, \mathbf{r}_1) := \texttt{Decompose}_q(\mathbf{w} - c\mathbf{s}_2, 2\gamma_2)$
11:    if $\|\mathbf{z}\|_\infty \geq \gamma_1 - \beta$ or $\|\mathbf{r}_0\|_\infty \geq \gamma_2 - \beta$ or $\mathbf{r}_1 \neq \mathbf{w}_1$ then $(\mathbf{z}, \mathbf{h}) := \perp$
12:    else
13:        $\mathbf{h} := \texttt{MakeHint}_q(-c\mathbf{t}_0, \mathbf{w} - c\mathbf{s}_2 - c\mathbf{t}_0, 2\gamma_2)$
14:        if $\|c\mathbf{t}_0\|_\infty \geq \gamma_2$ or the # of 1's in $\mathbf{h}$ is greater than $w$ then $(\mathbf{z}, \mathbf{h}) := \perp$
15:    $\kappa = \kappa + 1$
16: return $\sigma = (\mathbf{z}, \mathbf{h}, c))$
```

Algorithm 1 depicts the `Key Generation`. It has two main parts: (1) the expansion of two random seeds ρ and K to $\mathbf{A}$ and $(\mathbf{s}_1, \mathbf{s}_2)$, respectively, by the extendable output function SHAKE-128, and (2) the computation of the remaining components of both public and secret keys. SHAKE-256 is used as the collision resistant hash (CRH) in all algorithms. Note that $\mathbf{A}$ is directly given in the NTT domain representation, i.e. its elements are polynomials represented as vectors.

In Algorithm 2, the `Signing` generates a masking vector of polynomials $\mathbf{y}$, computes $\mathbf{Ay}$ and then considers $\mathbf{w}_1$, the "high-order" bits of $\mathbf{Ay}$ coefficients. The challenge is created by hashing $\mathbf{w}_1$ and the message M. The functions $\texttt{Power2Round}_q$, $\texttt{Decompose}_q$, $\texttt{HighBits}_q$ and $\texttt{LowBits}_q$ permit selecting properly $\mathbf{w}_1$, while $\texttt{MakeHint}_q$ and $\texttt{UseHint}_q$ reconstruct "the missing" bits for the `Verification` stage. This procedure allows reducing the public key by a factor of around 2.5 at the expense of additional hundred bytes in the signature. SHAKE-256 is used for the generation of $\mathbf{y}$ by the `ExpandMask` function and in the `H` function.

Algorithm 3 Verification $\texttt{Verify}(pk, M, \sigma = (\mathbf{z}, \mathbf{h}, c))$

```
1: $\mathbf{A} \in R_q^{k \times \ell} := \texttt{ExpandA}(\rho)$
2: $\mu \in \{0,1\}^{384} := \texttt{CRH}(\texttt{CRH}(\rho || \mathbf{t}_1) || M)$
3: $\mathbf{w}'_1 := \texttt{UseHint}_q(\mathbf{h}, \mathbf{Az} - c\mathbf{t}_1 \cdot 2^d, 2\gamma_2)$
4: return $\|\mathbf{z}\|_\infty < \gamma_1 - \beta$ and $c := \texttt{H}(\mu || \mathbf{w}_1)$ and $\#$ of 1's in $\mathbf{h}$ is $\leq w$
```

At last, the `Verification` is shown in Algorithm 3. The verifier computes $\mathbf{w}'_1$ and accepts the signature $\mathbf{z}$ if it is small enough (see the original article [8] for more details).

3 VHDL Implementation

This section describes our design and implementation of the Dilithium signature scheme and its main blocks in VHDL.

3.1 Methodology and Implementation of Chosen Functions

This section describes the methodology used through the article and the implementation of main basic functions as the components that form the basic operations for the Dilithium algorithms. Our hardware implementation of Dilithium is mainly based on our VHDL source codes. We do not use HLS in order to speed up the development process. The goal of avoiding HLS and using the pure VHDL approach is to provide a more efficient implementation (best trade off between used hardware sources and performance) than the HLS-based implementations [3, 20].

The implementation methodology consists of these steps:

1. *Implementation of individual functions* - some functions have been implemented from scratch and some primitive functions are based on public implementations. Table 2 shows an overview of the used functions with implementation specifications.

Table 2. Mapping Dilithium functions depending on where they are used. "All" states for "used in all algorithms", "Gen" for "used in `Key Generation`", "Sig" for "used in `Signing`" and "Ver" for "used in `Verification`".

Function	**Algorithm**	**Our impl.**	**Note**
Keccak	All	✓	Variants SHAKE-128 and SHAKE-256
ExpandA$_q$	All	✓	Using our Keccak SHAKE-128
ExpandMask$_q$	Sig	✓	Using our Keccak SHAKE-256
CRH	All	✓	Using our Keccak SHAKE-256
Hashing to a Ball	Sig	✓	Using our Keccak SHAKE-256
Random Number Generator	Gen	✗	Using LFSR-Random number generator [10]
NTT	All	✓	Using our 4 butterflies and Montgomery reduction
NTT^{-1}	All	✓	Using our 4 butterflies and Montgomery reduction
Decompose$_q$	Sig, Ver	✓	Special Dilithium procedure
PowerToRound$_q$	Gen	✓	Reduction with input coefficient division
HighBits$_q$	Sig	✓	Special Dilithium procedure
LowBits$_q$	Sig	✓	Special Dilithium procedure
MakeHint$_q$	Sig	✓	Special Dilithium procedure
UseHint$_q$	Ver	✓	Special Dilithium procedure

Note: ✓– our implementation, ✗– existing algorithm with reference in Note column.

2. *Testing the functionality and validity of implemented functions* - each function has been tested and optimized in order to be efficient from the performance (high frequency), and hardware resources (low number# of FFs and LUTs) perspective.
3. *Implementation of the Dilithium algorithms (security level III.)* - the integration of verified functions.
4. *Testing the functionality and validity of implemented Dilithium algorithms (security level III.)* - each algorithm has been verified by comparing input/ouput validity based on the reference C implementation [8].
5. *Extending the recommended algorithms* - the implementation has been extended by more security levels, i.e. I., II. and IV. variants.
6. *Performance testing and comparison* - getting the experimental results.

As shown in Table 2, we decided to implement the main functions of the Dilithium signature from scratch. This is due to the fact that our implemented algorithms are optimized to reach as high clock frequency as possible, and adapted for the 512-bit bus that is used in the chosen FPGA board (UltraScale+). Moreover, some algorithms were not implemented yet or made publicly available. More details are given in the description of the components below.

Keccak Component (SHAKE-128, SHAKE-256). Our implementation of the hash function Keccak is straight-forward. This component is the core of the extendable output functions **SHAKE-128** (‘r‘ = 1344) and **SHAKE-256** by specific settings of the generic parameter ‘r‘ (‘r‘ = 1088). SHAKE-128 and SHAKE-256 are used for generation of the matrix $\mathbf{A}$ and vectors/polynomials, e.g. the secret key components $\mathbf{s_1}$ and $\mathbf{s_2}$. The Keccak component absorbs the prepared input data with specific "101" padding in blocks of ‘r‘ bits and squeezes the output hash also into blocks of ‘r‘ bits. This component uses the standard Advanced eXtensible Interface (AXI) interface and has a simple implementation which could be easily adapted and optimized depending on the use. This VHDL implementation takes 24 time cycles due to the usage of parallelization during absorbance and squeezing of data [16].

ExpandA and ExpandMask Components. $\texttt{ExpandA}_q$ and $\texttt{ExpandMask}_q$ use the Keccak component set as the SHAKE-128 and SHAKE-256 hash functions, respectively. Both components use the standard AXI interface. In particular, there is a 256-bit seed at the input of $\texttt{ExpandA}_q$ components and the generated coefficients of the $\mathbf{A}$ matrix of 32 bits are written to the output one after the other. By setting a generic parameter, you can change the generation of matrix coefficients row by row or column by column. The $\texttt{ExpandA}_q$ component is located in all algorithms of the Dilithium scheme and is used to generate the uniform matrix $\mathbf{A}$ in the NTT domain representation. On the contrary, $\texttt{ExpandMask}_q$ belongs only to the `Signing` algorithm and it is used for the generation of the vector $\mathbf{y}$.

CRH Component. The CRH component uses the standard AXI interface and is based on the SHAKE-256 hash function. This function produces a 384-bit output. The CRH component is located in all algorithms of the Dilithium scheme. This component is implemented from scratch since there are no existing implementations in VHDL.

Hashing to a Ball Component. At the input of this component is a 64-bit signal, to which μ is firstly applied in blocks and then the coefficients of the vector of polynomials $\mathbf{w}$. At the output, it generates a 512-bit vector representing the values 1 and -1 of the polynomial c. The first part of 256 bits indicates the indices on which the polynomial contains 1, the second part indicates the indices of -1. This feature uses the SHAKE-256 and needed to be implemented from scratch since there are no existing implementations in VHDL.

Random Number Generator (RND) Component. Our Random number generator component is based on a linear feedback shift register (LFSR). LFSR is a shift register whose input bit is created by a linear function of its previous state. The output of the generator depends on the feedback function and the

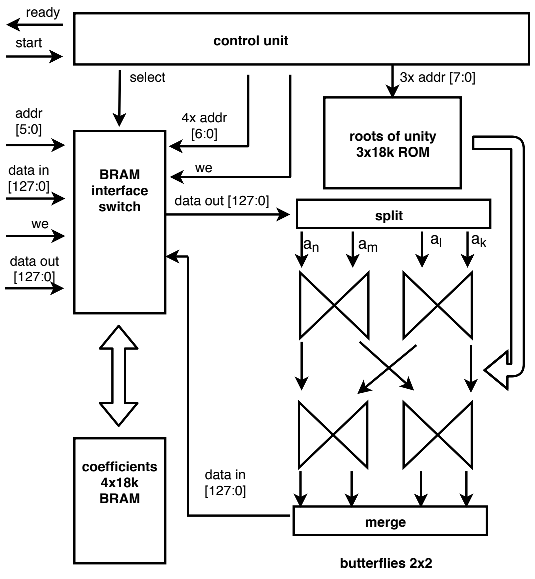

Fig. 1. Block scheme of NTT implementation for FPGA.

initial state called the seed. Thus, 3 signals come to the input of the generator - a clock pulse (clk), a control element (set_seed) and the mentioned seed (seed). For example, a three-bit random number (rand_out) is obtained at the generator output. The RND component uses Lal's open source VHDL implementation [10] which allows fast number generation (350 MHz, 384 LUTs and 128 FFs) and follows Dilithium specifications [8].

NTT and NTT^{-1} Components. Our described implementation is based on Fast Fourier Transform with decimation in the frequency domain and follows Dilithium specifications [8]. The transformation takes place over 256 samples, which corresponds to the calculation performed in 8 iterations ($256 = 2^8$). Within each iteration, 128 partial transformations are calculated using butterflies. The difference in the order of the coefficients entering this partial transformation is $\frac{128}{2^{n-1}}$, where n denotes the order of the iteration with indexing starting from 1. To compromise between the speed of calculation and hardware resources, the parallelization of the calculation using 4 butterflies in a 2x2 arrangement is deployed. This means a calculation in two iterations at the same time with two butterflies in each of them. The butterflies in the first iteration have to be shifted by a value corresponding to the required difference in the coefficients of the following iteration. The reason for spreading the calculation between two iterations is to reduce the number of output coefficients (for one iteration) and to minimize the number of Block Random Access Memory (BRAM) for storing intermediate results and to reduce the number of interludes between iterations. Each block memory has only two ports. Ports could be overwritten or read at the same time.

Moreover, the elementary arithmetic operations (adding and multiplication) in the Montgomery algorithm also run in parallel using digital signal processor (DSP) blocks in the FPGA platform. The computational structure of the Montgomery reduction is pipelined to reach the maximum clock frequency and have small latency between the first valid input and the first valid output while the component throughput is 1 output per 1 clock cycle. The block diagram of the NTT component is shown in Figure 1. Four BRAMs are used to store input values, intermediate results and output values. In the inactive mode, the interface of these memories is switched by the control unit to the component interface, then inputs and outputs can be written or read. During calculating the transformation, the memory interface is made available to the control unit, which sets the addresses, and to the computational structure consisting of butterflies in a 2x2 layout connected to the data buses. Four coefficients are transmitted in parallel over the data bus, which are divided between the inputs of the butterflies and merged again at the output. The last block is the ROM memory with the roots of the unit equation (so-called roots of unity), whose address is again controlled by the control unit and the data bus is connected to the computational structure.

The design of the Inverse NTT (NTT^{-1}) component is almost identical to the NTT scheme. Only the iterations are performed in the reverse order, and thus the distribution of the butterflies in the first and second parallel iterations is inverse. In contrast to NTT, the Montgomery reduction of the output coefficients is performed at the end of the whole calculation.

Other Dilithium Components. Besides the components described above, our hardware implementation of the Dilithium scheme also includes other components such as $\texttt{Decompose}_q$, $\texttt{Power2Round}_q$, $\texttt{HighBits}_q$, $\texttt{LowBits}_q$, $\texttt{MakeHint}_q$, and $\texttt{UseHint}_q$. These components are also implemented individually in order to be used in various Dilithium algorithms. The components use the standard AXI interface and have been tested before their integration into the algorithms that are described in the following subsections.

3.2 Implementation of Key Generation Algorithm

In the `Key Generation` algorithm, the default sizes of input/outputs parameters are chosen by the specification of the Dilithium scheme - (recommended) security level III. The description of the individual input and output signals of the key generation algorithm component is given in Table 3. The outputs include the **`m_axis_rho_data`** signal which is a randomly generated part of both the private and public keys and is used to generate t_r and the $\mathbf{A}$ matrix. Furthermore, the **`m_axis_key_data`** signal, which is a part of the private key, is used in `ExpandMask`. The **`m_axis_tr_data`** signal is the output of the CRH function and it is a part of the private key. The coefficients **s_coeffs** correspond to the coefficients s_1 and s_2 (a part of the private key), which are used during generating the private key. The coefficients are allocated at the address **s_coeffs_addr**, where the first 1024

Table 3. The input and output signals of the key generation algorithm component

Signal	**Size [b]**	**Type**	**Description**
rst	1	Input	the reset of the component
clk	1	Input	clock signal
s_axis_data	128	Input	RNG seed
s_axis_valid	1	Input	the indication of valid seed
m_axis_rho_data	256	Output	parameter ρ
m_axis_key_data	256	Output	parameter K
m_axis_tr_data	384	Output	output t_r into the function **CRH**
s_coeffs	128	Output	coefficients s_1 and s_2
s_coeffs_addr	10	Input	the addresses of coefficients s_1 and s_2
t0_coeff	32	Output	the parameter t_0
t0_coeff_addr	11	Input	the address of the parameter t_0
t1_coeff	32	Output	the parameter t_1
t1_coeff_addr	11	Input	the address of the parametert_1
m_axis_valid	1	Output	the indication of the key generation

addresses correspond to the coefficients of the polynomial s_1 and the other 1280 addresses correspond to the coefficients of s_2. The t_0 coefficients are contained in the `t0_coeff` and `t1_coeff` signals, which are parts of the private key at `t0_coeff_addr`. Similarly, the `t1_coeff` signal, which is a part of the public key, is used to verify the key (UseHint function) and stores a total of 1280 addresses in the address space under the name `t1_coeff_addr`. The address space of the above-mentioned coefficients is formed by an IP block (IP - Intellectual Property by Xilinx) of Random Access Memory (RAM), which contains 1280 addresses.

3.3 Implementation of Signing Algorithm

The `Signing` algorithm can be considered the most complex part of the Dilithium scheme. It is mainly caused by the presence of a `while` loop. Thus, the number of repetitions can be only determined with certain probability.

The `Signing` algorithm component communicates with the environment via input and output AXI Stream interfaces that have a data bus width of 512 bits. At the input, the parameters of the private key are received in the first AXI Stream transactions, followed by the message itself to be signed. If the message is signed with a private key identical to the key for the previous signature, then it is possible to load only the message to the component because the private key is already stored in the component. This fact is indicated by the first empty AXI Stream transaction with the highest bit set to logical 1. In such case, the component can skip the initial operations related to the generation of the necessary signature parameters. After the signature is completed, the corresponding vectors are passed through the output interface. The assignments of input and output parameters to specific AXI Stream transactions are shown in Table 4. The size of one transaction corresponds to the width of the data bus (512 bits).

Table 4. Assignment of input and output to AXI stream transactions.

ID of transaction	**Data**
Assignment of input	
0	ρ (parameter for matrix **A** generation)
1	t_r (parameter for **CRH**)
2	K (parameter for **CRH**)
3...66	polynomials represented as the vector s_1
67...146	polynomials represented as the vector s_1
147...226	polynomials represented as the vector t_0
227...	message for signing
Transactions of output	
0...15	polynomial c
16...95	polynomials represented as the vector h
96...159	polynomials represented as the vector z

The `Signing` algorithm can be divided into 2 parts. The shorter `initial part` (Lines 1-3 in Algorithm 2) can be executed only once. The second longer part is executed inside the `while` loop (Lines 4-15 in Algorithm 2). See Section 2.3 for more details. In particular, matrix **A** generation and vectors s_1, s_2, t_0 conversion to NTT area can be executed once and then stored in BRAM memories, i.e. a stored private key is used.

The operations of the initial part are independent of each other, and therefore it is possible to parallelize them. Since the generation of the matrix and **CRH** with a long message are much more time consuming than the NTT conversion (to NTT area), the NTT conversion of vectors $\mathbf{s_1}$, $\mathbf{s_2}$, $\mathbf{t_0}$ is executed sequentially by using a single **NTT** component.

With the exception of generating the vector **y**, all operations of the `while` cycle depend on the results of previous operations. In order to be able to parallelize this part and use the potential of the FPGA, the individual parts were created to generate outputs for future iterations, regardless of whether this iteration will be performed. The block diagram of the signature implementation is shown in Figure 2. The signing algorithm is divided into 18 parallel running processes that are interconnected to each other. For example, the `while` cycle part performs 11 **NTT** components in parallel. This parallel approach requires more DSP blocks but provides a smaller number of cycles than the sequential approach.

3.4 Implementation of Verification Algorithm

The description of the individual input and output signals of the `Verification` algorithm is depicted in Table 5. The coefficients are passed to the component in blocks of 4 coefficients, so that bits 31 to 0 are reserved for coefficients 1 to 64, and bits 63 to 32 for coefficients 65 to 128, etc. To generate the matrix **A**

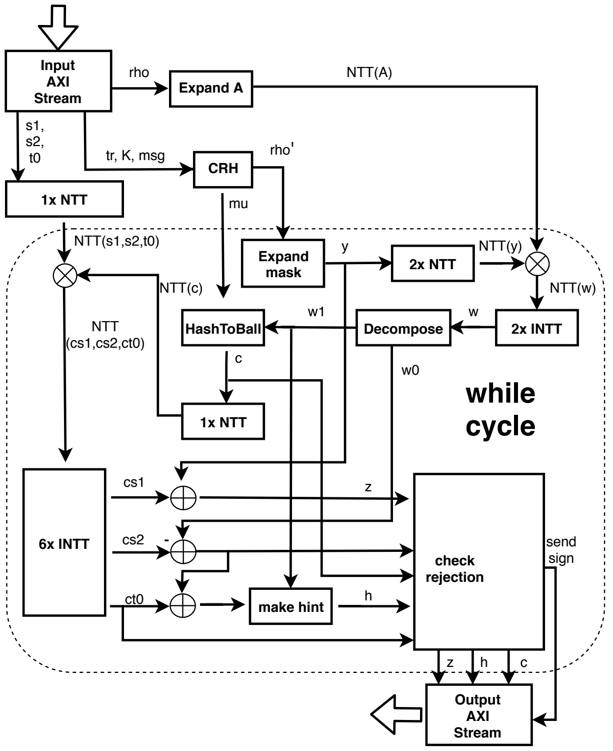

Fig. 2. Block diagram for the signing algorithm implementation.

of the ExpandA component, the verification algorithm uses the generation of coefficients by columns, because each row in a given column is multiplied by the same polynomial $\mathbf{z}$ and so this polynomial does not have to be stored or regenerated every time. The generated vector of polynomials $\mathbf{w}_1'$ is packed in a pair of coefficients (see the reference implementation [8]) and sent directly to the **Hashing to a ball** component in order to save time when accessing memory.

Control test-benches have been created to test all components. The correctness of the signing and verification components has been also checked by created test vectors from the C reference implementation [8].

4 Experimental Results and Comparison

This section describes the experimental results of our hardware implementation design for all Dilithium algorithms in all presented security levels (i.e, I., II., III. and IV variants). The target FPGA platform of our implementation is a chip from Xilinx, namely Virtex 7 UltraScale+ with the designation xcvu7p-flvb2104-

Table 5. Table of input and output signals of the verification algorithm component

Signal	Size [b]	Type	Description
rst	1	input	the reset of the conponent
clk	1	input	clock signal
s_axis_rho_tdata	256	input	ρ
s_axis_rho_tvalid	256	input	indication of valid ρ
s_axis_msg_tdata	1088	input	the part of message
s_axis_msg_tvalid	1	input	indication of the valid message
s_axis_msg_tuser	11	input	marking of the last byte of the message
s_axis_msg_tlast	1	input	indication of the last part of the message
s_axis_msg_tready	1	output	indication of the reading the message part
m_axis_tdata	1	output	indication of signature verification
m_axis_tvalid	1	output	indication of valid output
m_axis_tready	1	input	indication of the reading the output
Polynomial coefficients - equal for t_1, z, h, c			
s_axis_{X}_tdata	128	input	4 polynomial coefficients
s_axis_{X}_tvalid	1	input	indication of valid coefficient
s_axis_{X}_tready	1	output	indication of the reading the coefficient

2-i. Table 6 shows available hardware resources and specification of the Virtex 7 Ultrascale+ FPGA platform.

4.1 Required Hardware Resources and Performance Results on FPGA

The efficiency of hardware implementation can be measured by required hardware resources and performance (e.g., frequency, # number of cycles, Operations Per Second (OPS)) on the target platform (UltraScale+).

Table 7 shows the results after the synthesis of the individual components needed in `Key Generation`, `Signing` and `Verification` algorithms and of the aforementioned algorithms for security level III. These results were obtained by synthesis in Vivado 2017.4.1 for Virtex UltraScale +. The table depicts the number of used hardware sources (i.e., LUTs, FFs, and memory modules such as BRAM, LUTRAM) and also the theoretical operating frequency that can serve for the assessment of computational performance. Note that NTT and NTT^{-1} require DSP and BRAM due to their higher complexity. In particular, DSP blocks are used for optimization reasons, i.e., they allow to achieve higher

Table 6. Hardware specification of Virtex 7 Ultrascale+

LB	LUTs	FFs	RAM [MB]	UltraRAM [MB]
1724000	788000	1576000	50.6	180

Table 7. Hardware resources on FPGA for Dilithium components and algorithms (security level III.)

Component	LUT	FF	DSP	BRAM	LUTRAM	Frequency `[MHz]`	Consumption stat./dyn. `[mW]`
Dilithium Components							
SHAKE-128	3735	1608	0	0	0	587.2	1800/832
SHAKE-256	3361	1608	0	0	0	587.2	1792/816
Expand A	5003	3191	0	0	0	558.3	1659/626
CRH	4678	3085	0	0	0	456.8	1667/793
Sample in ball	7347	4418	0	0	0	435.4	1661/818
RNG	384	128	0	0	0	758.3	1022/36
Power2Round	12518	3085	0	0	1667	732.2	1385/185
NTT	1798	2532	48	3.5	438	637	1665/1214
NTT^{-1}	2547	3889	84	3.5	762	637	1668/1557
Dilithium Algorithms							
Key Generation	54183	25236	182	15	1808	350	1688/3578
Signing	68461	86295	965	145	8726	333	1800/13012
Verification	61738	34963	316	18	1922	158.2	1665/1251

frequency, and therefore a higher number of transformations per second. Then, BRAM blocks are needed to store NTT (and NTT^{-1}) input, intermediate and output values. In fact, this is reflected on algorithms' performance where `Signing` proportionally requires more NTT and NTT^{-1} computations with respect to `Key Generation` and `Verification`. However, the hardware resources of this scheme do not take up more than 1/4 of the total resources of the selected FPGA (UltraScale+) platform. The resource ratio is expressed as a percentage for LUT: 68461 (8.69%), LUTRAM: 8726 (2.21%), FF: 86295 (5.47%), BRAM: 145 (10.07%) and DSP: 965 (21.16%).

In Table 8, the performance results for various Dilithium variants (I., II., III., IV.) algorithms are shown. The size of a message is 59 B. For instance, our FPGA-based implementation of Dilithium for recommended version III. is able to sign up to 15832 messages per second and verify up to 10524 signatures per second. In particular, the number of generated polynomials increases with the security level. This has an impact on the clock cycles. On the contrary, the number of LUT, FF and LUTRAM is practically unchanged among the variants since $\mathcal{R}_q$ is the same for all security levels (see Section 2.3 for more details). For example, the `Signing` algorithm requires in average 66742, 69445, 70395, and 74395 LUTs for I., II., III., and IV. variants, respectively.

4.2 Comparison with HLS Implementations

As our hardware implementation is the first implementation written solely in VHDL, we can present only the comparison with the HLS-based implementa-

Table 8. Performance of Dilithium variants.

	Key generation		Signing		Verification	
Security Level	Cycles	Ops	Cycles	Ops	Cycles	Ops
I.	7990	43805	13110	25782	6770	23350
II.	12600	27778	18338	17723	10546	15000
III.	18193	19238	21033	15547	15032	10524
IV.	22981	15230	22362	14265	20221	7800

tions [3, 20]. Table 9 depicts the hardware resources and frequency for our VHDL and aforementioned HLS-based implementations. Basu *et al.* [3] HLS implementation has been performed on a Xilinx Artix-7 FPGA board. Basu *et al.* and do not provide `Key Generation` results. Their implementation of `Signing` takes 826832 cycles for 8.738 ns clock value. Their `Verification` takes 297592 cycles for 8.738 ns clock value. Soni *et al.* [20] HLS implementation of Dilithium has been performed on a Xilinx Artix-7 FPGA board. Their `Key Generation` takes 241102 cycles for 8.375 ns clock value. Their baseline `Signing` algorithm takes 1659851 cycles for 8.738 ns clock value. The optimized `Signing` by using loop unrolling takes 1565100 cycles for the same clock value. The `Verification` algorithm takes 292782 cycles for 8.738 ns clock value. After using loop unrolling the number of cycles is decreased to 242901 but with slightly higher clock value of 9.83 ns. It is to be noted that unrolling versions of both algorithms significantly increase the hardware resources to almost a double size. Soni *et al.* also present the loop pipelining optimization that reduces the latency while keeping the LUTs and FFs similar to the baseline version. This version requires 233420 cycles for key generation, 1618319 cycles for signing and 285100 cycles for verification. Therefore, the results of the pipelining optimization variant (as performance/resources trade-off) is used in our comparison in Table 9.

The results indicate that our VHDL-based implementation of all Dilithium algorithms requires less cycles than HLS-based implementations. Moreover, our VHDL implementation reaches higher frequencies than HLS-based implementations thus it provides more operations per second. This is achieved by using pipelined processing inside the logic blocks. Nevertheless, this performance optimization increases FFs.

Figure 3 compares the performance of our VHDL implementation with relevant HLS-based implementations, i.e., Soni *et al.* [20] and Basu *et al.* [3]. For each implementation, we compute the number of operations per second for each algorithm based on knowledge of the overall latency (cycles) and overall frequency. For instance, our implementation of the `Signing` algorithm is ca 226 times faster than the Soni *et al.* implementation [20] and ca 114 times faster than the Basu *et al.* implementation [3]. Our implementation of the `Verification` algorithm is then ca 26 times faster than Soni *et al.* implementation [20], and 27 times faster than Basu *et al.* implementation [3].

Table 9. Comparison of hardware resources and latency on FPGA for Dilithium implementations (sec.level III.). Not available parameters are marked as "-".

Implementation	**LUT**	**FF**	**Frequency** `[MHz]`	**Latency** `[Cycles]`
Key Generation				
Basu [3]	-	-	-	-
Soni [20]	86646	17674	119.4	233420
Our proposal	54183	25236	350	18193
Signing				
Basu [3]	123933	27308	114.4	826832
Soni [20]	90567	21160	114.4	1618319
Our proposal	81530	83926	333	21033
Verification				
Basu [3]	63980	14783	114.4	297592
Soni [20]	65274	15169	114.4	285100
Our proposal	61738	34963	158.2	15032

4.3 Comparison with Software Implementation

We also compare our hardware implementation with the reference Dilithium implementation written in C. The Software implementation was tested on Ubuntu VM (Ubuntu 18.04.3 LTS, 64-bit, CPU i5-7200 2.5 GHz, 1 GB). The results of the reference implementation (Dilithium-ref) for a random message of the length of 59 B, security version III., are as follows: (1) `Key generation` : 585217 cycles, 0.2251 ms, 4442 generation ops, `Signing`: 4181466 cycles, 1.608 ms, 622 signing ops, and (3) `Verification`: 676288 cycles, 0.2601 ms, 3844 verification ops.

VHDL-based implementations are usually more efficient than software implementations. Our `Signing` algorithm requiring 21033 cycles is able to create 15832 signatures per second (on FPGA with 333 MHz) while the C-implementation required 4181466 cycles and performs 622 signatures per second (on the CPU 2.5 GHz). Thus, our hardware-based implementation is 25 times faster than the software-based one.

5 Conclusion

In this work, we presented the fast hardware implementation of the Dilithium signature scheme. We implemented the components from scratch and optimized them to be efficient, i.e., achieve a low number of cycles and high frequency. The components are divided into three main Dilithium algorithms following the parallelization design approach, without using HLS. For instance, our signing algorithm runs in 18 parallel processes and a message can be signed within 21033 cycles and 333 MHz frequency (for the security level III.). The verification algorithm takes 15032 cycles within 158.2 MHz (for the security level III.). We

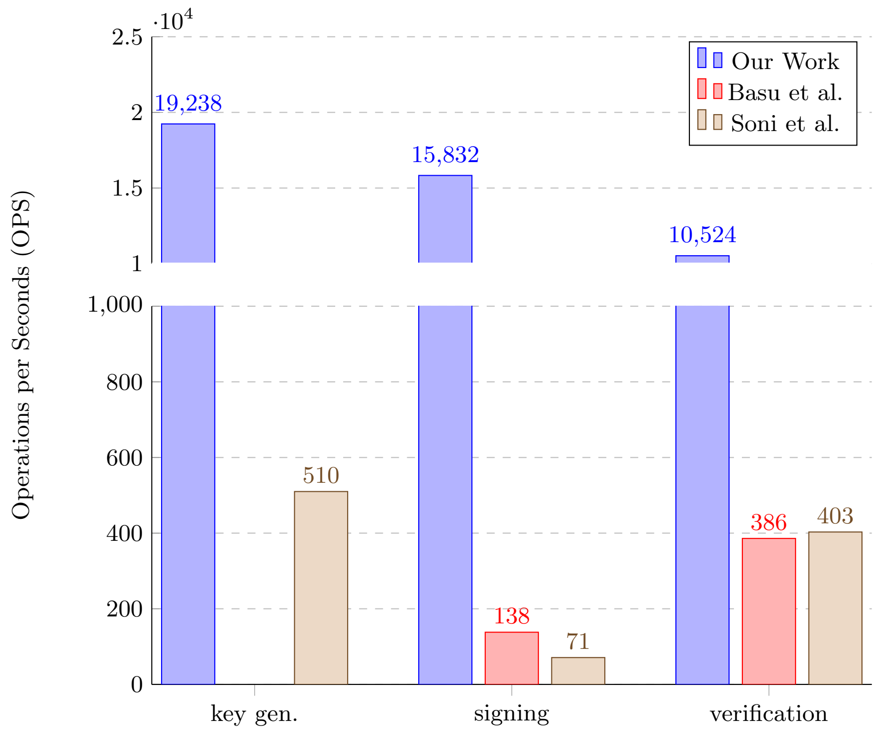

Fig. 3. Comparison of performance on FPGA for Dilithium implementations.

demonstrated that our VHDL-based implementation is much more efficient than existing HLS-based implementations (e.g. our signing is ca 114 times faster) and the software C-based reference implementation (e.g. our signing is ca 25 times faster). Our implementation of all Dilithium algorithms requires a reasonable amount of hardware resources (e.g. our signing takes 81350 LUTs and 83926 FFs) on the FPGA board (UltraScale+).

Our future work will be focused on the optimization of certain blocks such as CRH to increase the frequency of signing and verification algorithms. Further, we will investigate how to integrate and ensure resistance against side channel attacks, while still prioritizing high-performance and flexibility.

References

1. Nist - computer security resource center (csrc): Post-quantum cryptography - round 3 submissions. https://csrc.nist.gov/projects/post-quantum-cryptography/round-3-submissions, last accessed 09-November-2020.
2. Albrecht, M.R., Deo, A.: Large modulus ring-lwe $\geq$ module-lwe. In: International Conference on the Theory and Application of Cryptology and Information Security. pp. 267–296. Springer (2017)

3. Basu, K., Soni, D., Nabeel, M., Karri, R.: Nist post-quantum cryptography-a hardware evaluation study. IACR Cryptol. ePrint Arch. **2019**, 47 (2019)
4. Bauer, B., Wecker, D., Millis, A.J., Hastings, M.B., Troyer, M.: Hybrid quantum-classical approach to correlated materials. Physical Review X **6**(3), 031045 (2016)
5. Bernstein, D.J.: Introduction to post-quantum cryptography. In: Post-quantum cryptography, pp. 1–14. Springer (2009)
6. Brakerski, Z., Gentry, C., Vaikuntanathan, V.: (leveled) fully homomorphic encryption without bootstrapping. ACM Transactions on Computation Theory (TOCT) **6**(3), 1–36 (2014)
7. Chen, L., Chen, L., Jordan, S., Liu, Y.K., Moody, D., Peralta, R., Perlner, R., Smith-Tone, D.: Report on post-quantum cryptography, vol. 12. US Department of Commerce, National Institute of Standards and Technology (2016)
8. Ducas, L., Kiltz, E., Lepoint, T., Lyubashevsky, V., Schwabe, P., Seiler, G., Stehlé, D.: Crystals-dilithium: A lattice-based digital signature scheme. IACR Transactions on Cryptographic Hardware and Embedded Systems pp. 238–268 (2018)
9. Ferozpuri, A., Gaj, K.: High-speed fpga implementation of the nist round 1 rainbow signature scheme. In: 2018 International Conference on ReConFigurable Computing and FPGAs (ReConFig). pp. 1–8. IEEE (2018)
10. Lal, V.: Lfsr-random number generator. https://opencores.org/projects/lfsr_randgen (2010), last accessed 05-August-2016.
11. Langlois, A., Stehlé, D.: Worst-case to average-case reductions for module lattices. Designs, Codes and Cryptography **75**(3), 565–599 (2015)
12. Malina, L., Ricci, S., Dzurenda, P., Smekal, D., Hajny, J., Gerlich, T.: Towards practical deployment of post-quantum cryptography on constrained platforms and hardware-accelerated platforms. In: International Conference on Information Technology and Communications Security. pp. 109–124. Springer (2019)
13. Martín-López, E., Laing, A., Lawson, T., Alvarez, R., Zhou, X.Q., O'brien, J.L.: Experimental realization of shor's quantum factoring algorithm using qubit recycling. Nature Photonics **6**(11), 773 (2012)
14. Moses, T.: Quantum computing and cryptography. Entrust Inc. January (2009)
15. Nejatollahi, H., Dutt, N., Ray, S., Regazzoni, F., Banerjee, I., Cammarota, R.: Post-quantum lattice-based cryptography implementations: A survey. ACM Comput. Surv. **51**(6), 129:1–129:41 (Jan 2019). https://doi.org/10.1145/3292548, http://doi.acm.org.ezproxy.lib.vutbr.cz/10.1145/3292548
16. NIST: Fips pub 202 sha-3 standard: Permutation-based hash and extendable-output functions (2015), https://csrc.nist.gov/publications/detail/fips/202/final
17. PQCRYPTO-EU-project: Tu eindhoven leads multi-million euro project to protect data against quantum computers. https://pqcrypto.eu.org/press/press-release-post-quantum-cryptography-ENGLISH.docx (2016), last accessed 04-November-2018.
18. Roy, S.S., Basso, A.: High-speed instruction-set coprocessor for lattice-based key encapsulation mechanism: Saber in hardware. IACR Cryptol. ePrint Arch. **2020**, 434 (2020)
19. Soni, D., Basu, K., Nabeel, M., Aaraj, N., Manzano, M., Karri, R.: Hardware architectures for post-quantum digital signature schemes (2020)
20. Soni, D., Basu, K., Nabeel, M., Karri, R.: A hardware evaluation study of nist post-quantum cryptographic signature schemes. In: Second PQC Standardization Conference. NIST (2019)
21. Wang, W., Szefer, J., Niederhagen, R.: Fpga-based niederreiter cryptosystem using binary goppa codes. In: International Conference on Post-Quantum Cryptography. pp. 77–98. Springer (2018)