CoHA-NTT: A Configurable Hardware Accelerator for NTT-based Polynomial Multiplication

Kemal Derya, Ahmet Can Mert, Erdinç Öztürk, and Erkay Savaş, *Member, IEEE*

***Abstract*—In this paper, we introduce a configurable hardware architecture that can be used to generate unified and parametric NTT-based polynomial multipliers that support a wide range of parameters of lattice-based cryptographic schemes proposed for post-quantum cryptography. Both NTT and inverse NTT operations can be performed using the *unified* butterfly unit of our architecture, which constitutes the core building block in NTT operations. The multitude of this unit plays an essential role in achieving the performance goals of a specific application area or platform. To this end, the architecture takes the size of butterfly units as input and generates an efficient NTT-based polynomial multiplier hardware to achieve the desired throughput and area requirements. More specifically, the proposed hardware architecture provides *run-time* configurability for the scheme parameters and *compile-time* configurability for throughput and area requirements. This work presents *the first* architecture with both run-time and compile-time configurability for NTT-based polynomial multiplication operations to the best of our knowledge. The implementation results indicate that the advanced configurability has a negligible impact on the time and area of the proposed architecture and that its performance is on par with the state-of-the-art implementations in the literature, if not better. The proposed architecture comprises various sub-blocks such as modular multiplier and butterfly units, each of which can be of interest on its own for accelerating lattice-based cryptography. Thus, we provide the design rationale of each sub-block and compare it with those in the literature, including our earlier works in terms of configurability and performance.**

***Index Terms*—NTT, PQC, Polynomial Multiplication, Parametric, Hardware**

I. Introduction

The progress in quantum computer technologies has gained considerable momentum in recent years, which, in turn, calls attention once again to the vulnerability of current public cryptosystems to quantum computers. Lattice-based cryptography has emerged as one of the most promising cryptographic constructions for post-quantum cryptography (PQC) as it is based on a set of hard mathematical problems, which are conjectured to be resistant against attacks by quantum computers. In the NIST standardization process, which is in the third round as of writing, five out of the seven remaining candidates are lattice-based cryptographic schemes [1]. An efficient polynomial multiplier, where most of the execution time of the majority of lattice-based cryptosystems are expended, is conducive to practicable implementations of lattice-based cryptography [2].

K. Derya, A. C. Mert, E. Öztürk and E. Savaş are with the Faculty of Engineering and Natural Sciences, Sabanci University, 34956 Istanbul, Turkey. E-mail: {kemalderya, ahmetcanmert, erdinco, erkays}@sabanciuniv.edu

A. C. Mert is with the Institute of Applied Information Processing and Communications, Graz University of Technology, 8010 Graz, Austria. E-mail: ahmet.mert@iaik.tugraz.at

It is a significant challenge to implement polynomial multipliers efficiently for different platforms, from resource-constrained micro-controllers used in IoT applications to powerful data centers. IoT devices call for area-optimized and power-optimized designs with less emphasis on throughput, while data centers prefer throughput-optimized designs to meet aggressive timing requirements [3]. Therefore, it is essential to propose a design methodology that allows specifying area, power, and throughput as design parameters. This feature of the architecture is referred to as *compile-time configurability* (CTC).

Providing *run-time configurability* (RTC) is also an essential feature since it allows the hardware to be used in different PQC schemes without requiring its recompilation. In addition, the PQC schemes achieve different security levels by employing different parameters, which also supports the claim of configurability requirement. Thus, having a hardware implementation that supports various scheme parameters with *run-time configurability* will allow us to choose dynamically between different security levels of the same PQC scheme as well as different PQC schemes.

The number-theoretic transform (NTT) is a popular method utilized in the lattice-based cryptosystems to reduce the complexity of multiplication in the polynomial rings, $\boldsymbol{R}_q = \mathbb{Z}_q[x]/\phi(m)$, where the coefficient modulus q and the degree of the cyclotomic polynomial $\phi_m(x) = x^n + 1$, n, are referred to as the scheme parameters. The schoolbook polynomial multiplication method has the complexity of $O(n^2)$ and NTT reduces this complexity to $O(n \log n)$ [4]. When the polynomial multiplication in $\boldsymbol{R}_q$ is performed using NTT, the efficiency of the NTT operation determines the performance of the cryptosystem to a great extent. An NTT multiplier, which can work with various n and q values that appeared in various PQC schemes without recompilation, is referred to as RTC.

The core processing unit in an NTT multiplier is the butterfly circuit, which can take different forms such as Cooley-Tukey (CT) and Gentleman and Sande (GS) designs [5]. An NTT multiplier that features a butterfly unit (BU), which implements both CT and GS is referred to as a unified multiplier. It enables the implementation of NTT and inverse NTT (INTT) in the same circuit efficiently. An NTT-based multiplier that can be recompiled with a different number of BUs for throughput or time-area efficiency is referred to as CTC.

A. Related Works

There are multitudes of implementations for NTT and NTT-based polynomial multiplication operations of lattice-based

TABLE I
CONFIGURABILITY OF WORKS IN THE LITERATURE

Work	Platform	n	q	BU
[10]	Virtex-7	Fixed 256	Fixed 12-bit	Fixed
[11]	Artix-7	Fixed 256	Fixed 12-bit	Fixed
[12]	Virtex-7	Fixed 1024	Constant 32-bit	Fixed
[13]	RISC-V	RTC up to 1024	RTC up to 32-bit	Fixed
[14]	65 nm	RTC up to 1024	RTC up to 16-bit	Fixed
[15]	Virtex-7	RTC up to 4096	RTC up to 32-bit	Fixed
[16]	40 nm	RTC up to 2048	RTC up to 24-bit	Fixed
[17]	Artix-7	RTC up to 4096	RTC up to 39-bit	Fixed
[18]	Virtex-6	CTC up to 512	CTC up to 13-bit	Fixed
[19]	Artix-7	CTC up to 2048	CTC up to 30-bit	Fixed
[20]	Virtex-7	CTC up to 4096	CTC up to 60-bit	CTC
[21]	32 nm	CTC up to 4096	CTC up to 71-bit	CTC
This Work	Virtex-7	RTC 256 to 1024	RTC 12-bit to 30-bit	CTC

cryptosystems for various platforms with different area power and timing requirements. Those in [6], [7], [8], [9] are essentially software implementations, which can support the implementations of many lattice-based algorithms; therefore, they offer run-time configurability. Seiler [8] focused on optimizing the NTT operation for the AVX2 instruction set on Intel processors. Other works such as [6], [7], [9] proposed instruction set extensions for RISC-V architecture for hardware/software co-design. Since NTT is a highly parallelizable operation, involving many multiply-accumulate operations, CPU implementations reached their full potentials in terms of timing performance as they have a limited number of computational units, which does not facilitate further parallelization. This motivates research on parallel hardware implementations.

Some hardware architectures proposed in the literature support multiple scheme parameters for PQC, such as [13], [14], [16], [17], [18], [19]. However, their throughput and area are determined at design time, and thus they do not provide configurability thereof. There are also efforts for configurable NTT design using HLS [4], [22], [23]. However, HLS does not produce the most optimal design, leaving space for more optimizations of design parameters for RTL developers.

To make most of these optimization opportunities, we propose a novel architecture that can be used to obtain *run-time* and *compile-time* configurable (RTC and CTC, respectively) NTT-based polynomial multiplier architecture for NTT-friendly PQC schemes. Specifically, the proposed architecture provides RTC for scheme parameters (n and q) and CTC for area and performance (i.e., the number of BU).

Table I gives a summary of the current designs in the literature from the perspective of configurability. A parameter being fixed means that the specified work only supports a single parameter. If it is constant, it shows that the specified work supports a parameter in a constant range, i.e., constant $\lceil \log_2(q) \rceil$ bit-size. To the best of our knowledge, there are only two designs in the table, which propose NTT architectures providing compile-time configurability in terms of area and performance [20], [21]. While the architecture in [20] supports only the NTT operation, the one in [21] supports large scheme parameters typically used in homomorphic encryption applications. There is also another work [24] not appearing in Table I, which is exclusively designed for homomorphic encryption (supporting scheme parameters q from 109 to 438 bits and n from 4096 to 32768), which is therefore not comparable with our work as ours focuses on optimized architectures for relatively small parameters of lattice-based PQC schemes. Finally, none of the designs in [20], [21], [24] provides run-time configurability for scheme parameters.

B. Our Contribution

In this work, the proposed architecture can perform NTT, INTT, and NTT-based polynomial multiplication operations for NTT-friendly PQC schemes, namely CRYSTALS-KYBER (Kyber) with old and new parameters [25], [26], NewHope-512/1024 [27], CRYSTALS-DILITHIUM (Dilithium) [28], Falcon-I/II [29], and qTESLA-q-I [30]. Besides, the proposed multiplier can also be used for lattice-based schemes with ring degrees ranging from 256 to 1024 and NTT-friendly coefficients up to 30 bits. For instance, Fritzmann *et al.* [7] showed that NTT operations for SABER scheme [31] can be performed with three prime moduli. Thus, the proposed architecture is also suitable for the SABER PQC scheme without making any changes. Many algorithms benefit from the cryptographic agility offered by our architecture. Although some of the aforementioned schemes are not advanced to the final round in NIST's standardization process (i.e., NewHope) or their parameter sets are changed (i.e., Kyber (v1)), they or their variants still can be employed in different settings.

Furthermore, the architecture can be scaled to larger values of n so that it can also be utilized in various other lattice-based applications and schemes such as homomorphic encryption than PQC schemes proposed for NIST's competition. As homomorphic encryption applications are beyond the scope of this work, we include the implementations that are only relevant for PQC applications in this paper.

The proposed architecture can generate intermediate hard macroblocks in the hardware design process of lattice-based cryptosystems. Therefore, to provide developers of lattice-based cryptography with a better insight into the building blocks of our architecture, we analyze and compare each block with other similar proposals in the literature. We focus on the differences in each block and give a summary of their functionalities.

Specifically, our contribution in this paper is listed as follows:

- We introduce a configurable hardware architecture of NTT-based polynomial multiplier for lattice-based cryptography that is optimized for FPGA implementation.
- We propose a *run-time configurable word-level Montgomery modular multiplier unit* capable of multiplying up to 30-bit coefficients of the polynomials in $\boldsymbol{R}_q$ and

employing incomplete arithmetic [32]. This unit is configured at *run-time* to support modular multiplication operation on integers that have different bit lengths. Note that we use word-level Montgomery algorithm with NTT-friendly primes to decrease the complexity of the operation.
- We propose a *unified butterfly unit* that can perform CT and GS butterfly operations as the central processing element in our design. It is possible to configure this unit at *run-time* to be used for NTT, INTT and NTT-based polynomial multiplication operations.
- The number of butterfly units can be configured at compile-time to meet a particular design's throughput and area requirements. The CTC feature allows the architecture to avail itself to different applications that operate with a diverse set of requirements.
- We propose a *configurable memory control unit* that handles writing and reading coefficients to and from BRAMs of reconfigurable FPGA devices. The control unit ensures the order between the processing elements and the memory units. This unit dictates the state of the overall design so that it is possible to employ NTT, INTT, and NTT-based polynomial multiplication operations as desired. The configurability of this unit allows us to perform operations on polynomials with degrees ranging from 256 to 1024 for a different number of butterfly units.
- We provide a code sample that implements the proposed architecture via the GitHub repository (https://github.com/kemalderya/pqc-param-ntt). Note that there is a limited number of open-source hardware-based NTT designs in the literature. Our open-source hardware design can be beneficial for future studies in this field. Moreover, the researchers can easily integrate the NTT unit into their design and adjust its performance by just changing a single parameter.

The organization of the paper is as follows. Section II introduces background information. In Section III, the proposed hardware architecture is presented. Section IV presents the implementation results compared to prior works, and Section V concludes the paper.

II. Background

This section presents the notation used throughout the paper and the background on PQC and the NTT-based polynomial multiplication.

A. Notation

The ring $\mathbb{Z}_q$ consists of integers in $[0, q)$ and uses two operations on them: modular addition and modular multiplication, where q is the modulus. Also, $\phi_m(x)$ represents the cyclotomic polynomial, which is the unique irreducible polynomial. The cyclotomic polynomial has the form of $\phi_m(x) = x^n + 1$ where n is a power of two. Let $\boldsymbol{R}_q = \mathbb{Z}_q[x]/\phi_m(x)$ represent the polynomial ring reduced with $\phi_m(x)$ over $\mathbb{Z}_q$. Namely, it represents the polynomials with coefficients in $\mathbb{Z}_q$ and reduced with $\phi_m(x)$. Slightly abusing the terminology, we refer n as the degree of the ring.

Let lowercase letters (i.e., a) and boldface lowercase letters (i.e., $\boldsymbol{a}$ or $\boldsymbol{a}(x)$) represent integers and polynomials, respectively. Let $\overline{\boldsymbol{a}}$ represent the polynomial $\boldsymbol{a}$ in the NTT domain (i.e., $\overline{\boldsymbol{a}} = \text{NTT}(a)$). Let, finally, $\cdot$, $\times$ and $\odot$ represent the integer, polynomial, and the NTT-domain multiplication operations, respectively.

B. Lattice-based Cryptography

As the progress in the construction of quantum computers has accelerated in recent years, PQC has raised interest in academia as well as industry and different PQC schemes have been proposed based on cryptographic constructions such as lattice-based cryptography [28], [27], [30] and code-based cryptography [33]. Lattice-based cryptography schemes provide security even under worst-case scenarios, and they are claimed to be more efficient, simple, and parallelizable than other schemes [1].

Most of the lattice-based cryptosystems are based on the Learning with Errors (LWE) problem. There are different variants of the LWE problem that offer better performance such as Ring-LWE (R-LWE) used in NewHope [27] and Module-LWE (M-LWE) used in Kyber [25], [26]. The R-LWE problem is formulated using the equation in $\boldsymbol{R}_q$, $\boldsymbol{b} = \boldsymbol{a} \times \boldsymbol{s} + \boldsymbol{e} \pmod q$, where $\boldsymbol{a}$, $\boldsymbol{b} \in \boldsymbol{R}_q$ are public parameters, $\boldsymbol{s} \in \boldsymbol{R}_q$ is the secret key, and $\boldsymbol{e} \leftarrow \boldsymbol{D}_{0,\sigma}$ is the error polynomial, which is also the element of $\boldsymbol{R}_q$, whose coefficients are normally distributed with zero mean and (a small) σ standard deviation.

Two hard problems are given for R-LWE: (i) search problem intends to find the value $\boldsymbol{s}$ when the pair $(\boldsymbol{a}, \boldsymbol{b})$ is given and (ii) decision problem is to distinguish between the pair $(\boldsymbol{a}, \boldsymbol{b})$ and a random pair sampled from a uniform distribution over $\boldsymbol{R}_q$. M-LWE problem uses matrices of ring elements (i.e., polynomials in $\boldsymbol{R}_q$) instead of ring elements of R-LWE [31].

C. NTT-based Polynomial Multiplication

NTT is a discrete Fourier transform defined over $\boldsymbol{R}_q = \mathbb{Z}_q[x]/\phi_m(x)$. It facilitates fast convolutions over polynomials, which, in turn, makes polynomial multiplication more efficient. Let $\boldsymbol{a}(x) = \sum_{i=0}^{n-1} a_i \cdot x^i$ represent a polynomial over $\boldsymbol{R}_q$ with degree of $n-1$. Then, NTT of $\boldsymbol{a}(x)$ can also be represented in polynomial form, $\overline{\boldsymbol{a}}(x) = \sum_{i=0}^{n-1} \overline{a}_i \cdot x^i$ over $\boldsymbol{R}_q$ with degree of $n-1$, where coefficients $\overline{a}_i$ can be defined using Eqn. 1.

$$\overline{a}_i = \sum_{j=0}^{n-1} a_j \cdot \omega^{i \cdot j} \pmod q \text{ for } i = 0, 1, ..., n-1 \qquad (1)$$

As the coefficients in the NTT domain, $\overline{a}_i$ are computed over n values of the coefficients in the polynomial domain a_i, the computation is sometimes referred to as n-point NTT.

The NTT operation uses a primitive n-th root of unity constant called twiddle factor, $\omega \in \mathbb{Z}_q$ satisfying the conditions $\omega^n \equiv 1 \pmod q$, $\omega^i \neq 1 \pmod q$ $\forall i < n$ and $q \equiv 1 \pmod n$. The inverse NTT (INTT) operation is performed similarly except for using $\omega^{-1} \in \mathbb{Z}_q$ instead of ω and multiplying the resulting coefficients with n^{-1} in $\mathbb{Z}_q$ as the last step in the computation of INTT.

For random values of n, polynomial multiplication with NTT/INTT requires the input polynomials to be padded with 0 coefficients of n and a polynomial reduction operation to be performed to reduce the degree of the resulting polynomial to at most $n-1$ as the last step of the ring multiplication. When $\phi_m(x)$ has the form of x^n+1, however, *negative wrapped convolution* technique can be used, eliminating the need for padding the input polynomials and polynomial reduction. It requires input and output polynomials to be multiplied with the powers of ψ and ψ^{-1}, respectively, which are referred to as pre-processing and post-processing operations. Here, the constant ψ is a primitive $2n$-th root of unity satisfying the conditions $\psi^{2n} \equiv 1 \pmod q$, $\psi^i \neq 1 \pmod q$ $\forall i < 2n$ and $q \equiv 1 \pmod{2n}$.

Roy *et al.* proposed a method for merging the pre-processing and NTT operations, which requires utilizing CT butterfly structure for *merged* forward NTT operation [18]. Similarly, Pöppelmann *et al.* proposed a similar method merging INTT and post-processing operations, which requires utilizing GS butterfly structure for *merged* inverse NTT operation [34]. Thus, the pre-processing and post-processing steps can be eliminated from the computation by combining these approaches at the expense of using two different butterfly architectures. In our work, we employ this approach and performed NTT-based polynomial multiplication operation as shown in Eqn. 2, where $\texttt{NTT}_n$ and $\texttt{INTT}_n$ represent n-point (pt) merged NTT and INTT operations, respectively.

$$\boldsymbol{c} = \texttt{INTT}_n((\texttt{NTT}_n(\boldsymbol{a}) \odot \texttt{NTT}_n(\boldsymbol{b}))) \tag{2}$$

As shown in the subsequent sections, a unified design that allows both CT and GS formulation to be supported in the same unit is possible with minimum overhead in the hardware (e.g., area and latency).

Lyubashevsky *et al.* proposed a new method for NTT-based polynomial multiplication operation over $\boldsymbol{R}_q$, where the pre-processing and post-processing operations can be eliminated while satisfying only $q \equiv 1 \pmod n$ instead of $q \equiv 1 \pmod{2n}$ [26]. The Kyber scheme adopted this technique and optimized its parameters (i.e., reducing q from 7681 to 3329 and saving one bit from the integer operations of coefficients) and updated the NTT/INTT operation definition accordingly [35]. The Kyber scheme with old parameters and new parameters are referred to as Kyber (v1) and Kyber (v2), respectively. The NTT-domain multiplication of Kyber (v2) is also slightly different. In Kyber (v1), the NTT operation generates 256 degree-0 polynomials, and the coefficient-wise multiplication operation is performed with 256 modular multiplication operations. The NTT operation of Kyber (v2) outputs 128 degree-1 polynomials, and multiplication operation in the NTT domain is performed with 128 polynomial multiplications over $\mathbb{Z}_q[x]/(x^2-\omega^i)$ where i depends on the position of coefficients as shown in Algorithm 1. The unified merged NTT and merged INTT algorithms for Kyber (v2), and other NTT-friendly schemes are shown in Algorithm 2 and Algorithm 3, respectively, where $\texttt{br}(a,b)$ represents bit-reversal operation on b-bit integer a and fd determines the type of NTT/INTT operation (2 for Kyber (v2) and 1 for other schemes).

Algorithm 1 Algorithm of Multiplication in the NTT domain for Kyber (v2) [26]

Input: $\overline{\boldsymbol{a}}(x), \overline{\boldsymbol{b}}(x) \in \boldsymbol{R}_q$ in bit-reversed order
Input: $\omega \in \mathbb{Z}_q$
Output: $\overline{\boldsymbol{c}}(x) \in \boldsymbol{R}_q$ in bit-reversed order
1: **for** $(i = 0;\ i < 128;\ i++)$ **do**
2: $\quad a_0, a_1, b_0, b_1 \leftarrow \overline{\boldsymbol{a}}[2i], \overline{\boldsymbol{a}}[2i+1], \overline{\boldsymbol{b}}[2i], \overline{\boldsymbol{b}}[2i+1]$
3: $\quad \overline{\boldsymbol{c}}[2i] \leftarrow (a_0 \cdot b_1 + a_1 \cdot b_0) \bmod q$
4: $\quad \overline{\boldsymbol{c}}[2i+1] \leftarrow (a_1 \cdot b_1 \cdot \omega^{\texttt{br}(i,7)+1} + a_0 \cdot b_0) \bmod q$
5: **end for**
6: **return** $\overline{\boldsymbol{c}}$

Algorithm 2 Unified Forward NTT Algorithm

Require: $\boldsymbol{a}(x) \in \boldsymbol{R}_q$, in natural order
Require: $n, q, \omega \in \mathbb{Z}_q$ (or $\psi \in \mathbb{Z}_q$)
Require: $fd \in \{1, 2\}$ (final degree)
Ensure: $\overline{\boldsymbol{a}}(x) \in \boldsymbol{R}_q$, in bit-reversed order
1: $k, l, v = 1, (n/2), \log_2(n/fd)$
2: **while** $(l \geq fd)$ **do**
3: $\quad$ **for** $(s = 0;\ s < n;\ s = j + l)$ **do**
4: $\quad\quad w, k = \omega^{\texttt{br}(k,v)} \pmod q, k+1$
5: $\quad\quad$ **for** $(j = s;\ j < (s+l);\ j++)$ **do**
6: $\quad\quad\quad t = \boldsymbol{a}[j+l] \cdot w \pmod q$
7: $\quad\quad\quad \boldsymbol{a}[j+l] = \boldsymbol{a}[j] - t \pmod q$
8: $\quad\quad\quad \boldsymbol{a}[j] = \boldsymbol{a}[j] + t \pmod q$
9: $\quad\quad$ **end for**
10: $\quad$ **end for**
11: $\quad l = l/2$
12: **end while**
13: **return** $\boldsymbol{a}$

III. Proposed Work

In this section, the architectural details of the polynomial multiplier are given hierarchically, starting from the word-level Montgomery modular multiplier employing incomplete arithmetic. Then, we present the unified butterfly unit, which can perform both the GS and CT butterfly operations. Finally, we present the configurable memory control and the overall design. We also compare each building block with equivalent blocks proposed in our earlier works and the works in the literature.

A. Word-level Montgomery Modular Multiplier Unit

In this section, we first provide the details of our modular multiplier unit and then compare it with those in other works in the literature.

1) Our architecture: In our design, the modular multiplier unit consists of two parts: (i) a 32-bit integer multiplier (see Fig. 1) and (ii) a word-level Montgomery modular reduction unit (see Fig. 2) for NTT-friendly primes, which offers configurability for n and q parameters. The run-time configurable word-level Montgomery modular reduction unit utilizes the technique proposed in [12], employs incomplete arithmetic [32]. Our unit offers RTC for a wide range of scheme parameters and uses fixed word size. Therefore, it has less hardware complexity compared to the other run-time configurable word-level Montgomery modular multiplier

Algorithm 3 Unified Inverse NTT Algorithm

Require: $\overline{\boldsymbol{a}}(x) \in \boldsymbol{R}_q$, in bit-reversed order
Require: $n, q, \omega^{-1} \in \mathbb{Z}_q$ (or $\psi^{-1} \in \mathbb{Z}_q$)
Require: $fd \in \{1, 2\}$ (final degree)
Ensure: $\boldsymbol{a}(x) \in \boldsymbol{R}_q$, in natural order
1: $k, l, v = 0, fd, \log_2(n/fd)$
2: **while** $(l \geq (n/2))$ **do**
3: **for** $(s = 0;\ s < n;\ s = j + l)$ **do**
4: $w, k = \omega^{\mathrm{br}(k,v)+1} \pmod q, k + 1$
5: **for** $(j = s;\ j < (s + l);\ j + +)$ **do**
6: $\overline{\boldsymbol{a}}[j + l] = \overline{\boldsymbol{a}}[j] + \overline{\boldsymbol{a}}[j + l] \pmod q$
7: $\overline{\boldsymbol{a}}[j] = \overline{\boldsymbol{a}}[j] - \overline{\boldsymbol{a}}[j + l] \pmod q$
8: $\overline{\boldsymbol{a}}[j + l] = \overline{\boldsymbol{a}}[j + l] \cdot w \pmod q$
9: **end for**
10: **end for**
11: $l = 2 \cdot l$
12: **end while**
13: **for** $(i = 0;\ i < n;\ i + +)$ **do**
14: $\overline{\boldsymbol{a}}[i] = \overline{\boldsymbol{a}}[i] \cdot n^{-1} \pmod q$
15: **end for**
16: **return** $\overline{\boldsymbol{a}}$

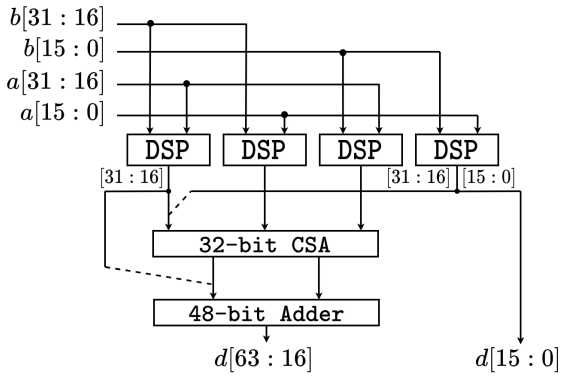

Fig. 1. 32-bit Integer Multiplier Unit

architectures in the literature [15]. In particular, the modular multiplier unit uses 8 DSP blocks and it can have up to five clock cycles latency, depending on the modulus size.

As shown in Fig. 1, the 32-bit integer multiplier unit multiplies the coefficients of the input polynomials, which can be up to 32 bits, i.e., it computes $d = a \cdot b$, where $a, b < 2^{32}$ and $d < 2^{64}$. In the first step, 32-bit inputs are divided into two parts, including the upper and lower 16-bit parts of the inputs. Then, 16-bit values are multiplied using four DSP blocks in FPGA. The results are registered and added using a 32-bit carry-save adder. The upper and lower 16-bit parts from DSP blocks are appended to the result of the carry-save adder to generate the multiplication result. Overall, the 32-bit integer multiplier unit has two clock cycle latency.

The word-level Montgomery reduction unit, depicted in Fig. 2, supports different parameters, and it is run-time configurable. The word-level Montgomery reduction divides reduction operation into smaller parts, and it leverages the special form of the NTT-friendly primes with *negative wrapped convolution* as shown in Eqns. 3-4, where w represents the word size for reduction operation, as proposed in [12].

$$q \equiv 1 \pmod{2n} \tag{3}$$

Algorithm 4 Word-Level Montgomery Reduction Algorithm

Input: $d = a \cdot b$ ($2K$-bit integer)
Input: $w = 8$ (word size)
Input: $L = \lceil \frac{K+2}{w} \rceil$ (iteration count)
Input: q (K-bit integer, $q = q_H \cdot 2^w + 1$)
Output: $c = d \cdot R^{-1} \pmod q$, $R = 2^{w \cdot L} \pmod q$
1: $T = d$
2: **for** $(i = 0;\ i < L;\ i + +)$ **do**
3: $T1_H = T \gg w$
4: $T1_L = T \pmod{2^w}$
5: $T2 = -T1_L \pmod{2^w}$
6: $C_{in} = T2[w - 1] \ \vee \ T1_L[w - 1]$
7: $T = T1_H + (q_H \cdot T2[w - 1 : 0]) + C_{in}$
8: **end for**
9: $c = T$
10: **return** c

$$q = q_H \cdot 2^w + 1 \text{ where } w = \log_2(2n) \tag{4}$$

For $k = \lceil \log_2(q) \rceil$, regular Montgomery reduction algorithm takes $d = a \cdot b$ as input and calculates $c = d \cdot R^{-1} \pmod q$ as shown in Eqn. 5, where $R = 2^k$ and $q' = q^{-1} \pmod R$. The proposed technique in [12] divides the operation in Eqn. 5 into w-sized parts and redefines R for w-sized operations as $R' = 2^w$. Thus, $q' = q^{-1} \pmod{2^w}$ becomes -1 when word size is selected as $w \leq \log_2(2n)$, which replaces $d \cdot q' \pmod R$ operation with simple 2's complement. Finally, w-sized reduction operation should be iterated for $L = \lceil \frac{k}{w} \rceil$ times for reducing $2k$-bit input d to k-bit output $d \cdot R^{-1} \pmod q$ where R is now $2^{w \cdot L}$.

$$c = \frac{d + q \cdot (d \cdot q' \pmod R)}{R} \tag{5}$$

Selection of word size (w) significantly affects the performance of modular reduction implementation since it determines the iteration count for w-sized operation (i.e., larger w is favored for a fewer number of iterations). We target supporting a wide range of parameters, and our architecture supports the values of n ranging from 256 to 1024. Schemes with different values of n can work with different word sizes, which require slightly different computations. The naive solution would be designing a modular reduction unit supporting multiple w values for each n parameter as proposed in [15]. However, this would require extra logic for configurability and increase the hardware complexity. Therefore, in our architecture, we select a fixed word size. Since we support Kyber (v2), which works with $n = 256$ and $q = 13 \cdot 2^8 + 1$, as the scheme with the smallest parameters, we select w as 8. The word-level Montgomery reduction algorithm is shown in Algorithm 4.

As shown in steps 5-7 of Algorithm 4, each w-sized reduction step can be implemented using one 2's complement unit, one OR gate, and one DSP block in FPGA performing multiply-and-accumulate operation $x \cdot y + z + cin$. Therefore, in this work, we used this approach for implementing one w-sized reduction step. A complete reduction operation requires L smaller w-sized reduction steps, which also determine the number of DSP blocks in the design.

The Montgomery reduction operation requires an extra subtraction operation at the end to bring the result back to the in-

terval $[0, q)$ [12]. Therefore, in this work, we utilize incomplete arithmetic in the Montgomery reduction operation so that the subtraction operation at the end of the Montgomery reduction algorithm is eliminated. In regular modular arithmetic, we always work with integers in the range $[0, q)$, and whenever an intermediate value becomes larger than or equal to the modulus q, a modular reduction operation is applied to bring it back in the interval $[0, q)$. A method called incomplete arithmetic [32] avoids the reduction operation whenever possible to perform word-level operations on the integers and eliminates the bit-level operations, which slow down the arithmetic. Therefore, instead of working in $[0, q)$, our modular arithmetic units can work with the integers in the range $[0, 2^{k+1})$.

In our design, inputs of the Montgomery reduction operation (a and b) should be in the range $[0, 2^{k+1})$. The output (c) should also stay in the same range for Montgomery reduction unit to work correctly. Thus, no extra reduction or subtraction will be required at the end of the Montgomery reduction operation. This requires R to be at least 2^{k+2}. When we substitute these values into Eqn. 5, we show that output also stays in the range $[0, 2^{k+1})$ as shown in Eqn. 6-7. Since we work with word-level Montgomery reduction, we define R as $2^{w \cdot L}$ where the iteration count L is now defined as $\lceil \frac{k+2}{w} \rceil$ as shown in Algorithm 4.

$$c < \frac{d + q \cdot (d \cdot q' \pmod R)}{R} = \frac{2^{2k+2} + 2^k \cdot 2^{k+2}}{2^{k+2}} \quad (6)$$

$$c < \frac{2^{2k+2} + 2^{2k+2}}{2^{k+2}} = \frac{2^{2k+3}}{2^{k+2}} = 2^{k+1} \quad (7)$$

Since we work with fixed word size ($w = 8$), the maximum supported size of coefficient modulus (k) determines the iteration count. For an iteration count of 3, we can support at most $k = 3 \cdot 8 - 2 = 22$, which does not support Dilithium and qTESLA-q-I schemes. Therefore, we select the iteration count as $L = 4$, enabling coefficient modulus size up to $k = 4 \cdot 8 - 2 = 30$. In the proposed hardware, we used four DSP blocks in our modular reduction unit. When a parameter set requiring an iteration count less than four (i.e., $k = 12$), extra DSP blocks are not used, and proper output is selected using output multiplexer. The proposed run-time configurable word-level Montgomery reduction hardware is shown in Fig. 2, where primitive output registers of DSP blocks are used for improving the critical path. The proposed modular reduction unit uses four DSP blocks, and its latency is three clock cycles.

The result of Algorithm 4 has an extra multiple R^{-1}, which needs to be eliminated by either multiplying the result or one of the inputs by R. In this work, we used the latter approach and multiplied constants ω/ω^{-1} and ψ/ψ^{-1} by R=$2^{w \cdot L}$ before loading to the hardware. For a better understanding of the word-level Montgomery reduction algorithm, we provide a Python script generating the steps of algorithm for different parameters using sample input data in GitHub repository of our work.

2) Comparison with other works: There are mainly three approaches for implementing efficient modular reduction units in hardware: (i) shift and add method, (ii) Montgomery reduction and (iii) Barrett reduction. The shift and add method

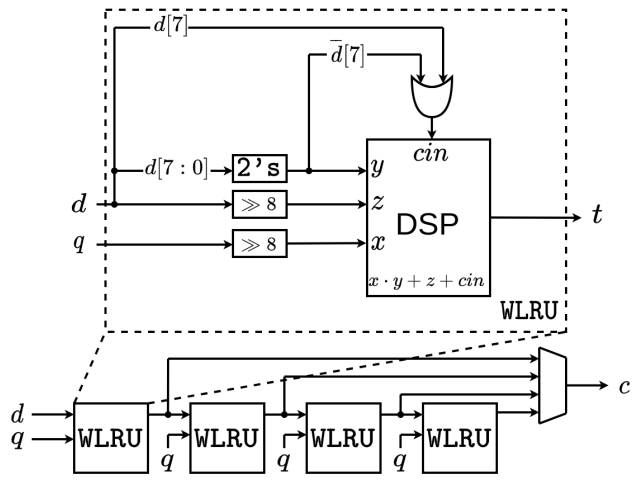

Fig. 2. Reconfigurable Word-level Montgomery Reduction Unit

is suitable for q with special form [36], while Montgomery and Barrett reduction techniques can work efficiently with arbitrary q values. There are also table-based modular reduction techniques that store the pre-computed reduced values in a table and use a sliding window for performing reduction operation step by step [37], [38]. However, these implementations mainly target applications working with very large values of q. Thus, they are not included for comparison.

The modular reduction unit in [10] supports a modular reduction for fixed $q = 3329$, which is used by Kyber (v2) scheme. It utilizes a similar approach by Zhang *et al.* [39], and it does not need to convert coefficients from Montgomery domain. Since it is optimized for a fixed modulus; it does not employ any multiplier unit for modular reduction operation. However, it lacks configurability, and it does not have the capabilities of our modular reduction unit. Similarly, the modular reduction unit in [11] only supports a fixed value $q = 3329$. It uses the Barrett reduction, but it does not have any configurability options as our modular reduction unit. Moreover, the modular reduction unit in [40] uses the Barrett reduction for q values up to 16-bit. Even though it offers RTC, it does not offer an operating window as large as what our design offers for scheme parameters.

The work in [12] utilizes the word-level Montgomery reduction algorithm with a fixed word size of 11. It employs three DSP units, supports the range $22 < \lceil \log_2(q) \rceil \leq 33$, and offers no configurability. More specifically, the modular reduction unit in [12] does not provide RTC for q, unlike our modular reduction unit offers.

The modular reduction unit in [20] also uses the word-level Montgomery reduction technique, and it provides CTC in which n, k, and w are fixed at the design time. Thus, it uses a fixed number of DSP blocks and offers no run-time flexibility for scheme parameters after the compilation of the design. It would use fewer area resources than our modular reduction unit since it is optimized to work with a single parameter at the run-time. Therefore, it does not have the configurability capabilities that our work offers. Furthermore, Wang *et al.* [19] use the regular Montgomery reduction algorithm for a coefficient modulus with constant

bit size. Their design offers CTC for a set of parameters. The complexity of their modular multiplier unit is low compared to our unit; however, it lacks the run-time configurability options.

The modular multiplier hardware in [15] has RTC over the scheme parameters. It uses the word-level Montgomery reduction technique, and it chooses w as shown in Eqn. 3-4. It has additional control hardware to change word size (w) at run-time; therefore, it occupies more area than our modular reduction unit although it provides similar level of configurability. Additionally, it employs a fixed number of DSPs, so it is only possible to configure it within a specific range of coefficient sizes once it is compiled. The work in [16] uses the Barrett algorithm for modular reduction. It offers RTC for q by using control signals. The complexity of the unit in [16] is high because it utilizes a different reduction unit for each value of q; therefore, the occupied area increases.

Overall, the modular multiplier unit can be designed adopting different approaches as observed in the literature. It is, also, expected that a reconfigurable unit increases the hardware area to enhance its capabilities. However, this increase should be negligible (if at all) as shown in our design.

B. Unified Butterfly Unit

1) Our architecture: The proposed unified butterfly unit (BU) performs the butterfly operations for NTT/INTT. This unit can be adjusted to perform CT or GS butterfly configurations. The CT configuration is used for NTT operation, while the GS configuration for INTT operation. The proposed butterfly unit uses one modular adder, one modular subtractor, and one word-level Montgomery modular multiplier, as shown in Fig. 3. The control input ct configures the output values, $even$ and odd, in run-time, depending on the operation, namely NTT or INTT. For the NTT operation, the output values $even$ and odd are computed as $a + b \cdot w \pmod q$ and $a - b \cdot w \pmod q$, respectively. For the INTT operation, the output values become $a + b \pmod q$ and $(a - b) \cdot w \pmod q$. The input w is set as a power of ω/ω^{-1} and ψ/ψ^{-1} for Kyber (v2) and other schemes, respectively, for NTT and INTT operations.

The word-level Montgomery modular multiplier unit can have latency between 2 and 5 clock cycles based on the selected parameters. The latency depends on the iteration count L. Therefore, $even$ and odd outputs need to be synchronized at the output. In our work, we used extra registers for generating extra delay for synchronization (shown as `2cc`, `3cc`, `4cc`, `5cc` in Fig. 3). The control signal i, determined by the word-level Montgomery reduction unit's iteration count, selects a delay path based on the parameters.

Polynomial multiplication requires coefficient-wise multiplication as shown in Eqn. 2, and it is possible to configure butterfly units to perform coefficient-wise multiplication. The butterfly unit with the GS configuration calculates $(a - b) \cdot w \pmod q$ as odd output. This output can be configured to perform $a \cdot w \pmod q$ by setting the input b as zero and a, w as input operands. In this work, we utilized this approach for performing coefficient-wise multiplication.

Kyber(v2) uses different coefficient-wise multiplication that requires multiply-accumulate operations, as shown in Algorithm 1. The proposed butterfly unit has add and sub outputs used for reading the results of modular addition and subtraction operations, respectively. These outputs are used for performing modular arithmetic operations as shown in Alg. 1. For Kyber(v2), the multiplication operation in the NTT domain first takes two degree-1 polynomials and performs multiplication for the coefficients of those polynomials. In the proposed hardware, this is performed using the butterfly unit in the GS configuration as already explained. These operations generate $a_0 \cdot b_0$, $a_0 \cdot b_1$ $a_1 \cdot b_0$ and $a_1 \cdot b_1$ intermediate values. Then, a modular adder is used to perform Step 3 of Alg. 1. Finally, the butterfly unit is used with the CT configuration to perform Step 4 for the $a + b \cdot w \pmod q$ operation. In this configuration, a is set as $a_1 \cdot b_1$, w is set as the appropriate power of the twiddle factor and b is set as $a_0 \cdot b_0$. We utilize this approach to perform the NTT domain multiplication in the Kyber(v2) scheme.

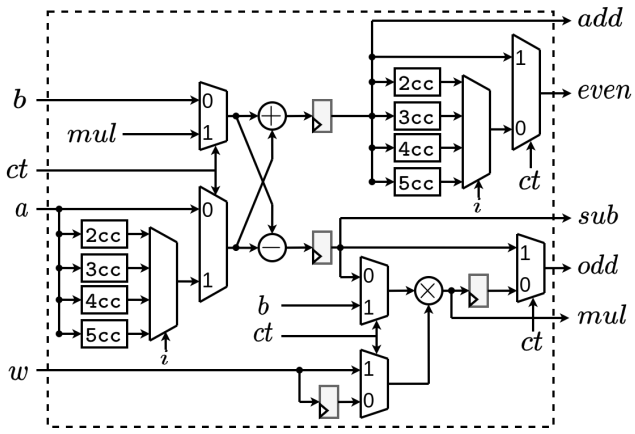

Fig. 3. Unified Butterfly Unit

2) Comparison with other works: There are mainly three approaches for implementing the butterfly operation in the hardware efficiently: (i) only the GS configuration [12], [15], [20], (ii) only the CT configuration [18] and (iii) unified GS and CT configurations [10], [11], [16], [19], [40]. The architectures using only GS or only CT configurations require pre-processing and post-processing operations during the NTT-based polynomial multiplication operation. For polynomial degree n, this requires extra $2n$ modular multiplication operations. As explained in Section II-C, the architectures with a unified GS-CT butterfly unit can eliminate pre-processing and post-processing operations by employing *merged* NTT and INTT operations. However, those designs need more control signals to configure for both operations; thus, these architectures have higher complexity and area than those with only GS and only CT configurations.

The butterfly unit in [10] utilizes a unified GS-CT configuration approach to perform butterfly operations. It can perform NTT, INTT, and coefficient-wise multiplication operations by changing control signals. However, it only works for Kyber (v2) scheme. It uses a technique by Zhang *et al.* [39] to eliminate multiplying coefficients by $n^{-1} \pmod q$ after INTT operation. Similarly, the butterfly unit in [11] follows the same approach for butterfly operations. These units offer Kyber (v2) scheme optimizations and do not support any other scheme due

to the fixed data length of the used arithmetic units.

The work in [12] uses the GS configuration for butterfly operations. The number of modular arithmetic units is the same as in our design. We gain configurability options by using more control hardware besides having the same number of modular arithmetic units. The butterfly unit in [20] uses the same approach for the implementation of the butterfly unit. The bit size of the supported coefficient modulus is determined at compile-time. Therefore it is not possible to change coefficient modulus at run-time. It does not have the configurability options that our BU offers. Furthermore, the design in [15] utilizes the same approach in the butterfly unit. Even and odd outputs have seven clock cycle latency in the unit in [15]; therefore, the latency stays the same with different scheme parameters where our design offers different delay paths. We use fewer clock cycles for the butterfly operation while offering more configurability options. Moreover, the design in [18] uses the CT configuration for butterfly operations. It has less complexity than our design. However, it lacks the configurability options offered by our design.

The design in [19] uses a unified butterfly unit with CT and GS configurations for butterfly operations. They utilize both configuration hardware into a unified design to avoid having two different hardware for two configurations. It fixes the coefficient modulus at compile-time; therefore, it does not offer RTC for scheme parameters. Furthermore, the work in [16] uses the same approach for butterfly operations. Our design uses one modular adder and subtraction unit, whereas the unit in [16] has one more modular adder and modular subtraction units. Similarly, the butterfly unit in [40] uses a similar implementation with one more modular adder and subtractor units. We utilize the same approach by having less hardware resources compared to those units. Overall, polynomial multiplication operations are performed with different design approaches, and we see that the complexity of hardware increases as it becomes configurable.

C. Configurable Memory Control and Overall Design

1) Our architecture: NTT/INTT algorithms shown in Algorithms 2-3 operate on loop structures. It is possible to unroll these loops and parallelize each operation. We can adjust the throughput and parallelization by changing the number of BUs. In our work, the number of butterfly units can be adjusted at compile-time, where it needs to be power-of-two and can be set as $n/2$ at maximum. Then, the proposed work generates hardware with the desired number of butterfly units and other necessary building blocks. For each butterfly unit, there are two BRAMs (BRAM 0 and BRAM 1 in Fig. 4) for storing input coefficients and one BRAM (BRAM TW in Fig. 4) for storing precomputed powers of twiddle factors (or primitive $2n$-th root of unity), as shown in Fig. 4.

Our design also has one compile-time parameter control unit that generates the necessary control signals, read and write address values for BRAMs based on the given parameters and the number of butterfly units. The address values are generated according to the ring size n. The depth of BRAMs

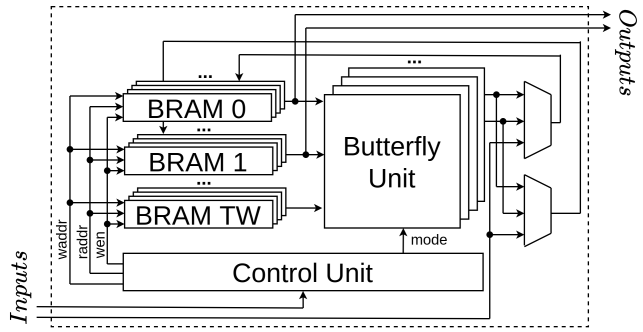

Fig. 4. Overall Design

storing input coefficients and the powers of twiddle factors are determined as $512/\texttt{BU}$ and $\sum_{i=0}^{i=9} 2^{\min(512/\texttt{BU},i)}$, respectively, where $\texttt{BU}$ represents the number of butterfly units. The data size of each BRAM address is set as 32-bit.

NTT (and INTT) algorithm consists of $\log_2(n)-1$ and $\log_2(n)$ stages for Kyber (v2) and other schemes, respectively, where each stage requires $n/2$ butterfly operations. An NTT operation can be implemented using a *divide-and-conquer* approach, where one NTT operation is divided into smaller NTT operations after each stage. After the first stage of n-pt NTT, the resulting coefficients can be processed using two $n/2$-pt NTT operations. This property enables an efficient addressing scheme that supports multiple n values. After each stage, the size of NTT operation is halved, and this approach is performed recursively. Therefore, an n-pt NTT operation can use the control logic of $n/2$-pt NTT twice after the first stage.

In the butterfly unit, an extra register is inserted for *odd* output. This is used to write two consecutive output values into the same BRAM block in two consecutive clock cycles as required by the NTT algorithm. Moreover, the INTT operation starts with smaller INTT operations and merges into larger INTT operations recursively. The address pattern of INTT is the same as NTT in reverse order.

The control unit generates two types of address values for memory units during the first $\log_2(n)-\log_2(\texttt{BU})-1$ NTT stages. It switces back and forth between these two types in consecutive clock cycles. The first type address value starts with 0 and second type address value starts with $n/(2^{\log_2(\texttt{BU})+s+2})$ where s is the NTT current stage. The control unit increments both types of address values by one in every two clock cycles; thus, the difference between them stays the same. The control unit generates address values until the second type address value becomes $n/(2{\cdot}\texttt{BU})-1$. For other NTT stages, there is only one type of address value generated. It starts with 0, and it increments by one for every clock cycle until it becomes $n/(2\cdot\texttt{BU})-1$.

A design with n as 16 and two BUs has a memory access pattern as shown in Fig. 5. In this scenario, the NTT operation starts with coefficients pairs of 0^{th}-8^{th} and 1^{st}-9^{th}. The coefficients are loaded into BRAMs as suitable to the coefficient pairs before the NTT operation starts. In the first two stages, the pattern shows a data dependency for BRAMs where the coefficients of BU operations need to be written on the same BRAM. During the first two stages, coefficients

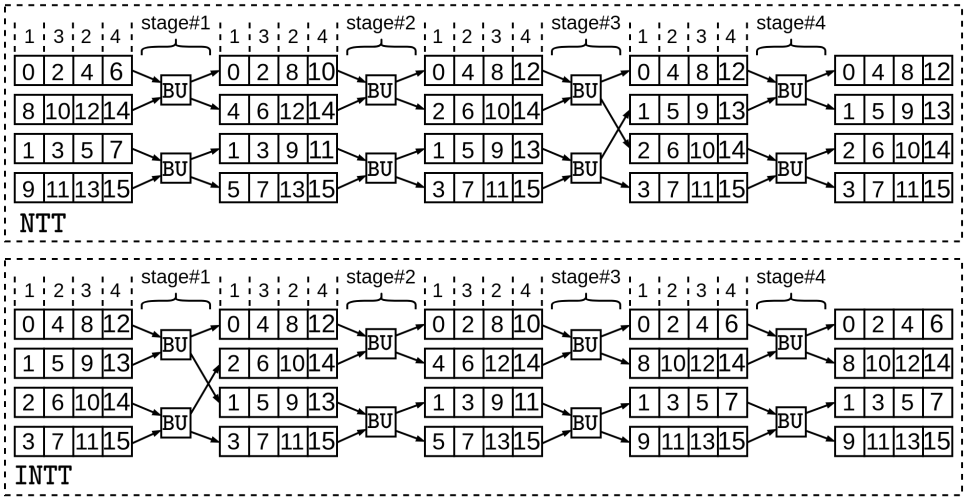

Fig. 5. Memory Access Pattern for $n = 16$ with two Butterfly Units

are read in an alternating order to overcome this problem, i.e. 0,2,1,3 instead of 0,1,2,3. We use an extra register with the *odd* output in BU to write the coefficients into the same BRAM in consecutive clock cycles. After the first two stages, the data dependency disappears, and the memory read pattern is made by incrementing the address after each BU operation, i.e., 0,1,2,3. After each operation, some coefficients need to be stored in different BRAM to be used in different BU. We use control signals to reorder the coefficients into suitable BRAMs when needed. The memory pattern for the INTT operation is the reverse of the NTT operation.

The control unit also produces address values for the coefficient-wise multiplication operation. The coefficients of a polynomial are stored in BRAMs in the address space from 0 to $512/\mathtt{BU}$. The control unit generates address values in this range and necessary control signals for the butterfly unit. Then, the butterfly unit performs the coefficient-wise multiplication operation with the GS configuration as a, w and b inputs take the first operand, the second operand, and the value of zero, respectively. Coefficient-wise multiplication of the Kyber (v2) scheme needs an address scheduling slightly different than the regular coefficient-wise multiplication operation. As shown in Steps 3-4 of Algorithm 1, it first requires modular multiplication and then modular addition. In our design, we first perform the necessary modular multiplication operation and store intermediate results in BRAMs. Then, the butterfly unit is used to perform Step 3 and Step 4 of Algorithm 1. The proposed design works in constant time for a given parameter set.

2) Comparison with other works: NTT-based polynomial multipliers are generally implemented in two ways: (i) an independent hardware accelerator [10], [12], [15], [20] and (ii) a sub-block unit in a cryptographic hardware [11], [16], [19], [18], [40]. Independent hardware accelerators utilize high-performance features of FPGAs. Even though they show better performance than those implemented as sub-blocks typically in low-constrained devices, they need a host processor to operate. Moreover, sub-block NTT multipliers are implemented on low-constrained devices to lower the energy consumption; thus, they prioritize low power over high performance.

The design in [10] introduces a hardware accelerator for NTT-based polynomial multiplication in the Kyber (v2) scheme. It follows a similar overall design concept with our design. It uses BRAMs to store input polynomials, and it utilizes a control unit to perform operations with a recursive NTT approach. The control unit demonstrates a similar memory access pattern to our design, and it is implemented to work with the Kyber (v2) scheme; thus, it does not support any other scheme. It uses Algorithm 1 for coefficient-wise multiplication with pipeline optimizations. We gain configurability capacity to accommodate other scheme parameters by using a slightly more resources while showing similar performance for the Kyber (v2) scheme compared to the design in [10] . Moreover, we offer CTC for design parameters to tune the throughput rate. The work in [12] follows the same approach for the implementation of the overall design. It has a fixed number of BRAMs and BUs; thus, it does not offer CTC for design parameters. The overall design follows a similar memory access pattern to perform operations while having no support for the Kyber (v2) scheme. We have more configurabililty options by having a slightly more complicated hardware.

The overall design in [15] introduces a hardware accelerator with a similar design concept. The work uses a fixed number of BRAMs and BUs, and it does not offer CTC options to adjust the parallelism. Its control unit offers support for different scheme parameters. The unit in [15] produces control signals according to the chosen parameter set. Even though the control unit offers RTC for scheme parameters, it uses many complex multiplexers. The control signals are chosen using multiplexers based on the current NTT stage. Our control unit produces control signals by using much less hardware that includes arithmetic operations and small multiplexers. Our design utilizes the configurable control unit much more efficiently. Furthermore, the design in [15] does not support the Kyber (v2) scheme. We show a similar performance with chosen parameter sets by having more configurability options. The work in [20] shows a similar approach for the overall design. The numbers of BRAMs and BUs are determined at

compile-time; therefore, the design [20] can be configured to adjust the throughput rate. The control unit is compiled for a specific scheme parameter, and it follows a similar memory access pattern. It is not possible to use its control unit for different scheme parameters on run-time. We have more configurability options by using a slightly more resources compared to the design in [20].

The design in [11] implements Kyber (v2) cryptographic scheme, using NTT structure to perform operations. NTT multiplier is implemented as a sub-block, and it utilizes 2 BUs with 2 RAM banks. The width of RAM banks is arranged as the width of two coefficients pair. Coefficients pairs are fetched from RAM banks to BUs, and the results are stored back. The overall design in that work does not offer RTC for different scheme parameters, and it is not possible to tune the throughput rate. Compared to the unit in [11], we have more configurabililty options by having a slighlty more hardware resources. Furthermore, a configurable crypto-processor [16] is implemented for PQC. It performs NTT multiplication and utilizes single-port SRAM banks and a constant geometry NTT structure to reduce hardware area. The constant geometry NTT is an out-of-place operation that requires input and output polynomials to be stored on different memory banks. Our memory units store input and output polynomials simultaneously; thus, it uses fewer memory banks than the work in [16].

The work in [19] introduces cryptographic hardware for PQC, and it utilizes a NTT-based polynomial multiplier. The memory access pattern is modified to reduce the amount of memory. The pattern is utilized to offer two butterfly operations at the same time. They utilize four memory blocks for NTT operation, where two of them store input polynomials and two store twiddle factors. The control unit offers CTC for the scheme parameters, and it is possible to use it for a specific set of parameters. Similarly, the design in [18] uses a similar implementation for the control unit with different memory bank configurations. We offer RTC for the control unit, which makes it applicable to various scheme parameters compared to those units. As we have discussed, the overall design is implemented in different ways for different applications. It is reasonable to say that the overall design becomes more complex to offer configurability.

IV. Results and Comparison

This section provides a discussion about our implementation results and their comparison with the works in the literature.

A. Prior Works

There are a plethora of works in the literature presenting efficient hardware and software implementations for PQC schemes. A highly optimized AVX2 implementation of NTT utilized in the NewHope and Kyber (v1) schemes is presented in [8]. While Alkim *et al.* implemented Kyber (v2) and NewHope schemes with ISA extension on RISC-V [9], the work in [41] introduces Cortex-M3 implementations for the Kyber (v2), Dilithium, and NewHope schemes.

A polynomial multiplier for the Kyber scheme (v2) is presented with optimizations for FPGA [10], and Banerjee *et al.* introduces a custom crypto-processor implemented on ASIC for NIST's round 2 candidates [16]. The hardware in [13] presents a RISC-V architecture in ASIC and FPGA supporting lattice-based PQC schemes. Fritzmann *et al.* [14] proposed an ASIC implementation of a low-power NTT accelerator for various PQC schemes. The work in [17] introduces RISC-V instruction set extensions while utilizing NTT multiplier for various PQC schemes. There are also NTT accelerators [4], [20] that offer CTC for design parameters. The architecture in [15] presents an accelerator for NTT-based multiplication while having RTC for scheme parameters.

TABLE II
Resource Utilization of Sub-Blocks

BU	Block	Virtex-7	Artix-7
		LUTs/FFs/DSPs/BRAMs	
1	Overall	2128/1144/8/3	2119/1058/8/3
	⌊Mem. Con.	786/263/0/0	775/263/0/0
	⌊But. Unit	703/474/8/0	705/488/8/0
	⌊Mod. Mul.	239/186/8/0	241/100/8/0
8	Overall	10973/5422/64/12	10908/5182/64/12
	⌊Mem. Con.	1358/422/0/0	1358/422/0/0
	⌊But. Unit	7529/3400/64/0	7461/3160/64/0
	⌊Mod. Mul.	1926/1096/64/0	1918/856/64/0
32	Overall	61731/17846/256/48	63032/18182/256/48
	⌊Mem. Con.	7410/1457/0/0	7738/1466/0/0
	⌊But. Unit	46553/10728/256/0	47459/11096/256/0
	⌊Mod. Mul.	7680/1512/256/0	7690/1835/256/0

B. Implementation Results

Our design is implemented in Verilog and synthesized using Vivado 2019.1 tool for Xilinx Virtex-7 FPGA (xc7vx690tffg1761-2) and Artix-7 FPGA (xc7a200t-2fbg676c) with default settings. Implementation results and their comparison with the works in the literature are presented in Table III. The smallest implementation of our hardware design employs one butterfly unit, which utilizes 2128 LUTs, 8 DSPs, 3 BRAMs and runs at 174 MHz on Virtex-7 FPGA. Additionally, Table II presents the resource utilization of sub-blocks on Virtex-7 and Artix-7 platforms. We also synthesized our hardware accelerator for ASIC using a 32 nm standard cell library. The synthesis results for the implementations with different number of BU numbers excluding on-chip memory are presented in Table III.

We present an area vs. latency graph for the proposed hardware with a different number of butterfly units, as shown in Fig. 6. As it requires only a single parameter change, our design can easily be configured to work with different number of butterfly units based on the area/performance requirements of the application. For example, an application targeting high-performance with fast or multiple SHA3 units will also require a fast NTT unit. It should be noted that only the parts of the works performing NTT operation for the same n and q targeting the same platform make a meaningful comparison. Therefore, the comparison presented in this section is not ideal, and it should be considered an estimate.

The area optimized Kyber polynomial multiplier in our earlier work [10] enjoys slightly better performance in comparison with the current design; but our previous architecture in [10] does not have any support for other lattice-based PQC schemes dissimilar from the current design. As expected, this

TABLE III
IMPLEMENTATION RESULTS AND THEIR COMPARISON WITH OTHER WORKS

Design	Platform	n	$\lceil \log_2(q) \rceil$	LUT/FF/DSP/BRAM (mm^2 for ASIC)	Freq. (MHz)	Latency (Clock Cycles) NTT	INTT	P.M.
[16]†	40 nm CMOS	256 512 1024	13 14 14	0.28	72	1289 2826 6155	–	–
[4]	Virtex-7	256	23	888 / – / 7 / 5 5K / – / 56 / 12	125	1096 200	–	–
		512	14	537 / – / 3 / 5.5 2.5K / – / 24 / 12		2340 324		
		1024	14	575 / – / 3 / 11 17.1K / – / 96 / 48		5160 200		
[13]*†	Zynq-7000	256 512 1024	12 14 14	2.9K / 170 / 9 / 0	45	1935 8169 18537	1930 8684 20171	–
[14]†	UMC 65 nm	256 512 1024	13 14 14	0.329	25	2056 4616 10248	–	–
[10]*	Spartan-6	256	12	985 / 444 / 1 / 5	138	904	904	3359
	Artix-7			948 / 352 / 1 / 2.5	190			
[17]*†	Artix-7	256 512 1024	–	2.4K / 1.9K / 7 / 4.5	153	3584 8192 20480	–	–
[20]	Virtex-7	1024	28	1K / 1K / 7 / 2	125	5290	–	–
				16K / 14K / 56 / 24		490		
[41]*	Cortex-M3	256 256 1024	12 23 14	– / – / – / –	16	10819 19347 77001	12994 21006 93128	–
[15]†	Virtex-7	256 512 1024	16	39.6K / – / 224 / 96	150	104 153 249	–	288 468 815
Ours*†	Virtex-7	256^a 256^a	12 13	2128 / 1144 / 8 / 3	174	922 1052	1184 1314	3812 3680
	Artix-7	256^a 512^a	23 14	2119 / 1058 / 8 / 3	117	1052 2334	1318 2854	3688 8072
	32 nm	1024^a 1024^a	14 29	0.053	462	5152 5162	6182 6195	17506 17552
	Virtex-7	256^b 256^b	12 13	11K / 5422 / 64 / 12	186	138 156	176 197	572 550
	Artix-7	256^b 512^b	23 14	11K / 5182 / 64 / 12	140	156 318	198 391	552 1100
	32 nm	1024^b 1024^b	14 29	0.353	416	672 682	812 819	2296 2320
	Virtex-7	256^c 256^c	12 13	61K / 17K / 256 / 48	167	84 95	101 112	306 319
	Artix-7	256^c 512^c	23 14	63K / 18K / 256 / 48	126	103 126	121 141	345 428
	32 nm	1024^c 1024^c	14 29	2.205	416	192 202	233 244	658 690

P.M.: Polynomial Multiplication; ^{a}: 1 BU; ^{b}: 8 BUs; ^{c}: 32 BUs; *: supports Kyber (v2); †: supports multiple n and q at run-time.

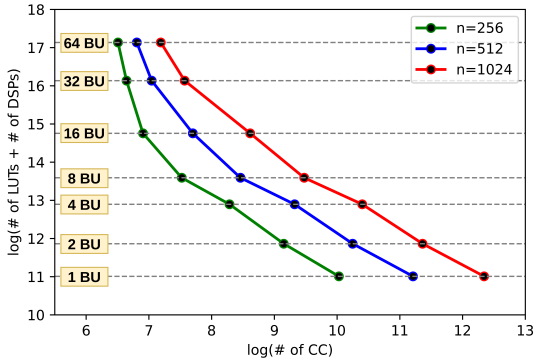

Fig. 6. Area vs Latency for n={256, 512, 1024} with Different Number of Butterfly Units (BU)

additional functionality to support different schemes and CTC & RTC come with an overhead in resource usage.

Our proposed design with one butterfly unit (i.e., one-BU design) uses similar levels of resources as those architectures in [13], [14], [17] with similar configurability capabilities (i.e. all three offering RTC as our one-BU design). Our one-BU design performs one NTT operation with much lower latency than them; thus, it offers better performance. Compared to the NTT unit of the custom crypto-processor Sapphire [16], our one-BU design offers improved performance with higher frequency and fewer clock cycles. Moreover, the one-BU design has much lower latency than Cortex-M3 implementation [41] for one NTT operation.

Our previous works in [4], [20] propose polynomial multipliers with CTC capability but no RTC. Our designs in this work show comparable performance results using slightly more resources than those designs in [4], [20]. Since the current design offers RTC for scheme parameters, these results

confirm our hypothesis that both RTC and CTC are possible without sacrificing performance.

The NTT-based polynomial hardware in another work of ours [15] attains similar performance with our current 32-BU design. It uses a fixed number of BUs, and therefore, it is impossible to use parallelism to control the throughput rate. It offers RTC for scheme parameters and no support for the Kyber (v2) scheme. It uses fewer resources than ours, but it does not work in our operating window and does not have the configurability options offered with the current design.

V. Conclusion

In this paper, we present a highly configurable NTT-based polynomial multiplier architecture for NTT-friendly PQC schemes. The proposed architecture provides run-time configurability for scheme parameters (i.e., n and q) and compile-time configurability for area and performance (i.e., the number of butterfly units). We also present the differences in the adopted design approaches by our design and other similar designs in the literature and especially emphasize configurability options offered by each design.

The proposed architecture can perform NTT, INTT, and NTT-based polynomial multiplication operations for ring sizes ranging from 256 to 1024 and coefficient modulus up to 30 bits, targeting NTT-friendly lattice-based PQC schemes. Therefore, Our architecture can be utilized as an accelerator in lattice-based PQC schemes, and it allows adjusting the trade-off between area and performance by changing a single parameter.

We implemented our architecture with various configurations both in FPGA and ASIC as target platforms and compared our results with those representing the state-of-art in the literature. As we have shown in Table III, the performance of our design is almost as efficient as other designs (including the fastest ones) in terms of latency. On one hand, the run-time configurability of our architecture for various scheme parameters provides acceleration support for a wide range of lattice-based PQC schemes. Its compile-time configurability for the time-area metric, on the other hand, can be used to meet specific design constraints. To the best of our knowledge, there is no other work in the literature that offers both. We achieved this with no impact on latency and with a negligible increase in the area.

Acknowledgements

This research is supported in part by the by The Scientific and Technological Research Council of Turkey under Grant Number 118E725.

References

[1] L. Chen, S. Jordan, Y.-K. Liu, D. Moody, R. Peralta, R. Perlner, and D. Smith-Tone, "Report on post-quantum cryptography," 2016-04-28 2016.

[2] T. Pöppelmann and T. Güneysu, "Area optimization of lightweight lattice-based encryption on reconfigurable hardware," in *2014 IEEE Int. Symp. on Circuits and Systems*, June 2014, pp. 2796–2799.

[3] H. Nejatollahi, N. Dutt, S. Ray, F. Regazzoni, I. Banerjee, and R. Cammarota, "Post-quantum lattice-based cryptography implementations: A survey," *ACM Comput. Surv.*, vol. 51, no. 6, Jan. 2019. [Online]. Available: https://doi.org/10.1145/3292548

[4] A. C. Mert, E. Karabulut, E. Ozturk, E. Savas, and A. Aysu, "An extensive study of flexible design methods for the number theoretic transform," *IEEE Transactions on Computers*, pp. 1–1, 2020.

[5] E. Chu and A. George, *Inside the FFT black box: serial and parallel fast Fourier transform algorithms*. CRC press, 1999.

[6] T. Fritzmann, U. Sharif, D. Müller-Gritschneder, C. Reinbrecht, U. Schlichtmann, and J. Sepulveda, "Towards reliable and secure post-quantum co-processors based on risc-v," in *2019 Design, Automation Test in Europe Conference Exhibition (DATE)*, 2019, pp. 1148–1153.

[7] T. Fritzmann, G. Sigl, and J. Sepúlveda, "RISQ-V: Tightly Coupled RISC-V Accelerators for Post-Quantum Cryptography," *IACR Trans. on CHES*, vol. 2020, no. 4, pp. 239–280, Aug. 2020.

[8] G. Seiler, "Faster AVX2 optimized NTT multiplication for Ring-LWE lattice cryptography," Crypto. ePrint Arch., Report 2018/039, 2018.

[9] E. Alkim, H. Evkan, N. Lahr, R. Niederhagen, and R. Petri, "ISA Extensions for Finite Field Arithmetic - Accelerating Kyber and NewHope on RISC-V," Cryptology ePrint Archive, Report 2020/049, 2020, https://eprint.iacr.org/2020/049.

[10] F. Yaman, A. C. Mert, E. Öztürk, and E. Savaş, "A hardware accelerator for polynomial multiplication operation of crystals-kyber pqc scheme," in *2021 Design, Automation Test in Europe Conference Exhibition (DATE)*, 2021, pp. 1020–1025.

[11] Y. Xing and S. Li, "A compact hardware implementation of cca-secure key exchange mechanism crystals-kyber on fpga," *IACR Transactions on Cryptographic Hardware and Embedded Systems*, vol. 2021, no. 2, p. 328–356, Feb. 2021. [Online]. Available: https://tches.iacr.org/index.php/TCHES/article/view/8797

[12] A. C. Mert, E. Ozturk, and E. Savas, "Design and implementation of a fast and scalable ntt-based polynomial multiplier architecture," Cryptology ePrint Archive, Report 2019/109, 2019.

[13] T. Fritzmann, G. Sigl, and J. Sepúlveda, "Risq-v: Tightly coupled risc-v accelerators for post-quantum cryptography," *IACR Transactions on Cryptographic Hardware and Embedded Systems*, pp. 239–280, 2020.

[14] T. Fritzmann and J. Sepúlveda, "Efficient and Flexible Low-Power NTT for Lattice-Based Cryptography," in *2019 IEEE Int. Symposium on HOST*, May 2019, pp. 141–150.

[15] A. C. Mert, E. Öztürk, and E. Savaş, "Fpga implementation of a run-time configurable ntt-based polynomial multiplication hardware," *Microprocessors and Microsystems*, vol. 78, p. 103219, 2020. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S014193312030380X

[16] U. Banerjee, T. S. Ukyab, and A. P. Chandrakasan, "Sapphire: A Configurable Crypto-Processor for Post-Quantum Lattice-based Protocols," *IACR Trans. on CHES*, pp. 17–61, 2019.

[17] T. Fritzmann, M. V. Beirendonck, D. B. Roy, P. Karl, T. Schamberger, I. Verbauwhede, and G. Sigl, "Masked accelerators and instruction set extensions for post-quantum cryptography," Cryptology ePrint Archive, Report 2021/479, 2021, https://ia.cr/2021/479.

[18] S. S. Roy, F. Vercauteren, N. Mentens, D. D. Chen, and I. Verbauwhede, "Compact Ring-LWE Cryptoprocessor," in *Cryptographic Hardware and Embedded Systems – CHES 2014*, L. Batina and M. Robshaw, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2014, pp. 371–391.

[19] W. Wang, S. Tian, B. Jungk, N. Bindel, P. Longa, and J. Szefer, "Parameterized hardware accelerators for lattice-based cryptography and their application to the hw/sw co-design of qtesla," *IACR Transactions on Cryptographic Hardware and Embedded Systems*, vol. 2020, no. 3, p. 269–306, Jun. 2020. [Online]. Available: https://tches.iacr.org/index.php/TCHES/article/view/8591

[20] A. C. Mert, E. Karabulut, E. Öztürk, E. Savaş, M. Becchi, and A. Aysu, "A Flexible and Scalable NTT Hardware : Applications from Homomorphically Encrypted Deep Learning to Post-Quantum Cryptography," in *2020 DATE*, 2020, pp. 346–351.

[21] W. Tan, B. M. Case, G. Hu, S. Gao, and Y. Lao, "An ultra-highly parallel polynomial multiplier for the bootstrapping algorithm in a fully homomorphic encryption scheme," *Journal of Signal Processing Systems*, pp. 1–14, 2020.

[22] D. T. Nguyen, V. B. Dang, and K. Gaj, "A high-level synthesis approach to the software/hardware codesign of ntt-based post-quantum cryptography algorithms," in *2019 International Conference on Field-Programmable Technology (ICFPT)*, 2019, pp. 371–374.

[23] ——, "High-Level Synthesis in Implementing and Benchmarking Number Theoretic Transform in Lattice-Based Post-Quantum Cryptography Using Software/Hardware Codesign," in *Applied Reconfigurable Computing. Architectures, Tools, and Applications*, F. Rincón, J. Barba, H. K. H. So, P. Diniz, and J. Caba, Eds. Cham: Springer International Publishing, 2020, pp. 247–257.

[24] M. S. Riazi, K. Laine, B. Pelton, and W. Dai, "Heax: An architecture for computing on encrypted data," in *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS '20. New York, NY, USA: Association for Computing Machinery, 2020, p. 1295–1309. [Online]. Available: https://doi.org/10.1145/3373376.3378523

[25] J. Bos, L. Ducas, E. Kiltz, T. Lepoint, V. Lyubashevsky, J. M. Schanck, P. Schwabe, G. Seiler, and D. Stehlé, "CRYSTALS – Kyber: a CCA-secure module-lattice-based KEM," Cryptology ePrint Archive, Report 2017/634, 2017, https://eprint.iacr.org/2017/634.

[26] V. Lyubashevsky and G. Seiler, "NTTRU: Truly Fast NTRU Using NTT," *IACR Trans. on CHES*, vol. 2019, no. 3, pp. 180–201, May 2019.

[27] E. Alkim, L. Ducas, T. Pöppelmann, and P. Schwabe, "Post-quantum key exchange—a new hope," in *25Th {USENIX} security symposium ({USENIX} security 16)*, 2016, pp. 327–343.

[28] L. Ducas, E. Kiltz, T. Lepoint, V. Lyubashevsky, P. Schwabe, G. Seiler, and D. Stehlé, "Crystals-dilithium: A lattice-based digital signature scheme," *IACR Trans. on CHES*, pp. 238–268, 2018.

[29] P.-A. Fouque, J. Hoffstein, P. Kirchner, V. Lyubashevsky, T. Pornin, T. Prest, T. Ricosset, G. Seiler, W. Whyte, and Z. Zhang, "Falcon: Fast-Fourier lattice-based compact signatures over NTRU," *Submission to the NIST's post-quantum cryptography standardization process*, 2018.

[30] E. Alkim, P. S. Barreto, N. Bindel, P. Longa, and J. E. Ricardini, "The Lattice-Based Digital Signature Scheme qTESLA," *IACR Cryptology ePrint Archive*, vol. 2019, p. 85, 2019.

[31] J.-P. D'Anvers, A. Karmakar, S. S. Roy, and F. Vercauteren, "Saber: Module-lwr based key exchange, cpa-secure encryption and cca-secure kem," Cryptology ePrint Archive, Report 2018/230, 2018, https://ia.cr/2018/230.

[32] T. Yanik, E. Savas, and C. K. Koc, "Incomplete reduction in modular arithmetic," *IEE Proceedings - Computers and Digital Techniques*, vol. 149, no. 2, pp. 46–52, March 2002.

[33] R. J. McEliece, "A Public-Key Cryptosystem Based On Algebraic Coding Theory," *Deep Space Network Progress Report*, vol. 44, pp. 114–116, Jan. 1978.

[34] T. Pöppelmann, T. Oder, and T. Güneysu, "High-performance ideal lattice-based cryptography on 8-bit ATxmega microcontrollers," in *International Conference on Cryptology and Information Security in Latin America*. Springer, 2015, pp. 346–365.

[35] J. Bos, L. Ducas, E. Kiltz, T. Lepoint, V. Lyubashevsky, J. M. Schanck, P. Schwabe, G. Seiler, and D. Stehlé, "CRYSTALS-Kyber: a CCA-secure module-lattice-based KEM," in *IEEE Euro S&P*, 2018, pp. 353–367.

[36] Z. Liu, H. Seo, S. S. Roy, J. Großschädl, H. Kim, and I. Verbauwhede, "Efficient ring-lwe encryption on 8-bit avr processors," in *International Workshop on Cryptographic Hardware and Embedded Systems*. Springer, 2015, pp. 663–682.

[37] S. Sinha Roy, F. Turan, K. Jarvinen, F. Vercauteren, and I. Verbauwhede, "Fpga-based high-performance parallel architecture for homomorphic computing on encrypted data," in *2019 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, 2019, pp. 387–398.

[38] A. C. Mert, E. Ozturk, and E. Savas, "Low-latency asic algorithms of modular squaring of large integers for vdf evaluation," *IEEE Transactions on Computers*, pp. 1–1, 2020.

[39] N. Zhang, B. Yang, C. Chen, S. Yin, S. Wei, and L. Liu, "Highly Efficient Architecture of NewHope-NIST on FPGA using Low-Complexity NTT/INTT," *IACR Trans. on CHES*, vol. 2020, no. 2, pp. 49–72, 2020.

[40] G. Xin, J. Han, T. Yin, Y. Zhou, J. Yang, X. Cheng, and X. Zeng, "VPQC: A Domain-Specific Vector Processor for Post-Quantum Cryptography Based on RISC-V Architecture," *IEEE Trans. on Circuits and Systems I: Regular Papers*, vol. 67, no. 8, pp. 2672–2684, 2020.

[41] D. O. C. Greconici, M. J. Kannwischer, and D. Sprenkels, "Compact dilithium implementations on cortex-m3 and cortex-m4," Cryptology ePrint Archive, Report 2020/1278, 2020, https://ia.cr/2020/1278.