IACR Transactions on Cryptographic Hardware and Embedded Systems
ISSN 2569-2925, Vol. 2022, No. 1, pp. 270–295. DOI:10.46586/tches.v2022.i1.270-295

A Compact and High-Performance Hardware Architecture for CRYSTALS-Dilithium

Cankun Zhao[1,+], Neng Zhang[1,+], Hanning Wang[1], Bohan Yang[1], Wenping Zhu[1], Zhengdong Li[1], Min Zhu[2], Shouyi Yin[1], Shaojun Wei[1] and Leibo Liu[1,*]

[1] School of Integrated Circuits, Tsinghua University, Beijing, China.
[2] Wuxi Micro Innovation Integrated Circuit Design Co., Ltd., Wuxi, China.
{zck19,zhangn16}@mails.tsinghua.edu.cn;{wanghn,bohanyang,zhuwp,lizd}@tsinghua.edu.cn;zhumin@mucse.com;{yinsy,wsj,liulb}@tsinghua.edu.cn
[+] These authors contributed equally to this work.
[*] Corresponding author.

Abstract. The lattice-based CRYSTALS-Dilithium scheme is one of the three third-round digital signature finalists in the National Institute of Standards and Technology Post-Quantum Cryptography Standardization Process. Due to the complex calculations and highly individualized functions in Dilithium, its hardware implementations face the problems of large area requirements and low efficiency. This paper proposes several optimization methods to achieve a compact and high-performance hardware architecture for round 3 Dilithium. Specifically, a segmented pipelined processing method is proposed to reduce both the storage requirements and the processing time. Moreover, several optimized modules are designed to improve the efficiency of the proposed architecture, including a pipelined number theoretic transform module, a SampleInBall module, a Decompose module, and three modular reduction modules. Compared with state-of-the-art designs for Dilithium on similar platforms, our implementation requires 1.4×/1.4×/3.0×/4.5× fewer LUTs/FFs/BRAMs/DSPs, respectively, and 4.4 × /1.7 × /1.4× less time for key generation, signature generation, and signature verification, respectively, for NIST security level 5.

Keywords: CRYSTALS-Dilithium · FPGA · post-quantum cryptography · digital signature · module learning with errors

1 Introduction

Post-quantum cryptography (PQC) refers to cryptographic algorithms that are secure against both quantum and classical computers. Since conventional public-key cryptographic algorithms, which are based on the mathematical hardness of computing integer factorizations and discrete logarithms, can be broken by Shor's algorithm [Sho94] with a large-scale quantum computer, the confidentiality and integrity of digital communications on the Internet and elsewhere are under threat. To ensure the security of information systems in the upcoming quantum era, researchers have begun to study quantum-resistant public-key cryptographic algorithms. The National Institute of Standards and Technology (NIST) initiated the PQC Standardization Process in 2016, and 69 algorithms were submitted for the first round in 2017. After two rounds of evaluation and review, seven finalists and eight alternate candidates were selected as the round 3 candidates in July 2020. There are three digital signature algorithms among the seven finalists, CRYSTALS-Dilithium [LDK+20a], FALCON [PFH+20], and Rainbow [DCP+20]. The security of Rainbow has been affected by recent cryptanalysis [Beu20, Din20], which increases the probability that Dilithium will eventually be standardized.

Received: 2021-07-15 Accepted: 2021-09-15 Published: 2021-11-19

There are a large number of polynomial multiplications in Dilithium, leading to both a long processing time and considerable storage requirements. In addition, compared with conventional signature schemes, the operations in Dilithium are more complicated and contain several unusual functions, which cause great difficulty in efficiently implementing Dilithium in hardware. Most existing works on implementing and evaluating Dilithium have used pure software methods [GKOS18, RGCB19, GKS21] or hardware-software codesign methods [BUC19a]. Full hardware implementations of Dilithium are still very rare. [SBNK19] implemented high-level synthesis (HLS)-based hardware designs for round 2 Dilithium and used optimizations such as loop unrolling and loop pipelining to speed up the algorithm. [RMJ+21] proposed the first manually designed hardware implementation of round 2 Dilithium, using a parallelization-based method to achieve high frequency and high speed. [LSG21] explored implementing round 3 Dilithium with fewer resources by efficiently using digital signal processors (DSPs).

However, these works did not sufficiently optimize their implementations for Dilithium, resulting in high resource consumption and low efficiency. [SBNK19] could not optimize its hardware structure specifically for Dilithium due to its HLS-based implementation method. [RMJ+21] used many resources to straightforwardly map the algorithm to hardware, which led to low efficiency. [LSG21] somewhat reduced its resource usage by reusing modules and using DSPs, but the architecture has a low degree of parallelism, which results in low utilization of its modules. Overall, an efficient Dilithium hardware architecture that is fully optimized for Dilithium is still unavailable.

In this paper, several optimization methods are proposed to achieve a compact, efficient hardware architecture for round 3 Dilithium. Our contributions are summarized as follows:

- A segmented pipelined processing method is proposed, in which operations in the algorithms are divided into multiple segments and the hardware processes one segment at a time in a pipelined manner. This method considerably reduces the storage requirements for intermediate results and hides the execution time of many operations. Meanwhile, the core modules are reused for different segments, endowing our design with high efficiency.

- Several optimized modules are designed for Dilithium, including a high-speed pipelined number theoretic transform (NTT) module, a BRAM-based SampleInBall module, a compact Decompose module, and three customized modular reduction modules. These optimized modules accelerate the processing of corresponding functions with limited resources.

- To accelerate algorithms on resource-constrained hardware, several design trade-offs are proposed and adopted. As a result, our design uses 30k LUTs, 10k FFs, 11 BRAMs, and 10 DSPs, 1.4×, 1.4×, 3.0×, and 4.5× fewer, respectively, than state-of-the-art designs for Dilithium on similar devices for NIST security level 5. Moreover, our design computes key generation, signature generation, and signature verification at speeds of 11,051, 1,977, and 10,716 operations per second (OP/s) for security level 5, which is approximately 4.4×, 1.7×, and 1.4× faster, respectively, than state-of-the-art designs.

The rest of this paper is structured as follows: Section 2 first introduces the notation used in this paper and then gives a brief introduction to Dilithium and some individualized functions in this scheme. Section 3 first describes the system architecture of our design, then introduces the segmented pipelined processing method, and finally presents the details of our storage scheme. Section 4 introduces the design of several optimized modules. Section 5 gives performance results on FPGA and presents comparisons with related works. Finally, Section 6 is our conclusion.

2 Preliminaries

2.1 Notation

We use $\mathbb{Z}_q$ to denote the ring of integers modulo prime q, $\mathbb{Z}_q[X]$ to denote the ring of integer polynomials modulo prime q, $R = \mathbb{Z}[X]/(X^n+1)$ to denote the ring of integer polynomials modulo X^n+1, and $R_q = \mathbb{Z}_q[X]/(X^n+1)$ to denote the ring of integer polynomials modulo both q and X^n+1. The values of q and n are always 8380417 and 256, respectively, in Dilithium. We use letters in regular font to denote elements in R or R_q, bold lower-case letters to denote column vectors with coefficients in R or R_q, and bold upper-case letters to denote matrices. For a positive integer α, we define $r' = r \bmod^+ \alpha$ to be the unique integer r' in the range $0 \leq r' \leq \alpha$ such that $r' \equiv r \bmod \alpha$, and we define $r' = r \bmod^\pm \alpha$ to be the unique integer r' in the range $-\frac{\alpha}{2} < r' \leq \frac{\alpha}{2}$ such that $r' \equiv r \bmod \alpha$. For an element $w \in \mathbb{Z}_q$, we define $||w||_\infty = |w \bmod^\pm q|$. For $w = w_0 + w_1X + \cdots + w_{n-1}X^{n-1} \in R$, we define $||w||_\infty = \max_i ||w_i||_\infty$. For $\boldsymbol{w} = (w_1, \ldots, w_k) \in R^k$, we define $||\boldsymbol{w}||_\infty = \max_i ||w[i]||_\infty$. In addition, we use S_η to denote all elements $w \in R$ such that $||w||_\infty \leq \eta$, and we use $\tilde{S}_\eta$ to denote all elements $w \in R$ whose coefficients are all in the range $-\eta < w_i \leq \eta$. B_τ is used to denote the set of elements of R that have τ coefficients that are either -1 or 1, while the rest are 0. The Boolean operator [[statement]] evaluates to 1 if the statement is true and to 0 otherwise.

Algorithm 1 $\mathsf{KeyGen}()$

1: $\zeta \Leftarrow \{0,1\}^{256}$
2: $(\rho, \rho', K) \in \{0,1\}^{256} \times \{0,1\}^{512} \times \{0,1\}^{256} \Leftarrow \mathsf{H}(\zeta)$
3: $(\boldsymbol{s}_1, \boldsymbol{s}_2) \in S_\eta^l \times S_\eta^k \Leftarrow \mathsf{ExpandS}(\rho')$
4: $\boldsymbol{A} \in R_q^{k \times l} \Leftarrow \mathsf{ExpandA}(\rho)$
5: $\boldsymbol{t} \Leftarrow \boldsymbol{A}\boldsymbol{s}_1 + \boldsymbol{s}_2$
6: $(\boldsymbol{t}_1, \boldsymbol{t}_0) \Leftarrow \mathsf{Power2Round}_q(\boldsymbol{t}, d)$
7: $tr \in \{0,1\}^{256} \Leftarrow \mathsf{H}(\rho||\boldsymbol{t}_1)$
8: **return** $(pk = (\rho, \boldsymbol{t}_1), sk = (\rho, K, tr, \boldsymbol{s}_1, \boldsymbol{s}_2, \boldsymbol{t}_0))$

Algorithm 2 $\mathsf{Sign}(sk, M)$

1: $\boldsymbol{A} \in R_q^{k \times l} \Leftarrow \mathsf{ExpandA}(\rho)$
2: $\mu \in \{0,1\}^{512} \Leftarrow \mathsf{H}(tr||M)$
3: $\rho' \in \{0,1\}^{512} \Leftarrow \mathsf{H}(K||\mu)$ (or $\rho' \Leftarrow \{0,1\}^{512}$)
4: $\kappa \Leftarrow 0, rej \Leftarrow 1$
5: **while** $rej = 1$ **do**
6: $\quad \boldsymbol{y} \in \tilde{S}_{\gamma_1}^l \Leftarrow \mathsf{ExpandMask}(\rho', \kappa)$
7: $\quad \kappa \Leftarrow \kappa + l$
8: $\quad \boldsymbol{w} \Leftarrow \boldsymbol{A}\boldsymbol{y}$
9: $\quad (\boldsymbol{w}_1, \boldsymbol{w}_0) \Leftarrow \mathsf{Decompose}_q(\boldsymbol{w})$
10: $\quad \tilde{c} \in \{0,1\}^{256} \Leftarrow \mathsf{H}(\mu||\boldsymbol{w}_1)$
11: $\quad c \in B_\tau \Leftarrow \mathsf{SampleInBall}(\tilde{c})$
12: $\quad \boldsymbol{z} \Leftarrow \boldsymbol{y} + c\boldsymbol{s}_1$
13: $\quad \boldsymbol{r}_0 \Leftarrow \boldsymbol{w}_0 - c\boldsymbol{s}_2$
14: $\quad$ **if** $||\boldsymbol{z}||_\infty < \gamma_1 - \beta$ **and** $||\boldsymbol{r}_0||_\infty < \gamma_2 - \beta$ **then**
15: $\quad\quad \boldsymbol{h} \Leftarrow \mathsf{MakeHint}'_q(\boldsymbol{r}_0 + c\boldsymbol{t}_0, \boldsymbol{w}_1)$
16: $\quad\quad$ **if** $||c\boldsymbol{t}_0||_\infty < \gamma_2$ **and** [[# of 1s in $\boldsymbol{h}$ is $\leq \omega$]] **then**
17: $\quad\quad\quad rej \Leftarrow 0$
18: **return** $\sigma = (\tilde{c}, \boldsymbol{z}, \boldsymbol{h})$

Algorithm 3 $\mathsf{Verify}(pk, M, \sigma = (\tilde{c}, \boldsymbol{z}, \boldsymbol{h}))$

1: $\boldsymbol{A} \in R_q^{k \times l} \Leftarrow \mathsf{ExpandA}(\rho)$
2: $\mu \in \{0,1\}^{512} \Leftarrow \mathsf{H}(\mathsf{H}(\rho||\boldsymbol{t}_1)||M)$
3: $c \in B_\tau \Leftarrow \mathsf{SampleInBall}(\tilde{c})$
4: $\boldsymbol{w}_1' \Leftarrow \mathsf{UseHint}_q(\boldsymbol{h}, \boldsymbol{A}\boldsymbol{z} - c\boldsymbol{t}_1 \cdot 2^d, 2\gamma_2)$
5: **return** $[[||\boldsymbol{z}||_\infty < \gamma_1 - \beta]]$ **and** $[[\tilde{c} = \mathsf{H}(\mu||\boldsymbol{w}_1')]]$ **and** $[[\#\text{ of 1s in } \boldsymbol{h} \text{ is } \leq \omega]]$

Table 1: Parameters of the three Dilithium security levels in round 3.

NIST Security Level	q	d	τ	γ_1	γ_2	(k,l)	η	β	ω
2	8380417	13	39	2^{17}	$(q-1)/88$	(4,4)	2	78	80
3	8380417	13	49	2^{19}	$(q-1)/32$	(6,5)	4	196	55
5	8380417	13	60	2^{19}	$(q-1)/32$	(8,7)	2	120	75

2.2 CRYSTALS-Dilithium

CRYSTALS-Dilithium is a post-quantum signature scheme based on the hardness of the module learning with errors (MLWE) problem. The scheme is based on the "Fiat-Shamir with Aborts" approach [Lyu09, Lyu12], and is similar to the scheme proposed in [GLP12, BG14]. A distinctive feature of Dilithium that makes it different from the previous schemes (e.g., [BG14] and qTESLA [ABB+19]) is that the public key size is reduced by a factor of approximately two at the cost of increasing the signature size by less than 100 bytes.

The pseudocode for Dilithium's key generation, signature generation and signature verification algorithms are presented in Algorithms 1, 2, and 3, respectively. A brief introduction to these algorithms from a computational perspective is given below. For complete information and details of the different functions, readers are referred to the original paper [LDK+21].

Main operations. From the perspective of computational complexity, a main operation in the entire scheme is polynomial multiplication over the ring R_q via the NTT. To be more precise, the multiplication operands in this scheme are vectors or matrices whose coefficients are polynomials in R_q, so there are many continuous polynomial multiplications in the scheme. Therefore, one focus of this work is to perform continuous NTT operations efficiently.

The other major time-consuming operation in Dilithium is hashing. Two hashing functions are used in Dilithium, i.e., SHAKE-256 and SHAKE-128. Specifically, the H, ExpandS, ExpandMask, and SampleInBall functions use SHAKE-256, and the ExpandA function uses SHAKE-128. The ExpandA function is used to generate the matrix $\boldsymbol{A}$ from a seed ρ so that the public key can contain only a 256-bit seed ρ instead of a matrix of $k \cdot l$ polynomials. As a tradeoff, in all three phases, the ExpandA function will require a long time to run.

In addition, several individualized functions are used in Dilithium to reduce the length of the public key or sample elements of B_τ; these functions are introduced in the next section.

Signature generation. The Sign algorithm first generates a seed ρ' and then performs a loop to generate a signature until it meets a series of security conditions. In the loop, the main operations are hashing and four multiplications, i.e., $\boldsymbol{A}\boldsymbol{y}$, $c\boldsymbol{s}_1$, $c\boldsymbol{s}_2$, and $c\boldsymbol{t}_0$.

Notably, the pseudocode for Sign in Algorithm 2 is not completely the same as in the original paper [LDK+21]. We use an alternative method of decomposing and computing the

hints, further details of which can be found in Section 5.1 of the original paper [LDK+21].

Parameter sets. Dilithium's NIST PQC submission for round 3 includes three parameter sets that correspond to NIST security levels 2, 3, and 5, as shown in Table 1. Compared to the round 2 version, a new parameter set corresponding to security level 5 was added, which has larger matrix and vector sizes and, thus, requires a larger storage space and longer processing time. In addition, a new modulus $(q-1)/88$ was added, whose modular reduction calculation is more complicated than that of the moduli in round 2.

2.3 Individualized Functions in Dilithium

This section introduces several individualized functions in Dilithium that are rarely used in other cryptographic algorithms. Straightforwardly implementing these functions will result in an unnecessary waste of resources and time; therefore, the functions introduced in this section are implemented using optimized methods or customized modules in this work. These functions include four functions for reducing the size of the public key (introduced in Section 2.3.1, 2.3.2) and a function for creating a random element in B_τ (introduced in Section 2.3.3).

2.3.1 Power2Round and Decompose

$\mathsf{Power2Round}_q$ and $\mathsf{Decompose}_q$ are used to break up elements in $\mathbb{Z}_q$ into their "high-order" bits and "low-order" bits. The former function is the straightforward bitwise way to break up an element $r = r_1 \cdot 2^d + r_0$, where $r_0 = r \bmod^{\pm} 2^d$ and $r_1 = (r - r_0)/2^d$. Since the $\mathsf{Power2Round}_q$ function is rather simple and is used only in KeyGen, below, we introduce the $\mathsf{Decompose}_q$ function.

Roughly speaking, for a finite field element r in $\mathbb{Z}_q$, $\mathsf{Decompose}_q$ computes high and low bits r_1 and r_0 such that $r = r_1 \cdot 2\gamma_2 + r_0$, where $-\gamma_2 < r_0 \leq \gamma_2$, except for the border case. $2\gamma_2$ is chosen to be a divisor of $q-1$. For the border case, when r minus r_0 is equal to $q-1$, the high bits r_1 are set to zero, and the low bits r_0 are reduced by one. Algorithm 4 shows the definition of the function $\mathsf{Decompose}_q$.

There are two methods to realize $\mathsf{Decompose}_q$. The first method is to perform modular reduction to obtain the centralized remainders r_0 and then calculate r_1. The second method is to find r_1 directly from the input r and then calculate r_0 according to r_1. The submissions of Dilithium to NIST for round 2 [LDK+19] and round 3 [LDK+20b] provide two reference software implementations of $\mathsf{Decompose}_q$, which use the first and second methods, respectively. However, the round 2 reference implementation of $\mathsf{Decompose}_q$ cannot work effeciently with the new modulus in round 3. The round 3 reference implementation of $\mathsf{Decompose}_q$ uses multiple multiplications and, thus, is too costly for hardware implementation.

Algorithm 4 $\mathsf{Decompose}_q(r)$

1: $r \Leftarrow r \bmod^{+} q$
2: $r_0 \Leftarrow r \bmod^{\pm} (2\gamma_2)$
3: **if** $r - r_0 = q - 1$ **then**
4: $\quad r_1 \Leftarrow 0$
5: $\quad r_0 \Leftarrow r_0 - 1$
6: **else**
7: $\quad r_1 \Leftarrow (r - r_0)/(2\gamma_2)$
8: **return** (r_1, r_0)

Algorithm 5 SampleInBall($\tilde{c}$)

1: Initialize $c \in R \Leftarrow 0 + 0 \cdot X + \cdots + 0 \cdot X^{n-1}$
2: **for** $i = n - \tau$ to $n - 1$ **do**
3: $\quad r \Leftarrow \{0, 1, \ldots, i\}$
4: $\quad s \Leftarrow \{0, 1\}$
5: $\quad c_i \Leftarrow c_r$
6: $\quad c_r \Leftarrow (-1)^s$
7: **return** c

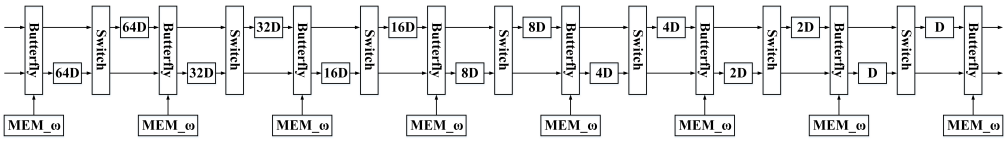


Figure 1: Block graph of a 256-point R2MDC FFT.

2.3.2 MakeHint and UseHint

The function called "MakeHint" records the carry to the "high-order" bits in the addition of an arbitrary element $r \in \mathbb{Z}_q$ and another small element $z \in \mathbb{Z}_q$, and $\mathsf{UseHint}_q$ uses the hint generated in this way to recover the "high-order" bits of the sum. The straightforward way to perform the former function, denoted by $\mathsf{MakeHint}_q$, is to calculate the high part of r and the high part of $r + z$ individually and compare them to determine whether they are the same. As mentioned above, we use an alternative method proposed in the original paper to calculate the hints, using four simple operations instead of two complex $\mathsf{Decompose}_q$ operations, denoted by $\mathsf{MakeHint'}_q$.

2.3.3 SampleInBall

SampleInBall is used to generate an element in B_τ, i.e. a polynomial in R that has only τ nonzero coefficients, whose values are either -1 or 1. This algorithm is an "inside-out" version of the Fisher-Yates shuffle algorithm [FY38], and its pseudocode is shown in Algorithm 5.

This algorithm is suitable for software implementation but is not friendly to hardware implementation because it needs frequent data movements, and every step exhibits data dependence on all previous steps. Specifically, in every loop of the algorithm, the coefficient of a random position r needs to be moved to a new position i, as shown in Line 5 of Algorithm 5. Meanwhile, the value of the polynomial before this movement depends on the operations in all previous loops. In addition, all operations after SampleInBall depend on the output c of this function, so its speed will directly affect the speed of the entire signature algorithm.

2.4 Radix-2 Multipath Delay Commutator

The Radix-2 Multipath Delay Commutator (R2MDC) architecture is a popular pipeline architecture for the fast Fourier transform (FFT) [HT96]. Compared with the popular in-place FFT architecture, R2MDC has fewer memory accesses, a more regular ordering of the input and output data, and simpler control logic, and it is better at processing multiple FFTs continuously. Figure 1 shows the architecture for a 256-point R2MDC FFT, which needs two input coefficients per cycle to achieve a 100% utilization rate of the butterfly units. This architecture can process both radix-2 decimation-in-time (DIT) FFTs and radix-2 decimation-in-frequency (DIF) FFTs by using different butterfly units and twiddle factors. In addition, it can process both the FFT and the inverse FFT (IFFT), with the

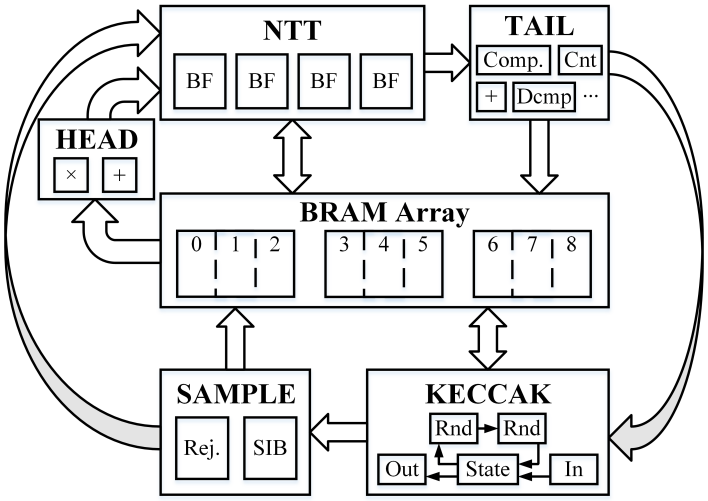


Figure 2: System architecture.

difference being that the IFFT requires additional postprocessing and different twiddle factors.

3 Design Decisions

This section will introduce the overall architecture design, while the next section will introduce the optimized modules. Our goals are to use limited resources to achieve a fast speed and use the same hardware architecture to support all phases and all available security levels.

3.1 System Architecture

Figure 2 shows the system architecture of our hardware design; for clarity, the control module and the packing/unpacking module are not shown. Our design has six main components, namely, a BRAM array, an NTT module, a HEAD module, a TAIL module, a KECCAK module, and a SAMPLE module. A brief introduction to these modules follows.

The BRAM array contains nine dual-port 36k BRAMs arranged in groups of three. This array is mainly used to store polynomials for the secret key, signature, and intermediate results. In addition, some areas of the BRAM array are used in place of the shift registers in the NTT module, which are introduced in Section 4.1. The consumption of BRAMs in our design is extremely low compared to that in related works due to our segmented pipelined processing method (introduced in Section 3.2), the on-the-fly matrix $\boldsymbol{A}$ calculation strategy, and the efficient use of the BRAM array. The details of the arrangement of the BRAM array are introduced in Section 3.3.

The high-speed pipelined NTT module is designed to accelerate polynomial multiplication. It contains four butterfly units and can be used to calculate pipelined NTTs, pipelined INTTs, or 4-way parallel pointwise multiplications. When calculating NTTs/INTTs, it takes only one coefficient as input and outputs one per cycle. It can perform continuous NTTs/INTTs on multiple polynomials, and the execution time is $256 \times k + 296$ clock cycles, where k is the number of processed polynomials. When calculating multiplications, the four butterfly units are reused to perform four modular multiplications and four modular additions per cycle. The details of this module are introduced in Section 4.1.

The HEAD module and TAIL module are designed for our segmented pipelined

processing. They are placed before and after the NTT module in the pipeline, respectively. The HEAD module contains a modular multiplier and a modular adder, which are used to calculate $\hat{\boldsymbol{y}}+\hat{c}\hat{\boldsymbol{s}}_1$, $\hat{c}\hat{\boldsymbol{s}}_2$, and $\hat{c}\hat{\boldsymbol{t}}_0$ in Sign. The TAIL module contains a comparator, a modular adder, a counter, the Decompose module, the Power2Round module, the MakeHint module, and the UseHint module. These submodules are used to compute operations such as $||\boldsymbol{z}||_\infty < \gamma_1 - \beta$, some additions after an INTT, counting the number of 1s in $\boldsymbol{h}$, and other corresponding functions. The HEAD and TAIL modules can reduce the storage requirements and speed up processing, the details of which are introduced in the next subsection.

The KECCAK module is designed for SHAKE-128 and SHAKE-256, which use the same Keccak-f[1600] permutation with different rates (1344 and 1088, respectively). Thus we implement one permutation core for both functions. The permutation core contains two cascaded straightforward implementations of the round function, i.e., the core can compute two rounds per cycle. Therefore, the 24 rounds of the whole Keccak permutation are performed in 12 cycles. In addition, the KECCAK module contains three large registers, i.e., a 1600-bit state register, a 1088-bit input register, and a 1344-bit output register. The state register stores the state array which is repeatedly updated within a computational procedure. The input register concatenates and stores the input bit strings temporarily until they are ready to be copied or added (exclusive-or) to the state register. The permutation results are squeezed out and stored in the output register waiting for sampling so that the permutation core can continue running without pause.

The SAMPLE module contains a rejection sampling module and a BRAM-based SampleInBall module. The former can perform rejection sampling based on several parameters. The latter is designed to execute the Fisher-Yates shuffle algorithm used in Dilithium. The SampleInBall module, which is introduced in Section 4.2, uses fewer resources and has a faster speed than similar works.

3.2 Segmented Pipelined Processing

A segmented pipelined processing method is proposed for Dilithium, in which operations in the algorithms are divided into several segments. Different segments are processed serially, and the operations within a segment are processed in a pipelined manner. Pipelining can reduce the storage requirements for intermediate results and reduce memory access. Meanwhile, segmentation can reduce the hardware resources required by the algorithm and allow full use to be made of our modules. This section first uses Sign as an example to introduce the segmented pipelined processing method, which is similar for KeyGen and Verify, and then briefly introduces the HEAD and TAIL modules designed for pipelined processing. Figure 3, 4, and 5 briefly show how this method is applied to Sign, KeyGen, and Verify, respectively. For clarity, the horizontal lengths of the different operations in

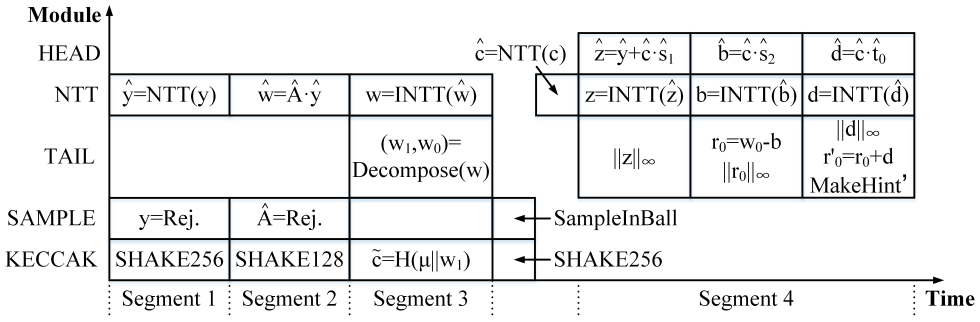


Figure 3: Segmented pipelined processing of Sign loop.

Figure 3, 4, and 5 are not proportional to the actual processing times.

Segmented pipelined processing of Sign. We take the core loop of the Sign algorithm (Lines 5-17 in Algorithm 2) as an example to introduce our segmented pipelined processing method. The operations in the loop are approximately divided into four segments, as shown in Figure 3.

The first segment corresponds to Line 6 of Algorithm 2. In this segment, the KECCAK module and the rejection sampling module are used to produce $\boldsymbol{y}$, and the result is sent to the NTT module for NTT transformation. The generation of every polynomial of $\boldsymbol{y}$ requires five rounds of Keccak permutation and thus needs 60 clock cycles. The NTT module needs 256 cycles to process one polynomial, so the generation speed of $\boldsymbol{y}$ can meet this demand.

The second segment corresponds to Line 8 of Algorithm 2. Every element of the matrix $\boldsymbol{A}$ is generated sequentially using the KECCAK module and the rejection sampling module. Then, the coefficients of $\boldsymbol{A}$ and $\boldsymbol{y}$ are sent to the NTT module to perform the 4-way parallel pointwise multiplication $\hat{\boldsymbol{A}} \cdot \hat{\boldsymbol{y}}$ in the NTT domain. For each element of $\boldsymbol{A}$, the random number generation process via the KECCAK module requires 60 cycles, the rejection sampling process via the SAMPLE module requires 70 cycles, and the multiplication process via the NTT module requires 64 cycles. The speeds of these three modules are approximately the same, which means that our modules have high utilization.

The third segment corresponds to Lines 8-10. First, the NTT module performs INTTs on $\hat{\boldsymbol{w}}$ and outputs $\boldsymbol{w}$. Then, the TAIL module performs the Decompose function on $\boldsymbol{w}$. Finally, the output $\boldsymbol{w}_1$ of Decompose is absorbed by the KECCAK module to prepare for calculating $\tilde{c}$.

The fourth segment corresponds to Lines 12-16. First, the HEAD module performs the pointwise multiplication of $\hat{c}$ and $\hat{\boldsymbol{s}}_1$, $\hat{\boldsymbol{s}}_2$, and $\hat{\boldsymbol{t}}_0$ in sequence. The results are sent to the NTT module for INTT transformation. Finally, the output of the NTT module is

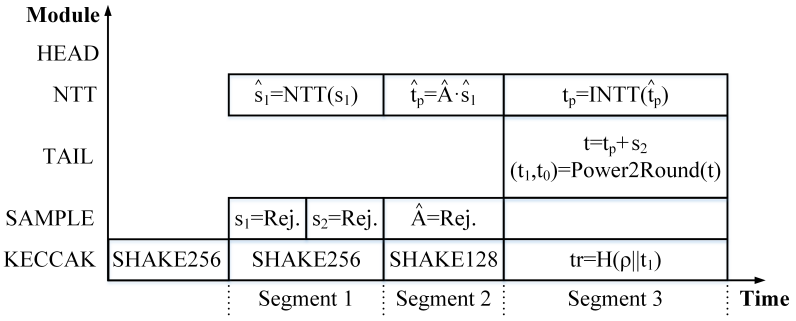


Figure 4: Segmented pipelined processing of KeyGen.

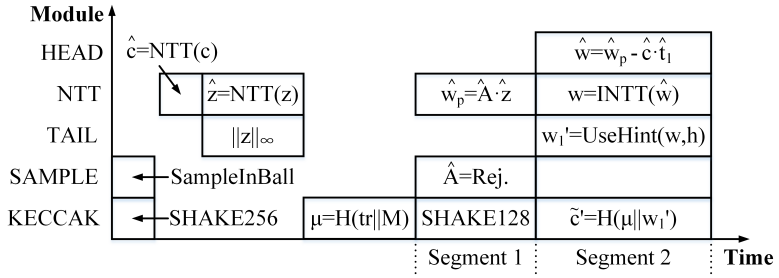


Figure 5: Segmented pipelined processing of Verify.

sent to the TAIL module to determine whether the generated signature meets the security conditions and to calculate the hints $\boldsymbol{h}$.

The HEAD and TAIL modules. The HEAD module is used to calculate the pointwise multiplication of $\hat{c}$ and $\hat{\boldsymbol{s}}_1, \hat{\boldsymbol{s}}_2, \hat{\boldsymbol{t}}_0$. The intermediate results generated by HEAD are processed by NTT immediately in a pipelined manner without being stored. Without this module, we would need to use the NTT module to perform pointwise multiplications, store the intermediate results, and use the NTT module to perform INTT transformation. For the highest security level, $7 + 8 + 8$ polynomials would need to be stored, and 6 BRAMs would need to be added. Meanwhile, these intermediate results have a very short life span, which would result in low utilization of the BRAMs. In addition, the NTT module would use 64 cycles to perform pointwise multiplications for one pair of polynomials. Thus, by using this module, we reduce the loop time by 23×64 clock cycles, as the multiplication time for 23 pairs of polynomials is hidden.

The TAIL module is composed of several submodules, as mentioned in Section 3.1. All corresponding functions of those submodules are performed on every element of the polynomials. By means of the TAIL module, every coefficient is processed immediately after INTT transformation. Thus, the execution time of those functions is hidden, and the storage requirements for intermediate results are reduced. Specifically, the Decompose operation on Line 9, the addition operations on Lines 13 and 15, the condition judgments on Lines 14 and 16, and the MakeHint' operation on Line 15 in the Sign algorithm are all hidden. If these operations were to be accomplished via the straightforward implementation method, for security level 5, they would require more than 3,000 cycles even with four copies of the corresponding submodules for acceleration. By contrast, with one TAIL module, our method needs a latency of only one cycle.

Overall, the proposed segmented pipelined processing method and the HEAD and TAIL modules reduce both the storage requirements and the number of processing cycles.

3.3 BRAM Array

To achieve low BRAM consumption, two design considerations are applied in this design, and the usage of the BRAM array is carefully managed to achieve a high utilization rate. This implementation is designed to support all three phases and all three security levels. The largest and most complex storage requirements arise for the Sign algorithm for security

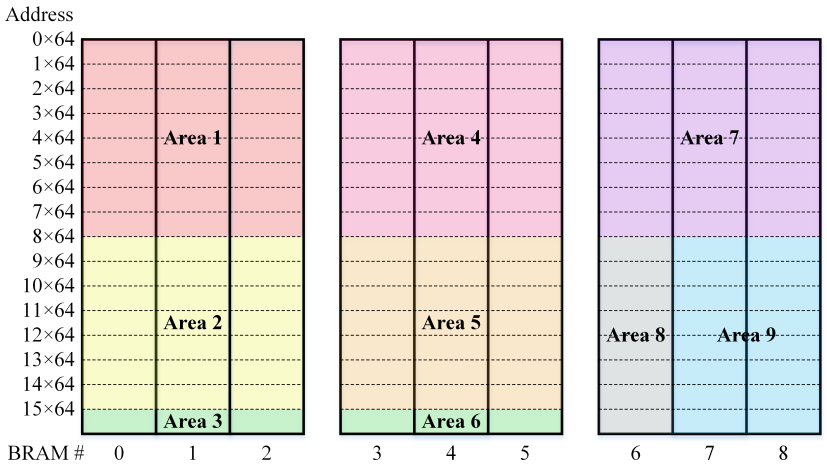


Figure 6: Structural arrangement of the BRAM array.

level 5, so this case determines the minimum number of BRAMs in our design. This section first introduces the two design considerations and then presents the storage scheme for the Sign algorithm for security level 5.

The first design consideration is that four coefficients should be stored in one address of three BRAMs. Since the bit width of the modulus q in Dilithium is 23 bits, the bit width of each coefficient that needs to be stored is also 23 bits. Meanwhile, the bit width of a BRAM is 36 bits. If each coefficient is straightforwardly stored in one address of one BRAM, 13 of the 36 bits will be wasted. Therefore, we use three BRAMs collectively as a group, and each address of the three BRAMs is used to store four coefficients of a polynomial so that only 16 of the corresponding 108 bits will be wasted. In this way, only 3 BRAMs are needed to store 16 polynomials, rather than 4 BRAMs in similar work [LSG21]. In addition, the NTT module can perform four pointwise multiplications per cycle, and this storage scheme exactly matches the corresponding reading and writing requirements.

The second design consideration is that the matrix $\boldsymbol{A}$ should be calculated on the fly instead of being precalculated. The reason is that the matrix $\boldsymbol{A}$ contains $7 \times 8 = 56$ polynomials for the highest security level. If $\boldsymbol{A}$ is precalculated and stored in BRAMs, 10.5 more BRAMs will be needed even with our improved storage method. Therefore, the matrix $\boldsymbol{A}$ is calculated on the fly, which reduces the overall storage requirements by half.

Finally, the specific storage scheme for the Sign algorithm for security level 5 is introduced as follows. The structural arrangement of our BRAM array is shown in Figure 6. One address of three adjacent BRAMs is used to store four coefficients. Every 64 addresses of the three BRAMs can store a complete polynomial. Thus, each group of three BRAMs can store 16 polynomials. Due to our segmented pipelined processing method, a large number of intermediate results do not need to be stored. Only the secret key and those intermediate results that will be used in later segments need to be stored. Specifically, $\hat{\boldsymbol{s}}_1$, $\hat{\boldsymbol{s}}_2$, and $\hat{\boldsymbol{t}}_0$ in the secret key need to be stored at all times, while the intermediate results $\hat{\boldsymbol{y}}$, $\hat{\boldsymbol{w}}$, $\boldsymbol{w}_0$, $\hat{c}$, $\boldsymbol{z}$, and $\boldsymbol{r}_0$ need to be stored for a certain period of time. In addition, the partial product polynomials produced during the matrix-vector multiplication $\hat{\boldsymbol{A}} \cdot \hat{\boldsymbol{y}}$ need to be temporarily stored. Some areas of the BRAMs also need to be used as shift registers for NTT/INTT processing, which is introduced in Section 4.1.

To make full use of the space and the ports of the BRAMs, the BRAM array is logically divided into several areas according to different uses, as shown in Figure 6. The arrangement of the storage location of each variable is mainly based on two considerations: the life spans and the port requirements. Areas 1, 2, 4, 5, and 7 are used to store polynomials with four coefficients in one address of three BRAMs. Area 8 is used in the SampleInBall module to generate and store c. Areas 3, 6, and 9 are used as shift registers in the NTT module.

4 Optimized Modules

This section introduces several optimized modules to achieve the goals of low resource consumption and high speed. These modules include the high-speed pipelined NTT module, which is used to accelerate the main operations in Dilithium; the BRAM-based SampleInBall module, which is used to run the Fisher-Yates shuffle algorithm efficiently; the compact Decompose module, which is used to perform the individualized function Decompose introduced in Section 2.3.1; the rejection sampling module, which is designed for high-speed on-the-fly generation of matrix $\boldsymbol{A}$; and three customized modular reduction modules, which are designed for the three moduli used in Dilithium.

4.1 NTT Module

The typical method used for hardware implementation of the NTT algorithm is to use butterfly units to perform layer-by-layer calculations in accordance with the butterfly diagram, as in [FS19, JGCS19, FSM+19, WTJ+20]. To accelerate the calculation of the NTT, some implementations use multiple butterfly units in parallel and calculate multiple butterfly operations in the same layer each time, as in [MOS19, ZYC+20, FSS20, XL20]. The disadvantage of this approach is that each butterfly unit needs to read two coefficients and write back two coefficients in each cycle, which means that k butterfly units need k times the number of memory ports. In addition, the memory access order is complex, necessitating a complex control module.

To accelerate the NTT algorithm without a large number of complex memory accesses, an optimized pipelined NTT structure is proposed in this paper, inspired by the R2MDC FFT structure introduced in Section 2.4. The R2MDC FFT architecture requires fewer and simpler memory accesses but is not suitable for direct use in Dilithium. First, the shift registers used to implement the delay units occupy up to $(64 + 32 + 16 + 8 + 4 + 2 + 1) \times 2 \times 23 = 5,842$ bits of FF resources. Although some modern FPGAs allow shift register implementation by certain LUTs (e.g., SRL in Xilinx FPGAs) to reduce FF consumption, FF-based SRs or SRL-based SRs update their entire states every cycle, leading to potentially high power consumption. Second, the utilization rate of eight butterfly units is only 50% when calculating pointwise multiplications, as our storage scheme cannot support reading sixteen coefficients and writing back eight coefficients per cycle. Third, the original R2MDC architecture allows data only to be input in normal order and output in bit-reversed order, which means that additional circuits and time are required for bit-reversal computation. Finally, eight layers require eight different twiddle

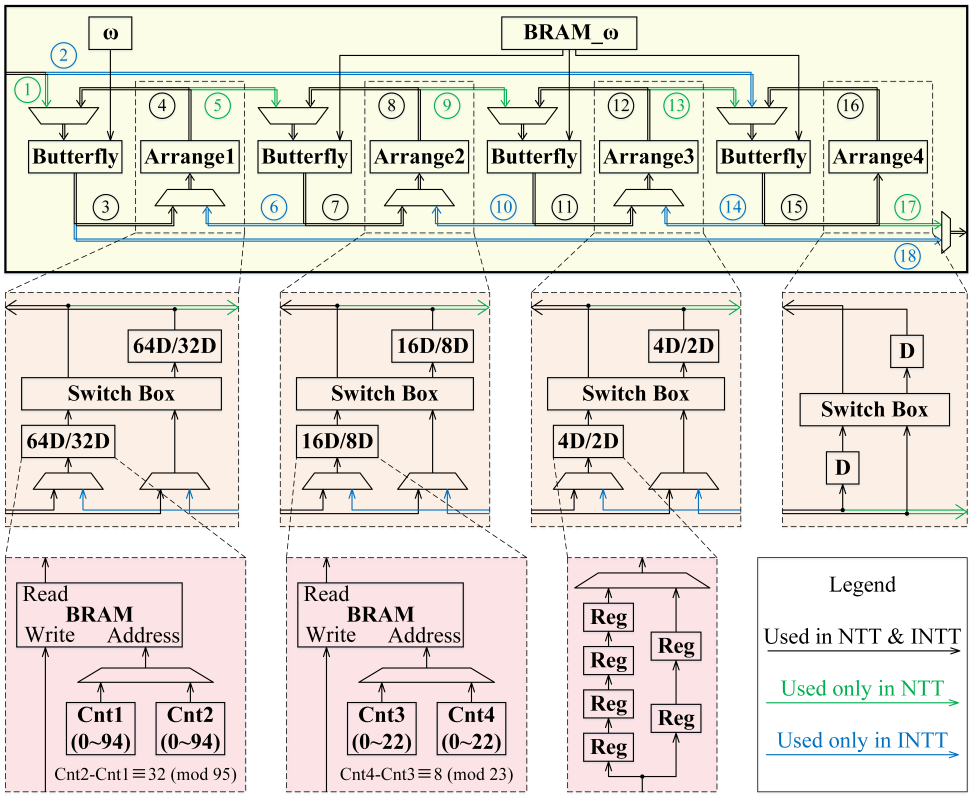


Figure 7: Block graph of the proposed NTT module.

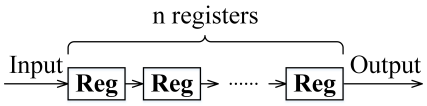
factors per cycle, which causes the original architecture to use eight memories for twiddle factors as shown in Figure 1.

The structure of the proposed NTT module, in which several improvements are made to mitigate the above shortcomings, is shown in Figure 7. The proposed module supports 256-point radix-2 DIT NTTs, 256-point radix-2 DIF INTTs, and 4-way parallel pointwise multiplications. This module uses four carefully designed butterfly units, a BRAM used to store precomputed twiddle factors, and some areas of the BRAM array in place of large shift registers. When calculating an NTT/INTT, the proposed module takes one coefficient as input and outputs one coefficient per cycle, and the delay from the beginning of the input to the beginning of the output is 296 cycles. The improvements are introduced in detail as follows.

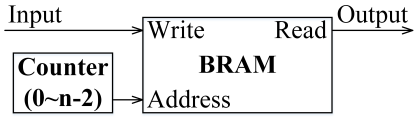
Replacing shift registers with BRAMs. The straightforward way to implement the delay units in R2MDC is to use shift registers, as shown in Figure 8(a). The proposed NTT module replaces the large shift registers with BRAMs and avoids additional BRAM consumption by using idle areas and ports of the BRAM array. A shift register with an n-cycle delay will store every input for n cycles and then output it, as shown in Figure 8(c). The proposed BRAM scheme uses a single-port BRAM and a counter to implement such a delay unit, as shown in Figure 8(b). The BRAMs used to replace the shift registers are embedded memory elements in an FPGA and are set to the read-first mode. In this mode, the input data on the write port will be stored at the input address in the next cycle, and the data previously stored at the input address will be output on the read port in the next cycle, as shown in the red and blue boxes in Figure 8(d). In addition, a counter with a period of n-1 cycles is used to provide addresses for the BRAM. For example, the input a_0 is stored at address 0 in the BRAM and is output after n cycles, as shown in Figure 8(d), the same behavior as the shift register scheme.

Folding transformation. The second improvement uses four butterfly units instead of eight by means of a method called folding transformation [PWB92]. In the folded structure, each butterfly unit calculates two adjacent NTT layers in a time-sliced fashion. For example, the first butterfly unit calculates the first layer in odd cycles and the second layer in even cycles. This improvement reduces the number of coefficients required in pointwise multiplications per cycle by half, and thus, the utilization rate of the butterfly units reaches 100% during parallel pointwise multiplications.

In addition, after folding transformation, the shift registers shift every two cycles



(a) Structure of the shift register scheme



(b) Structure of the BRAM scheme

Cycle	0	1	...	n	n+1	...
Counter						
Input	a_0	a_1	...	a_n	a_{n+1}	...
Output	n delay			a_0	a_1	...

(c) Timing diagram of the shift register scheme

Cycle	0	1	...	n-2	n-1	n	n+1	...
Counter	0	1	...	n-2	0	1	2	...
Input	a_0	a_1	...	a_{n-2}	a_{n-1}	a_n	a_{n+1}	...
Output					read	a_0	a_1	...
BRAM[0]	write a_0	a_0	a_0	a_0	a_0	a_{n-1}	a_{n-1}	...

(d) Timing diagram of the BRAM scheme

Figure 8: Two implementation schemes for an n-cycle delay unit.

Cycle	0	1	2	3	...	64	65	66	67	...	124	125	126	127	128	129	130	131	...
Counter 1	0	0	1	1	...	32	32	33	33	...	62	62	63	63	64	64	65	65	...
Counter 2	32	32	33	33	...	64	64	65	65	...	94	94	0	0	1	1	2	2	...
Input	a_0	b_{32}	a_1	b_{33}	...	a_{32}	b_{64}	a_{33}	b_{65}	...	a_{62}	b_{94}	a_{63}	b_0	a_{64}	b_1	a_{65}	b_2	...
Output							b_{32}		b_{33}	...		b_{62}		b_{63}	a_0	b_{64}	a_1	b_{65}	...

Legend : Address input of BRAM; Example of 64D; Example of 32D.

Figure 9: Timing diagram of the BRAM scheme for a 64D/32D unit. a_i denotes data passing through 64D, where i denotes the storage address of a_i in the BRAM. b_j denotes data passing through 32D, where j denotes the storage address of b_j in the BRAM.

and input/output data every two cycles. For example, the shift register delay, which is originally 64 cycles, changes to 128 cycles after folding transformation. However, it can store only 64 coefficients simultaneously, and each coefficient shifts only 64 times from being input to output, so it is still denoted by 64D in this paper. Because the reading and writing frequency is halved, a single-port BRAM can be used in place of two shift registers. For example, two shift registers 64D and 32D are replaced by one BRAM in Figure 7. The values of two counters are alternately used as the address input for the BRAM, with one counter for odd cycles and the other for even cycles. Both counters have $64 + 32 - 1 = 95$ possible values, i.e., the count range is 0 to 94. Their values increase by one every two cycles and always satisfy the condition $\text{Cnt2} - \text{Cnt1} \equiv 32 \pmod{95}$. The behavior of this scheme is illustrated in Figure 9 as an example. Accordingly, some unused storage space and four idle ports of the BRAM array are reused to replace $240 \times 23 = 5,520$ registers in the proposed module.

Supporting bit-reversed input. In addition to the naturally supported DIT NTT with input in normal order and output in bit-reversed order ($NTT^{DIT}_{no\to br}$), the proposed NTT module is also designed to support a DIF INTT with input in bit-reversed order and output in normal order ($INTT^{DIF}_{br\to no}$) through the addition of a new data flow path and the modification of the connection relationship of various processing blocks. For the naturally supported $NTT^{DIT}_{no\to br}$, the data flow passes through wires 1-3-4-3-5-7-8-7-9-11-12-11-13-15-16-17 in Figure 7, the same processing order as in the original R2MDC method. For the newly supported $INTT^{DIF}_{br\to no}$, new wires are added, as indicated in blue in Figure 7, and the corresponding data flow path is 2-15-16-14-12-11-12-10-8-7-8-6-4-3-4-18. With support for both $NTT^{DIT}_{no\to br}$ and $INTT^{DIF}_{br\to no}$, no additional bit-reversal computation is needed.

Reducing memory usage for twiddle factors. As mentioned above, the original R2MDC architecture uses eight memories for eight layers to provide eight different twiddle factors per cycle. In the proposed module, only one memory and three 23-bit registers are used to provide twiddle factors for eight layers. The twiddle factors used by the first butterfly unit, which calculates the first two layers of the NTT, have only $1 + 2$ values, so three 23-bit registers are used to store them. In addition, the other three butterfly units need three 23-bit twiddle factors per cycle. One dual-port 36k BRAM is used to store all the twiddle factors needed by the last three butterfly units, i.e., all twiddle factors used in the last six layers of the NTT. The two ports of this BRAM can provide $2 \times 36 = 72$ bits per cycle, in excess of the required $3 \times 23 = 69$ bits. Furthermore, the precomputed twiddle factors for $INTT^{DIF}_{br\to no}$ are simply the opposites of the values needed for $NTT^{DIT}_{no\to br}$, in reverse order. Thus, only the twiddle factors for $NTT^{DIT}_{no\to br}$ need to be stored. Their addresses in the BRAM are carefully arranged in accordance with their order of use so that no additional address calculation is required.

In addition, the proposed NTT module adopts the method in [ZYC+20] to merge the

preprocessing and postprocessing required for negative wrapped convolution (NWC) into the twiddle factors and to merge the postprocessing step of division by N in the INTT into the butterfly operations in each layer. Furthermore, modular reduction is performed after each multiplication in the butterfly units by using a modular reduction module designed specifically for the q in Dilithium, which is introduced in Section 4.5.

Overall, the proposed NTT module uses four butterfly units to accelerate NTT/INTT processing but needs to read only one coefficient and write only one coefficient per cycle. It involves much fewer and simpler memory accesses than the typical NTT accelerator. Compared to the original R2MDC architecture, our NTT module consumes considerably fewer register resources and uses only one dual-port 36k BRAM to store all of the involved twiddle factors.

4.2 SampleInBall Module

As mentioned in Section 2.3.3, the function SampleInBall is not friendly to implement in hardware due to data movements and data dependence. The work presented in [LSG21] used shift registers to generate and record the offsets of nonzero elements in c. However, this method needs more than five hundred registers for security level 5 and requires

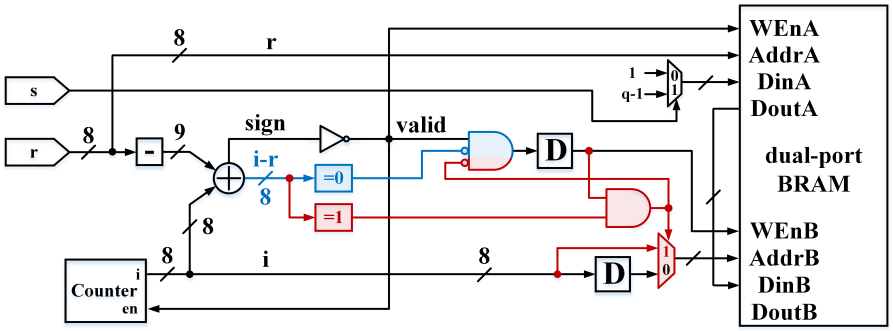


Figure 10: Structure of the SampleInBall module. The black, blue, and red components are used for processing in the basic case, the first special case, and the second special case, respectively.

Table 2: Timing diagram of the SampleInBall module in various cases.

	Basic			Case 1[a]			Case 1[b]			Case 2[a]		Case 2[b]		
Cycle	0	1	2	0	1	2	0	1	2	0	1	0	1	2
Counter	i	i+1	-	i	-	-	i	-	-	i	i+1	i	i+1	i+2
r	r	-	-	i	-	-	i	-	-	r'	i	r'	i	-
WEnA	1	-	-	1	-	-	1	-	-	1	1	1	1	-
AddrA	r	-	-	i	-	-	i	-	-	r'	i	r'	i	-
DinA	c_1	-	-	c_1	-	-	c_1	-	-	c_1	c_2	c_1	c_2	-
DoutA	-	c_0	-	-	0	-	-	0	-	-	c_0	-	c_0	0
WEnB	-	1	-	-	1	-	-	0	-	-	1	-	1	0
AddrB	-	i	-	-	i	-	-	i	-	-	i	-	i+1	i+1
DinB	-	c_0	-	-	0	-	-	0	-	-	c_0	-	c_0	0
BRAM[r]	c_0	c_1	-							BRAM[i+1]		0	0	c_0
BRAM[i]	0	0	c_0	0	c_1	0	0	c_1	c_1			0	0	c_2
BRAM[r']										c_0	c_1	c_0	c_1	c_1

[a] Error cases without special handling.
[b] Error cases with correct processing.

additional cycles to convert c into the standard format. This paper proposes a BRAM-based SampleInBall module which uses the BRAM to record the coefficients instead of using registers to record offsets. Therefore, it avoids the use of hundreds of registers and the need for additional format conversion.

The basic design idea of this module is shown by the black components in Figure 10. The module's input is a pseudorandom number generated by the KECCAK module, including a 1-bit sign bit s and an 8-bit random number r for rejection sampling. This module contains an 8-bit counter, whose value corresponds to the loop variable i in Algorithm 5. The input r is converted into its negative value and added to i (i.e., i minus r), and the sign bit of the addition result represents the rejection sampling result. If sampling is successful, the following operations will be performed. The value 1 or $q-1$, determined by s, will be written into the BRAM at address r through port A. The original value at address r will be read out through port A in the next cycle and then written to address i through port B. The counter will be incremented by one.

The BRAM used in this module is set to the read-first mode and can store the input data at the input address and read out the data previously stored at the input address in the next cycle. In addition to the basic case, two error cases may arise without special handling. The timing diagram for processing in the basic case and these two error cases is shown in Table 2. The first error case occurs when r is equal to i. In this case, the original value 0 at address i will be written back and will overwrite the correct data in the third cycle. The second error case occurs when r is equal to the counter's value minus one and sampling succeeded in the previous cycle. In this case, both ports will attempt to write to the same address in the second cycle. The hardware structures for handling these two error cases are marked in blue and red, respectively, in Figure 10. These structures will detect the occurrence of these two error cases and change the enable signal and the address of port B accordingly, as marked in green in Table 2.

In conclusion, the proposed SampleInBall module has the following advantages. First, it consumes negligible resources, because it reuses the free area in the BRAM array as its core part, and the logic of the remaining part is very simple. Second, the output c of this module is in standard polynomial format and can be directly subjected to NTT processing without additional format conversion. Third, straightforward serial processing would require one reading operation and two writing operations for each valid sample, corresponding to three clock cycles, whereas the proposed module needs only one cycle per sample on average.

4.3 Decompose Module

As mentioned in Section 2.3.1, the methods used in the reference software implementations of $\mathsf{Decompose}_q$ are too expensive for hardware implementation. The work presented in

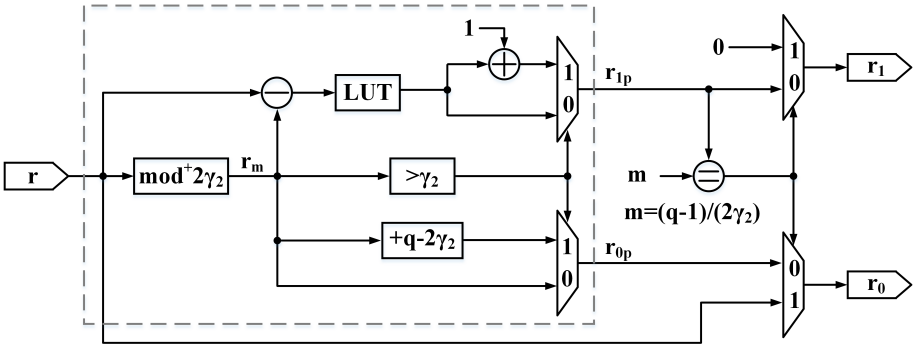


Figure 11: Structure of the proposed Decompose module.

Input r	0	1	...	γ_2	γ_2+1	...	$2\gamma_2$	...	$3\gamma_2$	$3\gamma_2$+1	...	$5\gamma_2$	...	q-γ_2	...	q-2	q-1
r_{0p}	0	1	...	γ_2	-γ_2+1	...	0	...	γ_2	-γ_2+1	...	γ_2	...	-γ_2+1	...	-1	0
r_{1p}	0	0	...	0	1	...	1	...	1	2	...	2	...	m	...	m	m
Output r_0	0	1	...	γ_2	-γ_2+1	...	0	...	γ_2	-γ_2+1	...	γ_2	...	-γ_2	...	-2	-1
Output r_1	0	0	...	0	1	...	1	...	1	2	...	2	...	0	...	0	0

Figure 12: Data correspondence for Decompose. Pink shading represents the border case.

[LSG21] used two large LUTs to obtain the high bits and $2\gamma_2$ times the high bits, respectively, followed by a subtraction to obtain the low bits, which is also costly. An efficient Decompose module is proposed in this paper to take full advantage of the capabilities of a hardware implementation, as shown in Figure 11.

The part of the structure inside the dashed box in Figure 11 is used to calculate $r_{0p} = r \bmod^{\pm} 2\gamma_2$ and $r_{1p} = (r - r_{0p})/2\gamma_2$. First, a specifically designed modular reduction module, which is introduced in Section 4.5, calculates $r_m = r \bmod^{+} 2\gamma_2$. Then, the remainder r_m is converted into a centralized remainder r_{0p} in $\mathbb{Z}_q$. To balance the delay on different paths, r_{1p} is obtained by calculating $(r - r_m)/2\gamma_2$, followed by a simple postprocessing step, instead of directly calculating $r_{1p} = (r - r_{0p})/2\gamma_2$.

The part of the structure outside the dashed box in Figure 11 is used to handle border cases. The border case is expressed as $r - r_0 = q - 1$ in the specification of this function and occurs when the input r is in the range $q - \gamma_2 \leq r \leq q - 1$, as shown in pink in Figure 12. It can be seen from Figure 12 that the border case occurs when $r_{1p} = m = (q - 1)/2\gamma_2$, which is used as the judgment condition for the border case in the proposed module. In addition, the calculation results according to the definition of Decompose show that r_0 is equal to $r - q$ in the border case, i.e., $r_0 \equiv r \bmod^{+} q$. Therefore, the proposed module processes the border case by setting r_1 to zero and setting r_0 to r.

In summary, the proposed Decompose module is the first manually designed hardware architecture for this function, which achieves both low latency and low resource consumption.

4.4 Rejection Sampling Module

The rejection sampling module is designed to rapidly sample coefficients of matrix $\boldsymbol{A}$ on the fly. The preceding stage is the KECCAK module performing SHAKE-128 to generate pseudorandom numbers for sampling. The sampled coefficients are sent to the NTT module for pointwise multiplications. As these three modules run in a pipelined manner, the speed of the rejection sampling module is designed to approximately match the speed of the other two modules.

When generating matrix $\boldsymbol{A}$ on the fly, the KECCAK module can generate 1344-bit

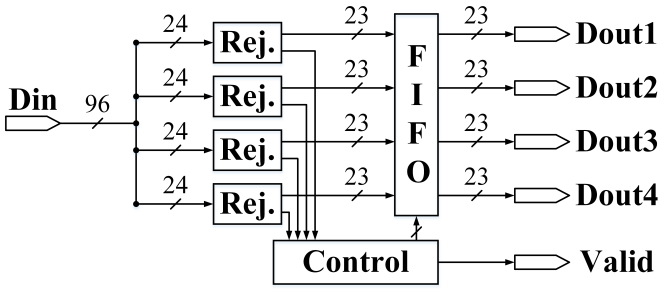


Figure 13: Block graph of the rejection sampling module.

pseudorandom numbers per 12 cycles. Every 24 bits are used for rejection sampling one coefficient. Therefore, the pseudorandom numbers generated every 12 cycles by the KECCAK module are used for 56 rejection samples. Thus, if the speed of the rejection sampling module is sampling $56/12 = 4.67$ times per cycle, it will ideally match the speed of the KECCAK module. The NTT module can perform pointwise multiplications for four successfully sampled coefficients of $\boldsymbol{A}$ per cycle, which means the ideal speed of the rejection sampling module is successfully sampling four coefficients per cycle.

To approximately match the speed without making the control logic too complicated, the rejection sampling module is designed to perform rejection sampling four times per cycle as a trade-off. As a result, the pseudorandom numbers generated by the KECCAK module in 12 cycles are processed by this module in $56/4 = 14$ cycles, i.e., the KECCAK module works for 12 cycles and waits for 2 cycles. On the output side, as the rejection sampling might fail, the NTT module waits until four successfully sampled coefficients are ready.

The block graph of the rejection sampling module is shown in Figure 13. The input $24 \times 4 = 96$ bits are processed by four parallel rejection sampling blocks. The successfully sampled coefficients are temporarily stored in the FIFO. When there are four or more coefficients in the FIFO, four coefficients are output and the valid signal is set to one. Additionally, this module is reusable for sampling $\boldsymbol{s}_1$ and $\boldsymbol{s}_2$ in KeyGen with some slight modifications.

4.5 Modular Reduction Modules

Three optimized modular reduction modules are proposed for the three moduli used in Dilithium. The optimization method is introduced as follows. First, the modulus is transformed into a canonical signed digit (CSD) representation [Har96]. This CSD representation is utilized to compress the bit width of the number to be reduced, and the compression process is repeated until the result is less than twice the modulus. A conditional subtraction is then performed after compression to obtain the final result.

The three moduli used in Dilithium are $q = 8380417$, $\alpha_1 = 2\gamma_{2,1} = (q-1)/44$, and $\alpha_2 = 2\gamma_{2,2} = (q-1)/16$. Their corresponding properties to be utilized for bit width compression are $2^{23} = 2^{13} - 1 \pmod{q}$, $2^{18} = 2^{16} + 2^{13} - 2^{11} \pmod{\alpha_1}$, and $2^{19} = 2^9 \pmod{\alpha_2}$, respectively. For example, a 23-bit number $r[22:0]$ can be compressed to $r[22:19] \cdot 2^9 + r[18:0]$ for modulus q. The overall structure of the reduction module for α_1 is shown in Figure 14 as an example, whose CSD representation is the most complicated.

5 Results and Comparison

The proposed design was simulated, synthesized, and implemented on a Xilinx Artix-7 FPGA (XC7Z020). All phases and all security levels of round 3 Dilithium are supported by the same hardware architecture.

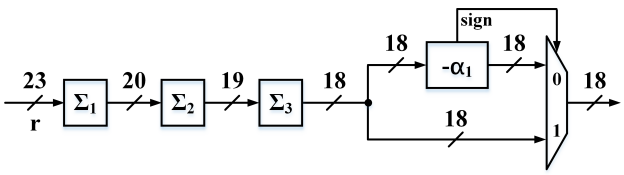


Figure 14: Block graph of the modular reduction module for modulus $(q-1)/44$.

5.1 Resources Usage and Performance Results

The resource consumption of the whole design and major modules is shown in Table 3. It can be seen that the KECCAK module occupies approximately 53% of the total LUTs and 44% of the total FFs. This is because the high-speed KECCAK module is used to calculate the matrix $\boldsymbol{A}$ on the fly and should match the speed of the 4-way parallel multiplication. The NTT module occupies approximately 6% of the LUTs, 13% of the FFs, and 8 DSPs. This consumption is relatively small for an NTT module that uses four butterfly units because our NTT module does not require complicated memory access and the large number of shift registers required by the original R2MDC structure is reduced by reusing BRAMs. In addition, because the BRAMs in the BRAM array are used in multiple modules, they are not included in the BRAM consumption of any single module in Table 3. Details of the BRAM usage are introduced in Section 3.3.

The key performance results of our implementation are presented in Table 4, with a maximum frequency of 96.9 MHz. All results in Table 4 were obtained based on 10,000 simulations. In the results for Verify, only simulations for valid signatures are included, as an invalid signature is processed much faster. Due to the nature of Dilithium, the number of cycles for Sign varies widely. Thus, multiple performance results are reported for Sign, including the minimum number of cycles without changing the key, the average number of cycles without changing the key, and the average number of cycles with new keys. In addition, because the message to be signed can be of any length, the clock cycle values listed in Table 4 do not include the time for message input.

To compare the different performance improvements brought by several proposed optimizations, we analyze the case of performing Sign without new keys for security level 5. Only the reduction in the number of cycles in the core loop is counted, which can be multiplied by the theoretical number of iterations 3.85 [LDK+21] to obtain the average improvements on the whole algorithm Sign. Without segmented pipelined processing, the loop takes approximately 24.5k cycles instead of 15.8k cycles, i.e., this strategy reduces about 8.7k cycles per loop. If the NTT module uses only one butterfly unit, each NTT takes 1024 cycles and the loop takes 47.3k cycles, i.e., the proposed NTT module reduces approximately 31.5k cycles. Compared to the method of reading or writing BRAM only once per cycle, the proposed SampleInBall module reduces 136 cycles per loop. The Decompose module reduces resource usage but has almost no impact on performance. Overall, the NTT module has the greatest impact on performance and the segmented pipelined processing method has the second greatest impact.

5.2 Comparison with Related Works

There are several existing FPGA implementations for Dilithium, including an HLS-based implementation [SBNK19], a high-performance implementation [RMJ+21], and a DSP-oriented implementation [LSG21]. In addition, [BUC19b] proposed an implementation for

Table 3: Resource consumption.

Module	LUTs	FFs	BRAMs	DSPs
KECCAK	15,784	4,557	0	0
SAMPLE	1,921	472	0	0
NTT	1,919	1,301	2	8
HEAD	293	187	0	2
TAIL	600	260	0	0
BRAM Array	274	0	9	0
Total	29,998	10,366	11	10

Table 4: Performance results.

Security Level	2		3		5	
Operation	Cycles	OP/s	Cycles	OP/s	Cycles	OP/s
$\mathsf{KeyGen}_{\text{avg}}$	4,172	23,217	5,851	16,555	8,765	11,051
$\mathsf{Sign}_{\text{min}}$ [a]	8,361	11,585	11,399	8,497	15,790	6,134
$\mathsf{Sign}_{\text{avg}}$ [a]	28,091	3,448	44,706	2,167	48,996	1,977
$\mathsf{Sign}_{\text{avg}}$ [b]	31,600	3,065	49,496	1,957	55,321	1,751
$\mathsf{Verify}_{\text{avg}}$	4,422	21,904	6,181	15,671	9,039	10,716

[a] Signing without changing the secret key.
[b] Signing with new secret keys.

several post-quantum lattice-based protocols including Dilithium based on a hardware-software codesign method. Compared to existing architectures, the proposed architecture in this work is mainly different in the following aspects. First, it is a dedicated pipelined architecture that minimizes the idle cycles of single modules by the proposed segmented pipelined processing method. Second, several modules are carefully designed and optimized for Dilithium in this work. Third, the use of BRAM space and ports is fully optimized, which reduces the BRAM consumption significantly.

A comparison of performances and resources with the related works is shown in Table 5. This work and [LSG21] implemented the round 3 version of Dilithium. The last three works [RMJ+21, SBNK19, BUC19b] were designed for the round 2 version of Dilithium, which changed substantially in round 3. Because the two lower security levels in round 2 did not exist in round 3, the corresponding data are not listed in Table 5. In addition, the numbers of cycles listed in Table 5 are average values and do not include the execution time for precalculation. When performing Sign with new keys (or Verify), our core unpacks the input keys (or signatures, respectively) and performs subsequent operations in a pipelined manner. The final results are stored in BRAMs, i.e., our core does not pack and send results as [LSG21]. Thus for a fair comparison, the execution time for packing and unpacking of [LSG21] is not considered in Table 5. Among pure hardware designs, the proposed design is the first to use an identical architecture to support all phases and all security levels of Dilithium. Compared with other implementations of round 3 Dilithium on the same device, this work uses the least resources to achieve the fastest speed.

[LSG21] proposed architectures for round 3 Dilithium on the same XC7Z020 FPGA device as ours; however, each architecture of [LSG21] supports only one security level. For all security levels, our design is at least 3.1×/1.6×/1.3× faster for KeyGen/Sign/Verify than the implementations in [LSG21]. There are two main reasons for our faster speed. First, the proposed segmented pipelined processing method reduces the execution time for a large number of operations. Second, our NTT module uses four butterfly units, twice the number in [LSG21], to accelerate the NTT/INTT and pointwise multiplication operations. Although our design supports all security levels, our LUT/FF consumption is only comparable to theirs for the two lower security levels and is only 70%/73% of their consumption for security level 5. In addition, our BRAM consumption is 73%/48%/33% of theirs for security level 2/3/5 because the segmented pipelined processing method reduces the number of intermediate results to be stored and our carefully designed storage scheme leads to higher utilization of the BRAMs. Since our NTT module is reused for multiply-accumulate operations instead of another dedicated module being used for this purpose, as in [LSG21], and the modular reduction calculations are performed by customized tiny modules in our design instead of by DSPs as in [LSG21], our DSP consumption is only 22% of theirs.

[RMJ+21] proposed high-speed hardware architectures for round 2 Dilithium on a Virtex-7 UltraScale+ FPGA. Different architectures are used for different phases, and the

Table 5: Comparison with related works. For each security level, the three rows of speed and resource results correspond to three phases: KeyGen, Sign and Verify.

Work (Device)	NIST Security Level	Speed Cycles	Speed OP/s	f MHz	Resources LUTs	Resources FFs	Resources BRAMs	Resources DSPs
This work (Artix-7)	2	4,172	23,217	96.9	29,998	10,366	11	10
		28,091	3,448					
		4,422	21,904					
	3	5,851	16,555					
		44,706	2,167					
		6,181	15,671					
	5	8,765	11,051					
		48,996	1,977					
		9,039	10,716					
[LSG21] (Artix-7)	2	18,761	7,462	140	24,320	9,668	15	45
		66,966	2,091					
		8,770	15,963					
	3	33,102	4,290	142	29,987	11,274	23	45
		105,129	1,351					
		12,084	11,751					
	5	50,982	2,491	127	42,860	14,136	33	45
		112,145	1,132					
		16,462	7,715					
[RMJ+21] (Virtex-7 UltraScale+)	2 [a]	18,193	19,238	350	54,183	25,236	15	182
		21,033	15,547	333	68,461	86,295	145	965
		15,032	10,524	158	61,738	34,963	18	316
	3 [a]	22,981	15,230	350	-	-	-	-
		22,362	14,265	319	-	-	-	-
		20,221	7,800	158	-	-	-	-
[SBNK19] (Artix-7)	2 [a]	233,420	512	119	86,646	17,674	-	-
		1,618,319	71	114	90,567	21,160	-	-
		285,100	401	114	65,274	15,169	-	-
	3 [a]	305,794	379	116	87,538	17,872	-	-
		1,106,053	103	114	91,605	21,322	-	-
		369,787	309	114	65,360	15,187	-	-
[BUC19b] (Artix-7)	2 [a]	167,433	149	25	14,975	2,539	14	11
		634,763	39					
		229,481	109					
	3 [a]	223,272	112					
		815,636	31					
		276,221	91					

[a] Round 2 parameters. In round 2, NIST security levels 2 and 3 correspond to Dilithium security levels 3 and 4, respectively.

resource consumption for NIST security level 2 only is reported in [RMJ+21]. Due to their straightforward implementation method, their implementations for KeyGen and Verify do not achieve good results. Even on higher-end devices, their speed is 1.2×/2.1× slower than ours for KeyGen/Verify for security level 2. Additionally, their resource consumption for only KeyGen or Verify is 1.8×/2.4×/1.4×/18× or 2.1×/3.4×/1.6×/32× greater than ours, respectively, when measured in terms of LUTs/FFs/BRAMs/DSPs. However, the implementation of [RMJ+21] can perform the Sign algorithm at a high speed, for two main reasons. First, Virtex-7 UltraScale+ is a high-performance and large-capacity FPGA series. Second, [RMJ+21] uses a straightforward method to implement every function in the loop of Sign and pipeline processes the whole loop. This method leads to high resource

consumption, especially in terms of BRAMs and DSPs. Their architecture for Sign at security level 2 uses 2.3× as many LUTs, 8.3× as many FFs, 13× as many BRAMs, and 97× as many DSPs as our architecture, which supports all phases and all security levels.

[SBNK19] evaluated round 2 Dilithium by means of an HLS-based method on an Artix-7 FPGA. Its speeds for KeyGen/Sign/Verify are 45×/49×/55× and 44×/21×/51× slower than ours for security levels 2 and 3, respectively. In addition, it requires 2.9×/3.0×/2.2× as many LUTs and 1.7×/2.0×/1.5× as many FFs as our architecture for KeyGen/Sign/Verify for both security levels 2 and 3. This large difference is probably due to our efficient architecture and the relative inefficiency of the HLS-based method.

[BUC19b] used a hardware-software codesign method to propose a configurable PQC accelerator that supports round 2 Dilithium. An NTT core, a Keccak core, and a sampler core were designed to accelerate the corresponding functions, and the accelerator needs to be coupled to a RISC-V processor to run the whole crypto algorithm. This is the reason why it requires only 50% of the LUTs and 24% of the FFs required by our implementation. However, most operations are performed serially and considerable time is wasted on data movements. Therefore, for KeyGen/Sign/Verify, this method is 156×/88×/201× and 148×/70×/172× slower than ours for security level 2 and 3, respectively. In addition, our BRAM/DSP consumption is 79%/91% as that of theirs for only the accelerator as a result of our careful arrangement and efficient storage scheme.

6 Conclusions and Future Work

This work presents a compact and high-performance hardware architecture for round 3 Dilithium that supports all three phases and three security levels. A segmented pipelined processing method is proposed to reduce the execution time of many operations and the storage requirements for many intermediate results. Several optimized modules are designed to use fewer resources while performing the corresponding functions faster, including a high-speed pipelined NTT module, a BRAM-based SampleInBall module, a compact Decompose module, and three optimized modular reduction modules. As a result, the proposed architecture uses 30k LUTs, 10k FFs, 11 BRAMs and 10 DSPs with $f_{\max} = 96.9$ MHz. For key generation, signature generation, and signature verification, our implementation can respectively perform 23,217, 3,448, and 21,904 OP/s for NIST security level 2; 16,555, 2,167, and 15,671 OP/s for NIST security level 3; and 11,051, 1,977, and 10,716 OP/s for NIST security level 5. Compared with state-of-the-art implementations of round 3 Dilithium, the proposed design uses 1.4×/1.4×/3.0×/4.5× fewer LUTs/FFs/BRAMs/DSPs to achieve 4.4×, 1.7×, and 1.4× faster calculation for key generation, signature generation, and signature verification, respectively, for security level 5. Our compact architecture makes it possible to accelerate Dilithium on resource-constrained devices while serving as a reference for algorithm evaluation.

From an application viewpoint, unprotected implementations of Dilithium face a potential threat of side-channel attacks [RJH+18, KLH+20, FDK20] and fault attacks [BP18, RRB+19]. In our future work, we will focus on investigating how to integrate countermeasures against side-channel attacks and fault attacks into the hardware architecture of Dilithium while prioritizing compact and high-performance.

Acknowledgments

This work is supported in part by the National Key R&D Program of China (Grant No. 2018YFB2202101), and in part by the National Science and Technology Major Project of the Ministry of Science and Technology of China (Grant No. 2018ZX01027101-002), and in part by the National Natural Science Foundation of China (Grant No. 61804088).

References

[ABB+19] Erdem Alkim, Paulo S. L. M. Barreto, Nina Bindel, Juliane Kramer, Patrick Longa, and Jefferson E. Ricardini. The lattice-based digital signature scheme qTESLA. Cryptology ePrint Archive, Report 2019/085, 2019. https://eprint.iacr.org/2019/085.

[Beu20] Ward Beullens. Improved cryptanalysis of UOV and rainbow. Cryptology ePrint Archive, Report 2020/1343, 2020. https://eprint.iacr.org/2020/1343.

[BG14] Shi Bai and Steven D. Galbraith. An improved compression technique for signatures based on learning with errors. In Josh Benaloh, editor, *CT-RSA 2014*, volume 8366 of *LNCS*, pages 28–47. Springer, Heidelberg, February 2014.

[BP18] Leon Groot Bruinderink and Peter Pessl. Differential fault attacks on deterministic lattice signatures. *IACR TCHES*, 2018(3):21–43, 2018. https://tches.iacr.org/index.php/TCHES/article/view/7267.

[BUC19a] Utsav Banerjee, Tenzin S. Ukyab, and Anantha P. Chandrakasan. Sapphire: A configurable crypto-processor for post-quantum lattice-based protocols. *IACR TCHES*, 2019(4):17–61, 2019. https://tches.iacr.org/index.php/TCHES/article/view/8344.

[BUC19b] Utsav Banerjee, Tenzin S. Ukyab, and Anantha P. Chandrakasan. Sapphire: A configurable crypto-processor for post-quantum lattice-based protocols (extended version). Cryptology ePrint Archive, Report 2019/1140, 2019. https://eprint.iacr.org/2019/1140.

[DCP+20] Jintai Ding, Ming-Shing Chen, Albrecht Petzoldt, Dieter Schmidt, Bo-Yin Yang, Matthias Kannwischer, and Jacques Patarin. Rainbow. Technical report, National Institute of Standards and Technology, 2020. available at https://csrc.nist.gov/projects/post-quantum-cryptography/round-3-submissions.

[Din20] Chengdong Tao Albrecht Petzoldt Jintai Ding. Improved key recovery of the HFEv- signature scheme. Cryptology ePrint Archive, Report 2020/1424, 2020. https://eprint.iacr.org/2020/1424.

[FDK20] Apostolos P. Fournaris, Charis Dimopoulos, and Odysseas G. Koufopavlou. Profiling dilithium digital signature traces for correlation differential side channel attacks. In Alex Orailoglu, Matthias Jung, and Marc Reichenbach, editors, *Embedded Computer Systems: Architectures, Modeling, and Simulation - 20th International Conference, SAMOS 2020, Samos, Greece, July 5-9, 2020, Proceedings*, volume 12471 of *Lecture Notes in Computer Science*, pages 281–294. Springer, 2020.

[FS19] Tim Fritzmann and Johanna Sepúlveda. Efficient and flexible low-power NTT for lattice-based cryptography. In *IEEE International Symposium on Hardware Oriented Security and Trust, HOST 2019, McLean, VA, USA, May 5-10, 2019*, pages 141–150. IEEE, 2019.

[FSM+19] Tim Fritzmann, Uzair Sharif, Daniel Müller-Gritschneder, Cezar Reinbrecht, Ulf Schlichtmann, and Johanna Sepúlveda. Towards reliable and secure post-quantum co-processors based on RISC-V. In Jürgen Teich and Franco Fummi,

editors, *Design, Automation & Test in Europe Conference & Exhibition, DATE 2019, Florence, Italy, March 25-29, 2019*, pages 1148–1153. IEEE, 2019.

[FSS20] Tim Fritzmann, Georg Sigl, and Johanna Sepúlveda. RISQ-V: Tightly coupled accelerators for post-quantum cryptography. *IACR TCHES*, 2020(4):239–280, 2020. https://tches.iacr.org/index.php/TCHES/article/view/8683.

[FY38] R. A. Fisher and F. Yates. Statistical tables for biological, agricultural and medical research. 1938.

[GKOS18] Tim Güneysu, Markus Krausz, Tobias Oder, and Julian Speith. Evaluation of lattice-based signature schemes in embedded systems. In *25th IEEE International Conference on Electronics, Circuits and Systems, ICECS 2018, Bordeaux, France, December 9-12, 2018*, pages 385–388. IEEE, 2018.

[GKS21] Denisa O. C. Greconici, Matthias J. Kannwischer, and Daan Sprenkels. Compact dilithium implementations on cortex-M3 and cortex-M4. *IACR TCHES*, 2021(1):1–24, 2021. https://tches.iacr.org/index.php/TCHES/article/view/8725.

[GLP12] Tim Güneysu, Vadim Lyubashevsky, and Thomas Pöppelmann. Practical lattice-based cryptography: A signature scheme for embedded systems. In Emmanuel Prouff and Patrick Schaumont, editors, *CHES 2012*, volume 7428 of *LNCS*, pages 530–547. Springer, Heidelberg, September 2012.

[Har96] R.I. Hartley. Subexpression sharing in filters using canonic signed digit multipliers. *IEEE Transactions on Circuits and Systems II: Analog and Digital Signal Processing*, 43(10):677–688, 1996.

[HT96] Shousheng He and Mats Torkelson. A new approach to pipeline FFT processor. In *Proceedings of IPPS '96, The 10th International Parallel Processing Symposium, April 15-19, 1996, Honolulu, Hawaii, USA*, pages 766–770. IEEE Computer Society, 1996.

[JGCS19] Arpan Jati, Naina Gupta, Anupam Chattopadhyay, and Somitra Kumar Sanadhya. SPQCop: Side-channel protected post-quantum cryptoprocessor. Cryptology ePrint Archive, Report 2019/765, 2019. https://eprint.iacr.org/2019/765.

[KLH+20] Il-Ju Kim, Tae-Ho Lee, Jaeseung Han, Bo-Yeon Sim, and Dong-Guk Han. Novel single-trace ML profiling attacks on NIST 3 round candidate dilithium. Cryptology ePrint Archive, Report 2020/1383, 2020. https://eprint.iacr.org/2020/1383.

[LDK+19] Vadim Lyubashevsky, Léo Ducas, Eike Kiltz, Tancrède Lepoint, Peter Schwabe, Gregor Seiler, Damien Stehlé, and Shi Bai. CRYSTALS-Dilithium submission package to nist for round 2, 2019. available at https://csrc.nist.gov/projects/post-quantum-cryptography/round-2-submissions.

[LDK+20a] Vadim Lyubashevsky, Léo Ducas, Eike Kiltz, Tancrède Lepoint, Peter Schwabe, Gregor Seiler, Damien Stehlé, and Shi Bai. CRYSTALS-DILITHIUM. Technical report, National Institute of Standards and Technology, 2020. available at https://csrc.nist.gov/projects/post-quantum-cryptography/round-3-submissions.

[LDK+20b] Vadim Lyubashevsky, Léo Ducas, Eike Kiltz, Tancrède Lepoint, Peter Schwabe, Gregor Seiler, Damien Stehlé, and Shi Bai. CRYSTALS-Dilithium submission package to nist for round 3, 2020. available at https://csrc.nist.gov/projects/post-quantum-cryptography/round-3-submissions.

[LDK+21] Vadim Lyubashevsky, Léo Ducas, Eike Kiltz, Tancrède Lepoint, Peter Schwabe, Gregor Seiler, Damien Stehlé, and Shi Bai. CRYSTALS-Dilithium – algorithm specifications and supporting documentation (version 3.1). Technical report, 2021. available at https://pq-crystals.org/dilithium/data/dilithium-specification-round3-20210208.pdf.

[LSG21] Georg Land, Pascal Sasdrich, and Tim Güneysu. A hard crystal - implementing dilithium on reconfigurable hardware. Cryptology ePrint Archive, Report 2021/355, 2021. https://eprint.iacr.org/2021/355.

[Lyu09] Vadim Lyubashevsky. Fiat-Shamir with aborts: Applications to lattice and factoring-based signatures. In Mitsuru Matsui, editor, *ASIACRYPT 2009*, volume 5912 of *LNCS*, pages 598–616. Springer, Heidelberg, December 2009.

[Lyu12] Vadim Lyubashevsky. Lattice signatures without trapdoors. In David Pointcheval and Thomas Johansson, editors, *EUROCRYPT 2012*, volume 7237 of *LNCS*, pages 738–755. Springer, Heidelberg, April 2012.

[MOS19] Ahmet Can Mert, Erdinc Ozturk, and Erkay Savas. Design and implementation of a fast and scalable NTT-based polynomial multiplier architecture. Cryptology ePrint Archive, Report 2019/109, 2019. https://eprint.iacr.org/2019/109.

[PFH+20] Thomas Prest, Pierre-Alain Fouque, Jeffrey Hoffstein, Paul Kirchner, Vadim Lyubashevsky, Thomas Pornin, Thomas Ricosset, Gregor Seiler, William Whyte, and Zhenfei Zhang. FALCON. Technical report, National Institute of Standards and Technology, 2020. available at https://csrc.nist.gov/projects/post-quantum-cryptography/round-3-submissions.

[PWB92] K.K. Parhi, C.-Y. Wang, and A.P. Brown. Synthesis of control circuits in folded pipelined dsp architectures. *IEEE Journal of Solid-State Circuits*, 27(1):29–43, 1992.

[RGCB19] Prasanna Ravi, Sourav Sen Gupta, Anupam Chattopadhyay, and Shivam Bhasin. Improving speed of dilithium's signing procedure. In Sonia Belaïd and Tim Güneysu, editors, *Smart Card Research and Advanced Applications - 18th International Conference, CARDIS 2019, Prague, Czech Republic, November 11-13, 2019, Revised Selected Papers*, volume 11833 of *Lecture Notes in Computer Science*, pages 57–73. Springer, 2019.

[RJH+18] Prasanna Ravi, Mahabir Prasad Jhanwar, James Howe, Anupam Chattopadhyay, and Shivam Bhasin. Side-channel assisted existential forgery attack on Dilithium - A NIST PQC candidate. Cryptology ePrint Archive, Report 2018/821, 2018. https://eprint.iacr.org/2018/821.

[RMJ+21] Sara Ricci, Lukas Malina, Petr Jedlicka, David Smekal, Jan Hajny, Petr Cibik, and Patrik Dobias. Implementing crystals-dilithium signature scheme on fpgas. Cryptology ePrint Archive, Report 2021/108, 2021. https://eprint.iacr.org/2021/108.

[RRB+19] Prasanna Ravi, Debapriya Basu Roy, Shivam Bhasin, Anupam Chattopadhyay, and Debdeep Mukhopadhyay. Number "not used" once - practical fault attack on pqm4 implementations of NIST candidates. In Ilia Polian and Marc Stöttinger, editors, *COSADE 2019*, volume 11421 of *LNCS*, pages 232–250. Springer, Heidelberg, April 2019.

[SBNK19] Deepraj Soni, Kanad Basu, Mohammed Nabeel, and Ramesh Karri. A hardware evaluation study of nist post-quantum cryptographic signature schemes. In *Second PQC Standardization Conference*. National Institute of Standards and Technology, 2019.

[Sho94] Peter W. Shor. Algorithms for quantum computation: Discrete logarithms and factoring. In *35th FOCS*, pages 124–134. IEEE Computer Society Press, November 1994.

[WTJ+20] Wen Wang, Shanquan Tian, Bernhard Jungk, Nina Bindel, Patrick Longa, and Jakub Szefer. Parameterized hardware accelerators for lattice-based cryptography. *IACR TCHES*, 2020(3):269–306, 2020. https://tches.iacr.org/index.php/TCHES/article/view/8591.

[XL20] Yufei Xing and Shuguo Li. An efficient implementation of the newhope key exchange on fpgas. *IEEE Trans. Circuits Syst. I Regul. Pap.*, 67-I(3):866–878, 2020.

[ZYC+20] Neng Zhang, Bohan Yang, Chen Chen, Shouyi Yin, Shaojun Wei, and Leibo Liu. Highly efficient architecture of NewHope-NIST. *IACR TCHES*, 2020(2):49–72, 2020. https://tches.iacr.org/index.php/TCHES/article/view/8544.