IACR Transactions on Cryptographic Hardware and Embedded Systems
ISSN 2569-2925, Vol. 2024, No. 3, pp. 99–135. DOI:10.46586/tches.v2024.i3.99-135

Optimized Hardware-Software Co-Design for Kyber and Dilithium on RISC-V SoC FPGA

Tengfei Wang[1,3], Chi Zhang[1,3,✉], Xiaolin Zhang[1,3], Dawu Gu[1,3,✉] and Pei Cao[2]

[1] School of Electronic Information and Electrical Engineering, Shanghai Jiao Tong University, Shanghai, China, {zcsjtu,dwgu}@sjtu.edu.cn
[2] Viewsource (Shanghai) Technology Company Limited, Shanghai, China
[3] State Key Laboratory of Cryptology, Beijing, China

Abstract. Kyber and Dilithium are both lattice-based post-quantum cryptography (PQC) algorithms that have been selected for standardization by the American National Institute of Standards and Technology (NIST). NIST recommends them as two primary algorithms to be implemented for most use cases. As the applications of RISC-V processors move from specialized scenarios to general scenarios, efficient implementations of PQC algorithms on general-purpose RISC-V platforms are required. In this work, we present an optimized hardware-software co-design for Kyber and Dilithium on the industry's first RISC-V System-on-Chip (SoC) Field Programmable Gate Array (FPGA) platform. The performance of both algorithms is enhanced through the utilization of hardware acceleration and software optimization, while a certain level of flexibility is still maintained. The polynomial arithmetic operations in Kyber and Dilithium are accelerated by the customized accelerators. We employ a unified high-level architecture to depict their shared characteristics and design dedicated underlying modular multipliers to explore their distinctive features. The hashing functions are optimized using RISC-V assembly instructions, resulting in improved performance and reduced code size without additional hardware resources. For other operations involving matrices and vectors, we present a multi-core acceleration scheme based on the multi-core RISC-V Microprocessor Sub-System (MSS). Combining these acceleration and optimization methods, experimental results show that the overall performance of Kyber and Dilithium across different security levels improves by 3 to 5 times, while the utilized FPGA resources account for less than 5% of the total resources provided by the platform.

Keywords: Post-quantum cryptography · RISC-V · Kyber · Dilithium · Hardware-software co-design · FPGA

1 Introduction

The rapid development of quantum computing technology poses serious challenges in cryptography. If a large-scale quantum computer is constructed in the future, it will be able to solve the hard mathematical problems used by classical public-key cryptography (PKC) such as RSA and Elliptic Curve Cryptography (ECC) in polynomial time [Sho94]. To ensure the security of PKC against both quantum and classical computers, the concept of post-quantum cryptography (PQC) has been raised in recent years. PQC refers to a new class of cryptographic algorithms that are designed to be resilient to attacks from quantum computers. Since 2016, the American National Institute of Standards and Technology (NIST) has been working on a process to solicit, evaluate, and standardize several PQC algorithms [NIS16]. After the third round of the PQC standardization process, NIST

Received: 2024-01-15 Accepted: 2024-03-15 Published: 2024-07-18

announced four candidate algorithms for standardization on July 5th, 2022. For most use cases, it is recommended to implement Kyber [BDK+18] and Dilithium [DKL+18] due to their strong security and excellent performance. The initial drafts of three Federal Information Processing Standards (FIPS) were published by NIST on August 24th, 2023. Among them, the draft FIPS 203 [oST23b] is derived from Kyber while the draft FIPS 204 [oST23a] is derived from Dilithium. This indicates that both algorithms will be widely used in the era of quantum computers.

Kyber and Dilithium are two cryptographic primitives included in the Cryptographic Suite for Algebraic Lattices (CRYSTALS) [Tea17]. Both of them are based on hard problems over module lattices. Kyber is an IND-CCA2-secure key-encapsulation mechanism (KEM), while Dilithium is a strongly EUF-CMA-secure digital signature scheme. From an implementation perspective, the two algorithms share many similarities in terms of mathematical structure and calculation process. Therefore, we can identify common software and hardware optimization methods for them. On the other hand, the unique features of each algorithm can also be explored to improve efficiency. These considerations make sense when both algorithms are implemented on the same platform.

Kyber and Dilithium can be implemented on various platforms to meet practical needs. To achieve optimal performance, a hardware design is often necessary, which can be implemented using an Application Specific Integrated Circuit (ASIC) [ZZZ+22] or a Field Programmable Gate Array (FPGA) [RMJ+21, XL21, ZZW+22] platform. However, this solution requires a longer development period and offers less flexibility compared to software implementation. If performance is not the primary consideration, then the software solution plays an important role in many cases. In the embedded scenario, ARM Cortex-M4 microcontrollers have traditionally been the main target platforms for PQC optimization [ABCG20, GKS21]. The ARM-based System-on-Chip (SoC) design offloads certain time-consuming operations to hardware, resulting in a significant acceleration effect [WZCG22, MCL+23]. In recent years, RISC-V processors have been receiving increasing attention due to their open-source nature and advanced Instruction Set Architecture (ISA) design concepts. Many works have developed ISA extensions for PQC based on open-source RISC-V processors [FSS20, AEL+20, FBR+22, ZXXH22, AY22, LMP23], demonstrating that application-specific instruction set processors (ASIP) can effectively balance performance and flexibility.

The ASIP is a type of hardware-software co-design that integrates tightly coupled accelerators with processors. Tightly coupled accelerators offer the advantage of low communication overhead, but they require modifications to the processor to support the extended ISA. On the contrary, loosely coupled accelerators are implemented outside processor cores, which provides convenience for integration and migration. Over the past few years, numerous mature RISC-V chips have entered the market and been applied in various scenarios, ranging from specialized to general. If a RISC-V hardcore is not specially designed for PQC while an efficient implementation of PQC on it is required, a loosely coupled accelerator is a favorable choice. For this reason, we select the industry's first RISC-V SoC FPGA named PolarFire[Mic], which is recommended by RISC-V International [Int], as the target platform to implement our hardware-software co-design for Kyber and Dilithium. A concise overview of this platform is provided in Subsection 2.3.

Related Works. Previous studies have presented numerous designs for implementing Kyber and Dilithium on RISC-V platforms. In [BUC19], a configurable lattice cryptography processor is presented. It is loosely coupled with a RISC-V microprocessor and supports the NIST Round-2 versions of Kyber and Dilithium. However, the authors in [FSS20] indicate that loosely coupled accelerators have disadvantages such as high data transfer overhead, a large amount of hardware resources, and low flexibility. Therefore, they propose tightly coupled accelerators with ISA extensions for polynomial operations, based on the open-source PULP[1] platform. At the same time, the work of [AEL+20] introduces a custom ISA extension for finite field arithmetic, which is based on the open-source VexRiscv[2] ecosystem. The design of [FSS20] outperforms that of [AEL+20] for Kyber due to more powerful hardware accelerators and extended instructions. In [XHY+20], the authors present a domain-specific vector processor integrated with an open-source 32-bit RISC-V MCU, SCR1[3]. The highly parallel hardware architecture significantly improves the performance of Round-2 Kyber. As Kyber and Dilithium enter the Round-3 competition and finally be standardized by NIST, research on the implementation of these two algorithms becomes more focused. In [NMZ+21], the authors introduce a dedicated Post-Quantum Arithmetic Logic Unit, embedded directly in the pipeline of 64-bit CVA6[4] RISC-V processor to speed up Kyber and Dilithium. A domain-specific processor with matrix extension of RISC-V for Kyber and Dilithium is presented in [ZXXH22]. This work achieves extremely high performance based on the Rocket RISC-V Core [AAB+16], but at a significant cost. The authors of [FBR+22] take the security against Differential Power Attacks (DPA) for Kyber into consideration, and propose masked accelerators and ISA extensions based on PULP. Their design combines loosely coupled accelerators for linear operations and tightly coupled accelerators for non-linear operations, which helps enhance performance and reduce code size. This design is used for reference by [KSFS22] to accelerate Dilithium, showing that the cycle counts can be significantly reduced. The authors of [AY22] provide a base architecture for developing ASIP for PQC. In [LMP23], the ASIP design for SHA-3 is realized. A loosely coupled SHA-3 accelerator is employed by [DMMM23] to speed up Kyber.

The related works mentioned above show that the hardware-software co-design based on RISC-V can efficiently speed up Kyber and Dilithium. However, there is no work evaluating its design for PQC on commercial general-purpose RISC-V platforms, even though such platforms have been available on the market for several years. The optimized implementation based on the general-purpose RISC-V hardcore with loosely coupled accelerators is meaningful for a wider application of PQC in the RISC-V ecosystem.

Contributions. In this work, we do not further explore ISA extensions for PQC on RISC-V. Instead, we focus on designing loosely coupled accelerators and doing our best to mitigate the negative effects that naturally come with them. Moreover, the software code running on general-purpose RISC-V processors is optimized using the standard ISA. Our code, both for hardware and software, is open-source and available at https://github.com/Acccrypto/RISC-V-SoC. We summarize our specific contributions as follows.

- We present customized loosely coupled accelerators for the polynomial arithmetic operations in Kyber and Dilithium, respectively. A unified high-level architecture is utilized to depict their shared characteristics. The dedicated modular multiplication units for the moduli used in Kyber and Dilithium are contained in the accelerators, optimizing the performance bottleneck of the underlying calculations.
- We design 4×1 butterfly units to speed up the calculations of Number Theoretic Transform (NTT) in Kyber and Dilithium. The hardware resources are reused to implement the polynomial base case multiplications and additions. Customized Finite State Machines (FSMs) are designed to control the data flow for different computational requirements of Kyber and Dilithium.
- We present several software optimizations for better use of the hardware accelerators. The Direct Memory Access (DMA) engine is utilized to improve the speed of data

[1]https://github.com/pulp-platform
[2]https://github.com/SpinalHDL/VexRiscv
[3]https://github.com/syntacore/scr1
[4]https://github.com/openhwgroup/cva6

transfer between the memory and the accelerator. The polynomials in matrices and vectors are sent to the accelerator in a carefully arranged order, aiming to avoid repetitive operations and data transfers. Compared with the baseline implementations, the accelerators can reduce the clock cycles of polynomial arithmetic operations by 70%-90%.

- We optimized SHA-3 functions employed by Kyber and Dilithium using assembly instructions based on RV64GC[5] ISA, without requiring any additional hardware resources. For other matrix and vector-related operations such as sampling and packing, we present a multi-core acceleration scheme based on the multi-core RISC-V Microprocessor Sub-System (MSS).
- We implement and evaluate our design on the industry's first multi-core RISC-V SoC FPGA platform. Experimental results show that our acceleration and optimization solutions can improve the overall performance of Kyber and Dilithium across different security levels by 3 to 5 times while utilizing less than 5% FPGA resources provided by the platform.

2 Preliminaries

2.1 Kyber and Dilithium

Both Kyber and Dilithium are constructed based on module lattices. A common security assumption of them is the hardness of the module-learning-with-errors (MLWE) [LS15] problem, which is described as follows.

The MLWE Problem. Let R_q denote a polynomial ring, where q is an integer. For integers m, k, and a probability distribution $D : R_q \to [0, 1]$, we say that the advantage of algorithm $\mathcal{A}$ in solving the decisional $\mathrm{MLWE}_{m,k,D}$ problem over the ring R_q is

$$\begin{aligned}\mathrm{Adv}^{\mathrm{MLWE}}_{m,k,D} = &|\Pr[b = 1 \mid \mathbf{A} \leftarrow R_q^{m\times k}; \mathbf{t} \leftarrow R_q^m; b \leftarrow \mathcal{A}(\mathbf{A}, \mathbf{t})] \\ &- \Pr[b = 1 \mid \mathbf{A} \leftarrow R_q^{m\times k}; \mathbf{s_1} \leftarrow D^k; \mathbf{s_2} \leftarrow D^m; b \leftarrow \mathcal{A}(\mathbf{A}, \mathbf{A}\mathbf{s_1} + \mathbf{s_2})]|.\end{aligned} \tag{1}$$

In the MLWE problem, the value of $\mathrm{Adv}^{\mathrm{MLWE}}_{m,k,D}$ is negligible for any known algorithm $\mathcal{A}$ with polynomial time complexity. In other words, when the parameters are sufficiently large, the adversary can't determine $(\mathbf{s_1}, \mathbf{s_2})$ according to $(\mathbf{A}, \mathbf{A}\mathbf{s_1} + \mathbf{s_2})$. Taking advantage of this hardness assumption, Kyber and Dilithium publicly disclose $(\mathbf{A}, \mathbf{A}\mathbf{s_1} + \mathbf{s_2})$ while keeping $(\mathbf{s_1}, \mathbf{s_2})$ confidential. The MLWE problem ensures theoretical security against attacks aimed at recovering the key or message.

Kyber. As an IND-CCA2-secure KEM, Kyber is constructed in two stages. First, an IND-CPA-secure public-key encryption scheme named Kyber.CPAPKE is introduced, which encrypts messages of a fixed length of 32 bytes. Second, the IND-CCA2-secure KEM named Kyber.CCAKEM is constructed by using a slightly tweaked Fujisaki-Okamoto (FO) transform [FO99]. The Kyber.CPAPKE comprises algorithms for key generation, encryption, and decryption. Let R_q denote the polynomial ring $\mathbb{Z}_q[X]/(X^n + 1)$, where $\mathbb{Z}_q$ is a finite field containing integers within the range of $[0, q)$.

- **Key generation**. The public key is generated through the operation $\mathbf{t} = \mathbf{A}\mathbf{s} + \mathbf{e} \in R_q^k$, where the matrix $\mathbf{A}$ is sampled uniformly from $R_q^{k\times k}$, and the secret vectors $\mathbf{s}, \mathbf{e} \in R_q^k$ are sampled from binomial distributions.

[5]RISC-V 64-bit ISA, where G=IMAFD

- **Encryption**. To encrypt a message m of 32 bytes, the encryption algorithm calculates $\mathbf{u} = \mathbf{A}^T\mathbf{r} + \mathbf{e_1} \in R_q^k$ and $v := \mathbf{t}^T\mathbf{r} + e_2 + \text{Decompress}_q(m, 1) \in R_q$, where $\mathbf{r}, \mathbf{e_1} \in R_q^k$, $e_2 \in R_q$ are sampled from binomial distributions. The function $\text{Decompress}_q(m, 1)$ converts m to an element in R_q.
- **Decryption**. The ciphertext $(\mathbf{u}, v)$ can be decrypted using the secret key $\mathbf{s}$. The decryption operation is $m := \text{Compress}_q(v - \mathbf{s}^T\mathbf{u}, 1)$, where the function Compress_q is the reverse of the function Decompress_q.

The Kyber.CCAKEM comprises algorithms for key generation, encapsulation, and decapsulation. It generates a shared key based on Kyber.CPAPKE and ensures that the procedure is IND-CCA2 secure via a slightly tweaked FO transform. More details can be found in Kyber's official documentation [ABD⁺b].

Kyber defines three parameter sets known as Kyber512, Kyber768, and Kyber1024, corresponding to different NIST security levels. Some of the parameters are listed in Table 1. Across the parameter sets, n and q are fixed values that determine the size of R_q. The module dimension k increases with higher security levels, leading to an expansion in data size. The remaining parameters are used in the sampling and compress functions.

Table 1: Parameter sets for Kyber

Parameter Set	NIST Security Level	n	q	k	η_1	η_2	(d_u, d_v)
Kyber512	1	256	3329	2	3	2	(10,4)
Kyber768	3	256	3329	3	2	2	(10,4)
Kyber1024	5	256	3329	4	2	2	(11,5)

Dilithium. The digital signature scheme Dilithium is designed based on the "Fiat-Shamir with Aborts" approach [Lyu09]. It comprises algorithms for key generation, signing, and verification. Similar to Kyber, the major operations of Dilithium are also conducted on the polynomial ring $\mathbb{Z}_q[X]/(X^n + 1)$ denoted by R_q. Let S_η denote the subset of R_q where the elements have small coefficients of size at most η.

- **Key generation**. The public key is calculated by the operation $\mathbf{t} = \mathbf{A}\mathbf{s_1} + \mathbf{s_2} \in R_q^k$, where the matrix $\mathbf{A} \in R_q^{k\times l}$ and the secret vectors $(\mathbf{s_1}, \mathbf{s_2}) \in S_\eta^l \times S_\eta^k$ are all generated using uniform sampling.
- **Signing**. The signing algorithm first generates a masking vector of polynomial $\mathbf{y} \in S_{\gamma_1-1}^l$. It then computes $\mathbf{Ay}$ and sets $\mathbf{w_1}$ to be the "high-order" bits of the coefficients in this vector. The challenge c is created by hashing the message and $\mathbf{w_1}$ and then sampled as a polynomial in R_q with exactly τ ±1's and the rest 0's. The potential signature is computed as $\mathbf{z} = \mathbf{y} + c\mathbf{s_1}$. To avoid the dependency of $\mathbf{z}$ on the secret key, Dilitium uses rejection sampling. A `while` loop keeps being repeated until all conditions are satisfied.
- **Verification**. The verification algorithm computes $\mathbf{Az} - c\mathbf{t}$ and sets $\mathbf{w_1'}$ to be the high-order bits of the result vector. It accepts if all the coefficients of $\mathbf{z}$ are less than $\gamma_1 - \beta$ and if c is the hash of the message and $\mathbf{w_1'}$.

To improve the efficiency, Dilithium introduces some "hints" as part of the signature. More details can be found in Dilithium's official documentation [BDK⁺].

Dilithium also defines three parameter sets for different NIST security levels. Some of the parameters are listed in Table 2. Across the parameter sets, n and q are fixed values that determine the size of R_q. The module dimension (k, l) increases with higher security levels, leading to an expansion in data size. The remaining parameters are used in the sampling and decompose functions.

Table 2: Parameter sets for Dilithium

NIST Security Level	n	q	(k,l)	d	τ	η	γ_1	γ_2	β
2	256	8380417	(4,4)	13	39	2	2^{17}	$(q-1)/88$	78
3	256	8380417	(6,5)	13	49	4	2^{19}	$(q-1)/32$	196
5	256	8380417	(8,7)	13	60	2	2^{19}	$(q-1)/32$	120

2.2 Basic Operations

The basic operations in Kyber and Dilithium have something in common. Some basic operations that have a great impact on performance are listed below.

Modular Arithmetic. Modular arithmetics on $\mathbb{Z}_q$ are fundamental operations in Kyber and Dilithium. They can be performed using ordinary arithmetics followed by modular reductions. For modular addition and subtraction, the modular reduction is usually implemented as a conditional subtraction and addition. For modular multiplication, the straightforward method of modular reduction is rather inefficient. Instead, Montgomery reduction [MP85] and Barrett reduction [Bar86] are two efficient algorithms commonly adopted by previous studies. However, both of these algorithms involve additional multiplications, which are time-consuming and require lots of hardware resources. Therefore, modular multiplication remains the performance bottleneck in the underlying calculations.

Number Theoretic Transform. The Number Theoretic Transform (NTT) is a generalization of the Fast Fourier Transform (FFT) over a finite field. By applying NTT to polynomial multiplication in R_q, the computational complexity can be reduced from $O(n^2)$ to $O(n\log(n))$. To calculate the multiplication of $a, b \in R_q$, the first step is to transform them into the NTT domain. This transformation is represented as $\hat{a} = \mathrm{NTT}(a) \in \mathbb{Z}_q^{256}$ and $\hat{b} = \mathrm{NTT}(b) \in \mathbb{Z}_q^{256}$. Next, the pointwise multiplication of $\hat{a}$ and $\hat{b}$ can be performed. Finally, the result can be transformed back to the regular domain using inverse NTT (INTT). The NTT-based polynomial multiplication can be represented as

$$c = \mathrm{INTT}(\mathrm{NTT}(a) \circ \mathrm{NTT}(b)), \tag{2}$$

where $a, b, c \in R_q$, and $\circ$ denote the pointwise multiplication operation.

To apply the NTT on the polynomial ring $R_q = \mathbb{Z}_q[X]/(X^n + 1)$, it requires that the primitive $2n$-th root of unity in the base field $\mathbb{Z}_q$ exists. In Dilithium, the parameters are chosen such that the $2n$-th root of unity is $r = 1753$. The definition of the NTT domain in Dilithium follows a bit-reversed order. That is,

$$\hat{a} = \mathrm{NTT}(a) = (a(r_0), a(-r_0), ..., a(r_{127}), a(-r_{127})), \tag{3}$$

where $r_i = r^{\mathrm{br}_8(128+i)}$ with $\mathrm{br}_j(k)$ the bitreversal of the j-bit number k.

In Kyber, the condition for NTT is not fully satisfied because $\mathbb{Z}_q$ contains primitive n-th roots of unity $\zeta = 17$ but not primitive $2n$-th roots. Therefore, Kyber employs an incomplete NTT to define the NTT domain. That is,

$$\hat{a} = \mathrm{NTT}(a) = (\hat{a}_0 + \hat{a}_1 X, \hat{a}_2 + \hat{a}_3 X, ..., \hat{a}_{254} + \hat{a}_{255} X) \tag{4}$$

with

$$\hat{a}_{2i} = \sum_{j=0}^{127} a_{2j} \zeta^{(2\mathrm{br}_7(i)+1)j}, \tag{5}$$

$$\hat{a}_{2i+1} = \sum_{j=0}^{127} a_{2j+1} \zeta^{(2\mathrm{br}_7(i)+1)j}. \tag{6}$$

Under this definition, the base case multiplication denoted by $\hat{c} = \hat{a} \circ \hat{b}$ should not follow the pointwise manner. Instead, it consists of the 128 products

$$\hat{c}_{2i} + \hat{c}_{2i+1}X = (\hat{a}_{2i} + \hat{a}_{2i+1}X)(\hat{b}_{2i} + \hat{b}_{2i+1}X) \mod X^2 - \zeta^{2\mathrm{br}_7(i)+1} \tag{7}$$

of linear polynomials.

Hashing. Kyber and Dilithium employ the SHA-3 [NIS15] family of functions to generate message digests, derived keys, and pseudorandom bits. The SHA-3 family consists of four cryptographic hash functions, called SHA3-224, SHA3-256, SHA3-384, and SHA3-512, and two extendable-output functions (XOFs), called SHAKE128 and SHAKE256. Kyber instantiates two hash functions with SHA3-256 and SHA3-512, a pseudorandom function with SHAKE256, an XOF with SHAKE128, and a key-derivation function with SHAKE256. Dilithium uses SHAKE128 and SHAKE256 to instantiate hash functions and XOFs. The SHA-3 functions are based on the same underlying permutation KECCAK-f[1600] with a state size of 1600 bits. The differences among them depend on the parameters of the sponge structure, including the padding rule, rate, and output length.

Sampling. Kyber and Dilithium use sampling functions to generate random polynomials from the output strings of XOFs. In Kyber, the polynomial matrix $\mathbf{A}$ is generated through uniform sampling, while the noise is sampled from a centered binomial distribution. In Dilithium, the polynomial matrix $\mathbf{A}$ and vectors $\mathbf{s_1}, \mathbf{s_2}, \mathbf{y}$ are all generated through uniform sampling. Due to its uniform random property, the matrix $\mathbf{A}$ can be viewed as a representation in the NTT domain. Therefore, some expensive transformations can be avoided. Binomial sampling can be performed by calculating the difference in Hamming weight between two random numbers.

2.3 RISC-V SoC FPGA

The target hardware platform used in this work is the industry's first RISC-V SoC FPGA named PolarFire [Mic], which is produced by Microchip. It integrates a powerful 64-bit 5x core RISC-V Microprocessor Sub-System (MSS) with the FPGA fabric in a single device. The MSS combines a SiFive E51 monitor core and four SiFive U54 application cores. The RISC-V cores are all implemented as ASICs, offering high performance and software programmability. The FPGA fabric offers scalable and programmable hardware features, making it an ideal device for implementing our hardware accelerators.

The E51 monitor core supports RV64IMAC ISA, which includes the base integer instruction set, as well as the standard extensions for integer multiplication and division, atomic, and compressed instructions. The U54 application cores support RV64GC ISA, which means that they add standard extensions for single-precision and double-precision floating-point instructions. All the cores are executed with a single-issue, in-order, 5-stage pipeline. They have an exclusive L1 cache and a shared L2 cache. The AXI4 bus interfaces are provided to enable the microprocessors to access main memory, peripherals, and FPGA fabric. The DMA transfer with four independent channels is supported.

The FPGA fabric is built on state-of-the-art 28 nm non-volatile process technology. Its resources can be categorized into three types: logic elements, embedded memory blocks, and math blocks. The logic element consists of a 4-input Lookup Table (4LUT) and a D-type flip-flop (DFF). The embedded memory block involves LSRAM (RAM1K20) and uSRAM (RAM64×12) with different memory sizes. The math block supports multipliers up to a maximum of 18×18 bits for signed multiplication and 17×17 bits for unsigned multiplication.

The development tools recommended by Microchip include Libero SoC for FPGA designers and SoftConsole for embedded designers. We use them to implement and evaluate our hardware-software co-design for Kyber and Dilithium on the PolarFire SoC FPGA platform.

3 Design Rationale

The PolarFire SoC FPGA platform provides the conditions for FPGA acceleration, execution of RISC-V assembly instructions, and multi-core acceleration. Our design takes advantage of these conditions to speed up the implementations of Kyber and Dilithium. The architecture of the RISC-V MSS, integrated with the hardware accelerator and other components, is depicted in Figure 1. It contains five RISC-V microprocessors, including one E51 monitor core and four U54 application cores. The E51 core is used to boot the program and monitor user applications on U54 cores. The U54 cores named U54_1 to U54_4 are used to execute the bare-metal user applications. The *TileLink* bus provides the microprocessors with coherent access to *L2 Cache*. The *L2 Cache* can also be configured as *Scratchpad Memory* to improve the memory access speed. The *AXI SWITCH* enables the CPU Core Complex to access the Double Data Rate (DDR) memory, peripherals, and FPGA fabric. Based on this architecture, we present a hardware-software co-design that includes hardware acceleration and software optimization to enhance the performance of Kyber and Dilithium.

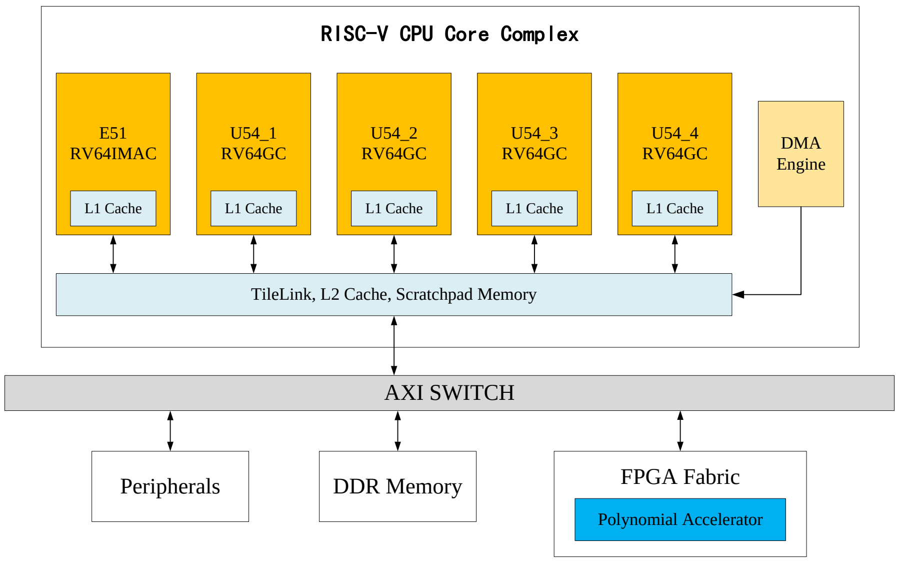

Figure 1: RISC-V Microprocessor Sub-System Integrated with Hardware Accelerator

Since polynomial arithmetic operations are relatively independent of other operations and time-consuming in software implementation, we decided to use FPGA to accelerate this part of the calculations. As shown in Figure 1, the *Polynomial Accelerator* module is designed and implemented on the FPGA Fabric. To mitigate the disadvantages of loosely coupled accelerators, we make the following design decisions. Firstly, to decrease the data transfer overhead, the accelerators do not output the results until the entire process of polynomial matrix-vector multiplication and addition is completed. Secondly, the hardware resources are optimized by using an efficient architecture for NTT-based polynomial multiplication. Thirdly, the accelerator does not perform other operations such as hashing and sampling so that it can be compact and make the overall design maintain a certain level of flexibility. The NTT functions play a crucial role in efficiently calculating polynomial multiplication for many lattice-based cryptography schemes. According to our test, the NTT-related operations account for 23%-59% of the computation time in the software implementation of Kyber and Dilithium. In this work, we focus our main efforts on improving them through hardware because it is both representative and cost-effective. In addition, NTT is vulnerable to side-channel analysis (SCA). We also introduce some typical countermeasures in our design and evaluate their costs.

The software optimization involves three parts. Firstly, when working with the *Polynomial Accelerator*, the *DMA Engine* is employed and the data transmission sequence is carefully arranged. Secondly, the SHA-3 functions running on U54 processors are optimized using RISC-V assembly instructions. This part of optimization only requires a processor that supports the ISA standard extensions of “D” and “C”, instead of a specific RISC-V design. We decided not to use hardware to accelerate SHA-3 because it is less efficient compared to accelerating NTT-based functions. Thirdly, other computational tasks for polynomial matrices and vectors such as sampling and packing can be assigned to multiple cores. Since the E51 core does not support the “D” extension, we only use the four U54 cores to implement the multi-core acceleration. By combining software optimization and hardware acceleration, an efficient hardware-software co-design for Kyber and Dilithium can be realized on the PolarFire SoC FPGA platform. More details are provided in Section 4 and Section 5.

Although our design is specifically proposed for Kyber and Dilithium, it can be extended to other lattice schemes with certain modifications. Software optimization is more generally applicable compared to hardware acceleration. For example, the FrodoKEM [ABD$^+$a] scheme incorporates SHA-3, sampling, and matrix-vector products as its main operations. Our software optimization for SHA-3 can be directly applied to it. The sampling functions it uses can also be assigned to multiple cores in a similar way to ours. As the matrix-vector product utilizes a power-of-two integer modulus and does not need NTT, the modular multiplier and NTT-related states inside the hardware accelerator should be modified, while DMA and data transmission sequence can be maintained.

4 Hardware Acceleration

4.1 Modular Multiplier

The modular arithmetic operations are performed by the modular arithmetic units at the underlying level of the accelerators. The modular arithmetic units consist of modular adders, modular subtractors, and modular multipliers. The design of modular adders and subtractors refers to [BUC19]. For modular multipliers, we propose dedicated designs for Kyber and Dilithium, respectively, based on the features of their moduli.

4.1.1 Dedicated Modular Multiplier for Kyber

In Kyber, the prime $q = 13 \cdot 2^8 + 1 = 3329$ is used as the modulus of $\mathbb{Z}_q$. Dedicated modular multipliers for this prime have been designed in several previous studies. The work of [LN16] presents a fast modular reduction technique named KRED, which is tailored for moduli with a special form of $k \cdot 2^m + 1$. This technique is adopted and modified by [BAK21] to design a dedicated modular multiplier for Kyber. In our design, we present a dedicated Montgomery modular multiplier specifically optimized for $q = 3329$. The optimization is based on the fact that if the parameter R of Montgomery modular multiplication is selected as 2^{12}, then the factor $w = q^{-1} \mod R$ equals $2^9 + 2^8 + 1 = 769$. We take advantage of the binary representations of q and w to replace the multiplications in the Montgomery reduction algorithm with shift and addition operations. The modified Montgomery reduction algorithm is shown in Algorithm 1.

Algorithm 1 Modified Montgomery Reduction for Modulus $q = 3329$

```
Require: $q = 3329$, $R = 2^{12}$, $w = q^{-1} \mod R = 769$, $m \in [0, 2q)$
Ensure: $z = mR^{-1} \mod q$
1: $u \leftarrow ((m \ll 9) + (m \ll 8) + m) \mod R$        ▷ $u \leftarrow m \cdot w \mod R$
2: $t \leftarrow (u \ll 11) + (u \ll 10) + (u \ll 8) + u$        ▷ $t \leftarrow u \cdot q$
3: $v \leftarrow (m - t)/R$
4: if v $< 0$ then
5:     $z \leftarrow v + q$
6: else
7:     $z \leftarrow v$
8: end if
9: return $z$
```

According to Algorithm 1, the dedicated modular multiplier for Kyber is designed as depicted in Figure 2. The procedure of modular multiplication is divided into three pipeline stages. The shift and summation operations in the first two steps of Algorithm 1 are optimized by using bitwise splicing. The summation units do not need to handle the least significant 8 bits of the operands, making them lightweight and having a short logic delay. The critical path of this multiplier lies in the last stage of the pipeline. The FPGA resources utilized by the modular multiplier are listed in Table 3. For comparison, the synthesis result of our design targeting Artix-7 FPGA is also listed. Unlike Polarfire, the Artix-7 FPGA employs 6-input LUT (6LUT) as logic elements and digital signal processing (DSP) slices as multipliers. The result shows that our design utilizes a similar amount of resources as [BAK21]. However, our design ensures that the output falls within the range of $[0, q)$, whereas in [BAK21], the output can only be guaranteed to have a 12-bit width.

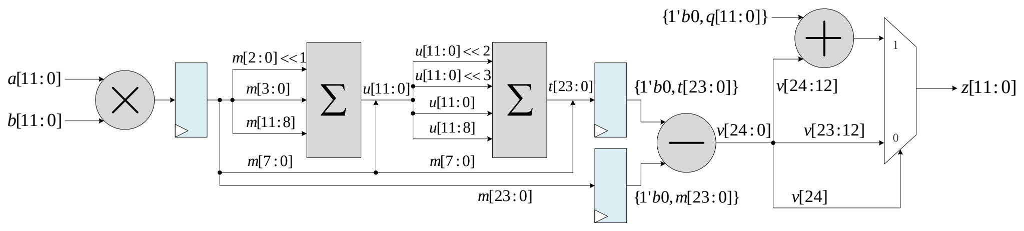

Figure 2: Dedicated Modular Multiplier for Kyber

Table 3: FPGA resources utilized by modular multipliers for Kyber

Reduction Algorithm	Platform	4LUT	DFF	Math	6LUT	FF	DSP
Algorithm 1	PolarFire	112	60	1	-	-	-
	Artix-7	-	-	-	53	32	1
K^2-RED[BAK21][1]	Artix-7	-	-	-	54	30	0

[1] This does not include the multiplication before reduction.

4.1.2 Dedicated Modular Multiplier for Dilithium

Dilithium uses the prime $q = 2^{23} - 2^{13} + 1 = 8380417$ as the modulus of $\mathbb{Z}_q$. Several studies present dedicated modular multipliers for this prime. In [BUC19], the multiplications in Barrett reduction for the modulus are converted to bit-shifts, additions, and subtractions, but it requires long carry chains to perform these operations. The authors of [LSG21] propose a fast reduction method by recursively exploiting the relationship $2^{23} \equiv 2^{13} - 1 \mod q$. However, the range of the result is not explained strictly, which has been pointed out and addressed by [WZCG22]. In this work, we follow the reduction method of [WZCG22] with some modifications. The modular multiplier in our design is slightly different from that of the Zynq-7000 FPGA because we are targeting the PolarFire FPGA which employs 18×18-bit math blocks for multiplications instead of 25×18-bit DSPs. Therefore, we first divide both 23-bit operands $a, b \in \mathbb{Z}_q$ into their high 11 bits (a_h, b_h) and low 12 bits (a_l, b_l). Next, the schoolbook multiplication method is applied, that is,

$$a \cdot b = 2^{24} \cdot a_h b_h + 2^{12} \cdot (a_h b_l + a_l b_h) + a_l b_l. \tag{8}$$

Then, we take advantage of the relationship $2^{23} \equiv 2^{13} - 1 \mod q$ to reduce the separate items in Equation 8. It should be noted that $2^{24} \cdot a_h b_h + a_l b_l$ can be treated as one item by concatenating them bitwise. Finally, the reduced items are added up and further reduced. We use the syntax of Hardware Description Language (HDL) to represent the operations of splitting and concatenation. The process of the dedicated modular multiplication is described in Algorithm 2. According to the bit width, the value r_3 in step 12 falls within the range of $[-(2^{13} - 1) \cdot 16, 2^{23} + 2^{13} \cdot 15 - 16]$, which is contained within the interval $(-q, 2q)$. Therefore, by using a conditional addition or subtraction at the end, it can be ensured that the final result falls within the range of $[0, q)$.

According to Algorithm 2, the dedicated modular multiplier for Dilithium is designed as depicted in Figure 3. Similar to Figure 2, this modular multiplier also divides the procedure into three pipeline stages. The critical path lies in the second stage of the pipeline. The FPGA resources utilized by this modular multiplier are listed in Table 4. For comparison with [WZCG22], the synthesis result targeting Zynq-7000 FPGA is also provided. Since the authors of [BUC19] did not give detailed utilization information, we re-implement the modular multiplier on the PolarFire FPGA using their method. It can be observed that our design utilizes fewer LUT and FF resources compared to the design in [BUC19]. The design of [WZCG22] utilizes fewer DSPs because it is customized for 25×18-bit DSPs instead of 18×18-bit Math Blocks.

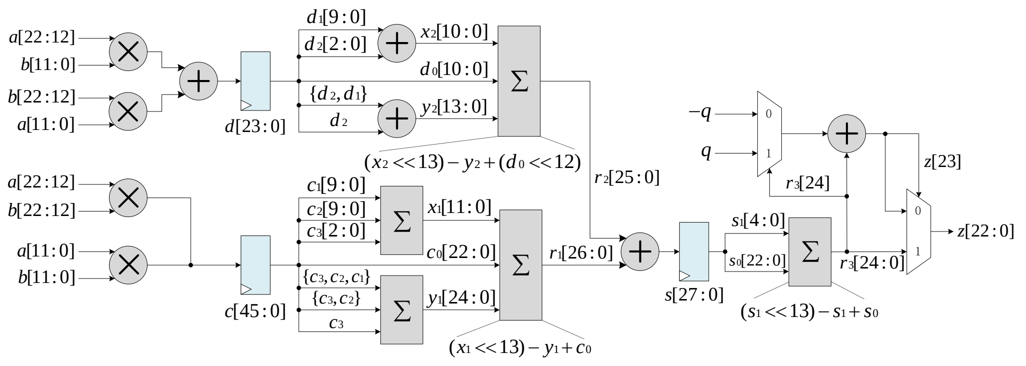

Figure 3: Dedicated Modular Multiplier for Dilithium

Algorithm 2 Dedicated Modular Multiplication for Modulus $q = 8380417$

Require: $q = 8380417,\ a = 2^{12} \cdot a_h + a_l \in [0, q),\ b = 2^{12} \cdot b_h + b_l \in [0, q)$, where $0 \le a_h, b_h < 2^{11}$, $0 \le a_l, b_l < 2^{12}$

Ensure: $z = ab \mod q$

```
1: $m_1 \leftarrow a_h b_h,\ m_2 \leftarrow a_l b_l,\ m_3 \leftarrow a_h b_l,\ m_4 \leftarrow a_l b_h$
2: $c \leftarrow \{m_1, m_2\},\ d \leftarrow m_3 + m_4$
3: $c_3 \leftarrow c[45:43],\ c_2 \leftarrow c[42:33],\ c_1 \leftarrow c[32:23],\ c_0 \leftarrow c[22:0]$
4: $d_2 \leftarrow d[23:21],\ d_1 \leftarrow d[20:11],\ d_0 \leftarrow d[10:0]$
5: $x_1 \leftarrow c_3 + c_2 + c_1,\ y_1 \leftarrow c_3 + \{c_3, c_2\} + \{c_3, c_2, c_1\}$
6: $x_2 \leftarrow d_2 + d_1,\ y_2 \leftarrow d_2 + \{d_2, d_1\}$
7: $r_1 \leftarrow (x_1 \ll 13) - y_1 + c_0$
8: $r_2 \leftarrow (x_2 \ll 13) - y_2 + (d_0 \ll 12)$
9: $s \leftarrow r_1 + r_2$
10: $s_1 \leftarrow s[27:23],\ s_0 \leftarrow s[22:0]$
11: $r_3 \leftarrow (s_1 \ll 13) - s_1 + s_0$
12: if $r_3 < 0$ then
13:     $z \leftarrow r_3 + q$
14: else if $r_3 \ge q$ then
15:     $z \leftarrow r_3 - q$
16: else
17:     $z \leftarrow r_3$
18: end if
19: return $z$
```

4.2 NTT Design

A commonly used method for implementing NTT and INTT is to employ the Cooley-Tukey (CT) butterfly for NTT and the Gentleman-Sande (GS) butterfly for INTT [LN16]. This method is efficient because it can avoid the cost of bit reversal operations. For hardware implementations, multiple butterfly units can be adopted to accelerate the algorithms through parallel computation. The authors of [CYY+22] propose a framework for the design of radix-2/4 NTT with various numbers of butterfly units. In this work, we follow the design rationale of [CYY+22] and present an architecture for radix-2 NTT/INTT with 4×1 butterfly units. The high-level architecture is unified for both Kyber and Dilithium, with dedicated underlying modular multipliers. In the following, the same symbols are used to denote the modular multiplier and other modular arithmetic units for Kyber and Dilithium since they have the same number of pipeline stages.

Table 4: FPGA resources utilized by modular multipliers for Dilithium

Reduction Algorithm	Platform	4LUT	DFF	Math	6LUT	FF	DSP
Algorithm 2	PolarFire	463	172	4	-	-	-
	Zynq-7000	-	-	-	212	28	4
Barrett [BUC19][1]	PolarFire	519	240	4	-	-	-
Dedicated [WZCG22]	Zynq-7000	-	-	-	260	36	2

[1] Our estimation by re-implementing this work on the PolarFire FPGA.

4.2.1 Radix-2 NTT/INTT with 4×1 Butterfly Units

The work of [BAK21] implements a 2×2 butterfly core to merge two layers of NTT/INTT and perform two butterfly operations in each layer. However, it is not the optimal solution for Kyber because the NTT/INTT has an odd number of layers. To make full use of hardware resources, our design employs 4×1 butterfly units to perform four butterfly operations in each layer without merging. The butterfly operations are defined as

$$\begin{aligned} &\mathrm{BF\_CT}(a,b,\omega) = (a + b\omega \mod q,\ a - b\omega \mod q), \\ &\mathrm{BF\_GS}(a,b,\omega) = (2^{-1}(a+b) \mod q,\ 2^{-1}(a-b)\omega \mod q), \end{aligned} \tag{9}$$

where BF_GS is used in NTT and BF_GS is used in INTT. The simple multiplications with 2^{-1} are adopted in BF_GS to eliminate the post-processing with n^{-1} [ZYC+20]. Based on modular arithmetic units, the data paths for the CT butterfly and GS butterfly are depicted as Figure 4. They are both divided into four pipeline stages. The modular arithmetic units and registers are shared by both data paths in the actual configurable butterfly circuit. Four identical butterfly units are instantiated in our design to accelerate the NTT and INTT operations. The radix-2 NTT algorithms, which are adapted to our 4×1 butterfly units, are shown in Algorithm 3.

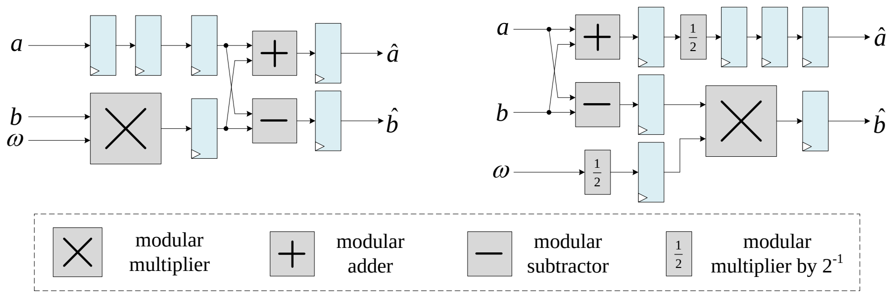

Figure 4: Data Paths for CT Butterfly (Left) and GS Butterfly (Right)

As can be seen from Algorithm 3, the NTT process is divided into three stages according to the index of the outer loop (i.e., the layer). The first stage involves layers 1 to $\log_2 n - 2$, where the butterfly units execute operations within a single group. The second stage involves layer $\log_2 n - 1$, where the butterfly units execute operations within two adjacent groups. The third stage involves the last layer, where the butterfly units execute operations within four adjacent groups. The last layer will be skipped when the signal *sel* equals 0, corresponding to the incomplete NTT in Kyber. The INTT is performed in the opposite direction, using GS instead of CT butterfly units.

4.2.2 Memory Map

The 4×1 butterfly units require eight input data (excluding twiddle factors) and produce eight output data every clock cycle. Therefore, the memory should contain eight banks to access these data in parallel. In [CYY+22], a conflict-free memory mapping scheme for NTT is proposed to make the access efficient. This scheme is applied to our design with a few modifications. Since the data from two consecutive addresses are always sent together to the 4×1 butterfly units, we can use a single address to store both of them. As a result, the number of banks decreases from eight to four. For the polynomial degree $n = 256$, the original address space decreases from 256 to 128. Following this modification, the bank index BI and the inner address of each bank BA, which are mapped by the

Algorithm 3 Radix-2 NTT with 4×1 CT Butterfly Units

Require: a vector $a = (a[0], a[1], ..., a[n-1]) \in \mathbb{Z}_q^n$ in standard order, a pre-computed table $\Psi \in \mathbb{Z}_q^n$ storing twiddle factors in bit-reversed order, a signal *sel* indicating whether the last layer needs to be processed

Ensure: $a = \mathrm{NTT}(a) \in \mathbb{Z}_q^n$ in bit-reversed order

```
1:  $k \leftarrow 0$
2:  for ($l = n/2;\ l > 0;\ l = l >> 1$) do
3:      for ($s = 0;\ s < n;\ s = s + 2l$) do
4:          if ($l >= 4$) then
5:              $\omega_1 \leftarrow \Psi[++k]$
6:              for ($j = s;\ j < s + l;\ j = j + 4$) do
7:                  $(a[j], a[j+l]) \leftarrow \mathrm{BF\_CT}(a[j], a[j+l], \omega_1)$
8:                  $(a[j+1], a[j+l+1]) \leftarrow \mathrm{BF\_CT}(a[j+1], a[j+l+1], \omega_1)$
9:                  $(a[j+2], a[j+l+2]) \leftarrow \mathrm{BF\_CT}(a[j+2], a[j+l+2], \omega_1)$
10:                 $(a[j+3], a[j+l+3]) \leftarrow \mathrm{BF\_CT}(a[j+3], a[j+l+3], \omega_1)$
11:             end for
12:         else if ($l == 2$) then
13:             $\omega_1 \leftarrow \Psi[++k],\ \omega_2 \leftarrow \Psi[++k]$
14:             $(a[s], a[s+2]) \leftarrow \mathrm{BF\_CT}(a[s], a[s+2], \omega_1)$
15:             $(a[s+1], a[s+3]) \leftarrow \mathrm{BF\_CT}(a[s+1], a[s+3], \omega_1)$
16:             $(a[s+4], a[s+6]) \leftarrow \mathrm{BF\_CT}(a[s+4], a[s+6], \omega_2)$
17:             $(a[s+5], a[s+7]) \leftarrow \mathrm{BF\_CT}(a[s+5], a[s+7], \omega_2)$
18:             $s \leftarrow s + 2l$
19:         else if ($l == 1$)&&($sel == 1$) then
20:             $\omega_1 \leftarrow \Psi[++k],\ \omega_2 \leftarrow \Psi[++k],\ \omega_3 \leftarrow \Psi[++k],\ \omega_4 \leftarrow \Psi[++k]$
21:             $(a[s], a[s+1]) \leftarrow \mathrm{BF\_CT}(a[s], a[s+1], \omega_1)$
22:             $(a[s+2], a[s+3]) \leftarrow \mathrm{BF\_CT}(a[s+2], a[s+3], \omega_2)$
23:             $(a[s+4], a[s+5]) \leftarrow \mathrm{BF\_CT}(a[s+4], a[s+5], \omega_3)$
24:             $(a[s+6], a[s+7]) \leftarrow \mathrm{BF\_CT}(a[s+6], a[s+7], \omega_4)$
25:             $s \leftarrow s + 6l$
26:         end if
27:     end for
28: end for
29: return $a$
```

original address A for radix-2 NTT with 4×1 butterfly units, are calculated as

$$\begin{aligned} BI &= (A[1:0] + ((A[6] + A[5] + A[4] + A[3] + A[2]) \mod 2) \cdot 2) \mod 4, \\ BA &= A >> 2. \end{aligned} \tag{10}$$

Taking 16 points as an example, the memory map and data flow for radix-2 NTT with 4×1 butterfly units are illustrated in Figure 5. The 16 points are stored in pairs, occupying 8 memory addresses. According to Equation 10, the 8 addresses are mapped to 4 banks. For each clock cycle, the 4×1 butterfly units access four addresses, which are mapped to four separate banks. Each bank is instantiated as a dual-port Random Access Memory (RAM), with one port for reading and the other for writing. As a result, the process of NTT can be fully pipelined. For Dilithium, the NTT process takes about $n/8 \times \log_2 n + 4 = 260$ clock cycles. For Kyber, in which the last layer is omitted, the NTT process takes about $n/8 \times (\log_2 n - 1) + 4 = 228$ clock cycles. The INTT is executed in the opposite direction, achieving the same performance as the NTT.

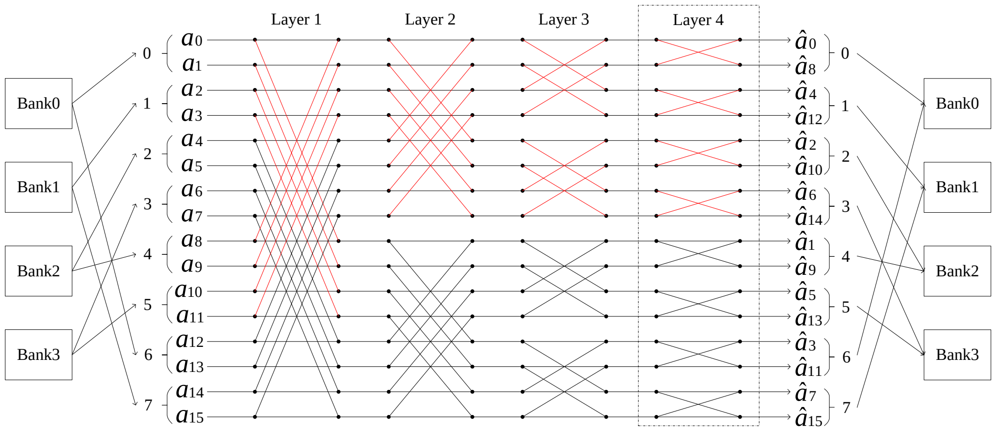

Figure 5: Memory Map and Data Flow for Radix-2 NTT with 4×1 Butterfly Units

4.3 NTT-Based Polynomial Arithmetic Operations

4.3.1 Polynomial Multiplication in NTT domain

The polynomial multiplications in the NTT domain for Kyber and Dilithium are different, as described in Subsection 2.2. For Dilithium, the multiplications are performed in a pointwise manner. We employ four modular multipliers to process four pairs of coefficients simultaneously. For Kyber, if we use the schoolbook method to calculate Equation 7, it requires five modular multipliers. To avoid the need for an extra modular multiplier that is not required by other operations, such as the 4×1 butterfly units, we adopt the Karatsuba algorithm to calculate Equation 7. That is,

$$\begin{aligned} \hat{c}_{2i} &= \hat{a}_{2i}\hat{b}_{2i} + \hat{a}_{2i+1}\hat{b}_{2i+1} \cdot \zeta^{2\mathrm{br}_7(i)+1}, \\ \hat{c}_{2i+1} &= (\hat{a}_{2i} + \hat{a}_{2i+1})(\hat{b}_{2i} + \hat{b}_{2i+1}) - (\hat{a}_{2i}\hat{b}_{2i} + \hat{a}_{2i+1}\hat{b}_{2i+1}). \end{aligned} \tag{11}$$

According to Equation 11, two pairs of coefficients can be processed by four modular multipliers, four modular adders, and one modular subtractor simultaneously. These hardware resources are reused from those constituting the butterfly units. Figure 6 depicts the data paths for polynomial multiplications in NTT domains of Kyber and Dilithium.

The polynomial multiplications in NTT domains are also executed in a fully pipelined manner. For Kyber, the data path is divided into seven pipeline stages. It takes about $n/2 + 7 = 135$ clock cycles to perform a multiplication. For Dilithium, the data path is divided into three pipeline stages, and a multiplication takes only about $n/4 + 3 = 67$ clock cycles. The coefficients are fetched from the memory banks in a continuous sequence, ensuring conflict-free data access through the memory mapping scheme.

4.3.2 High-Level Architecture

We use a unified block diagram to represent the high-level architecture of the accelerators for Kyber and Dilithium, despite the differences in implementation details. The block

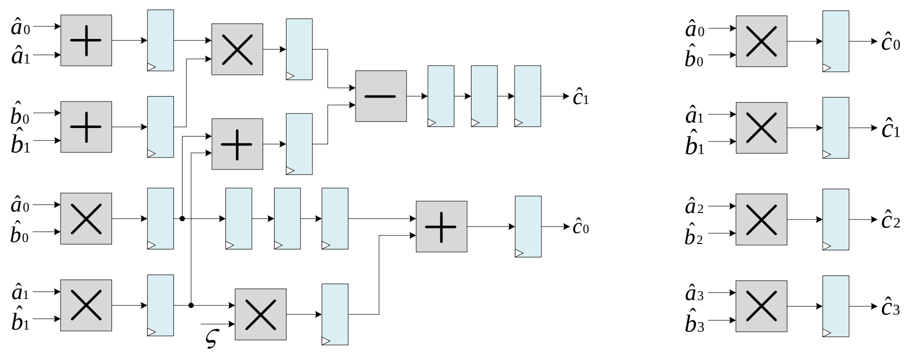

Figure 6: Data Paths for Polynomial Multiplication in NTT domains of Kyber (Left) and Dilithium (Right)

diagram is shown in Figure 7. This architecture consists of three main components: a *Polynomial Arithmetic Unit*, three 4-bank RAMs, and a *Finite State Machine* (FSM). The *Polynomial Arithmetic Unit* is responsible for executing arithmetic operations on polynomials. It comprises four modular multipliers, four modular adders, four modular subtractors, and four modular multipliers by 2^{-1}. The data path can be configured as mentioned above for various functions, such as NTT, INTT, and multiplication in the NTT domain. Besides, the pointwise additions can also be performed by this unit. A *Controller* in this unit manages the connections between the modular arithmetic units, registers, and the Read-Only Memory (ROM) storing twiddle factors. It generates a *pm_done* signal when the calculation of a function is completed. The *TF_Addr* unit generates proper addresses for the Twiddle ROM. The Twiddle ROM is instantiated as two or three separate ROMs with different data widths. It can meet the requirement of accessing multiple twiddle factors per clock cycle at different stages of NTT/INTT.

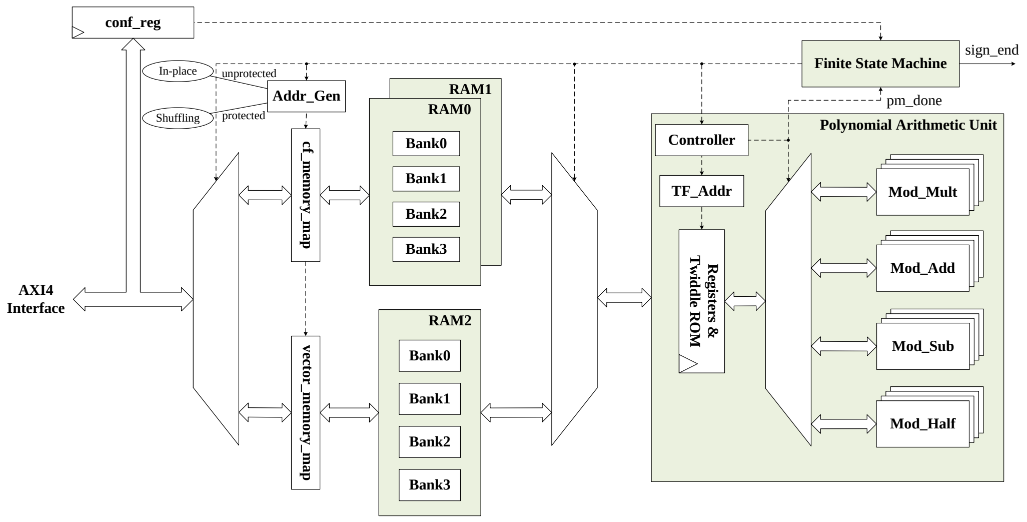

Figure 7: High-Level Architecture of the Accelerators for Kyber and Dilithium

The 4-bank RAMs, namely *RAM0*, *RAM1*, and *RAM2*, are utilized for storing polynomial coefficients. The *RAM0*, as well as *RAM1*, has a capacity of one polynomial. The polynomial coefficients stored in these two RAMs adhere to the conflict-free memory mapping scheme mentioned in Subsubsection 4.2.2. When executing NTT or INTT, the *Polynomial Arithmetic Unit* interacts with either *RAM0* or *RAM1*. When executing polynomial multiplications in NTT domains, the *Polynomial Arithmetic Unit* fetches data from both RAMs and writes the result back to *RAM0*. The *RAM2* is designed for storing a polynomial vector, which is useful in the process of polynomial matrix-vector multiplication. Compared to the other two RAMs, it does not require merging two consecutive coefficients into a single address, and the memory map is simpler. In the *vector_memory_map*, the least significant two bits of the original address determine the bank index, while the other bits including the vector index form the inner address of each bank. When executing polynomial additions, the *Polynomial Arithmetic Unit* fetches data from *RAM0* and *RAM2* and writes the result back to *RAM2*. The *Addr_Gen* generates the original addresses for the operations. Furthermore, as a loosely-coupled accelerator mounted on the bus, it provides the *AXI4 Interface* that connects the bus with the RAMs. Through the *AXI4 Interface*, the software can send the original data to the RAMs and receive the processed result from them according to the AIX4 protocol.

4.3.3 Customized FSMs for Kyber and Dilithium

Our accelerators aim to speed up NTT-based polynomial operations in Kyber and Dilithium, especially the time-consuming matrix-vector multiplication. Therefore, the basic functions mentioned above should be performed in a proper sequence. We use the FSM unit to control the process sequence. As the actual cases of calculations vary in Kyber and Dilithium, customized FSMs for each of the algorithms can improve efficiency. We define the states, which are associated with the high-level architecture and commonly used by both algorithms, as follows.

- IDLE: Do not perform any operations and wait for instructions stored in *conf_reg*.
- WRITE0: Write a polynomial from *AXI4 Interface* to *RAM0*.
- WRITE1: Write a polynomial from *AXI4 Interface* to *RAM1*.
- WRITE2: Write a polynomial vector from *AXI4 Interface* to *RAM2*.
- NTT0: Perform NTT for the polynomial stored in *RAM0*.
- NTT1: Perform NTT for the polynomial stored in *RAM1*.
- PWM: Perform polynomial multiplication in the NTT domain for the polynomials stored in *RAM0* and *RAM1*, and write the result back to *RAM0*.
- INTT0: Perform INTT for the polynomial stored in *RAM0*.
- MACC: Perform polynomial addition for the polynomials stored in *RAM0* and *RAM2*, and write the result back to *RAM2* at a specified vector index.
- READ: Read a polynomial vector from *RAM2* to *AXI4 Interface*.

The transitions of the states for Kyber and Dilithium are carefully designed to meet actual demands. The state transition diagrams are shown in Figure 8 and Figure 9. The transition conditions are determined by the values of both the *conf_reg* and the *pm_done* signal. The principles of these state transition diagrams need to be explained in conjunction with the software component, which can be seen in the following section.

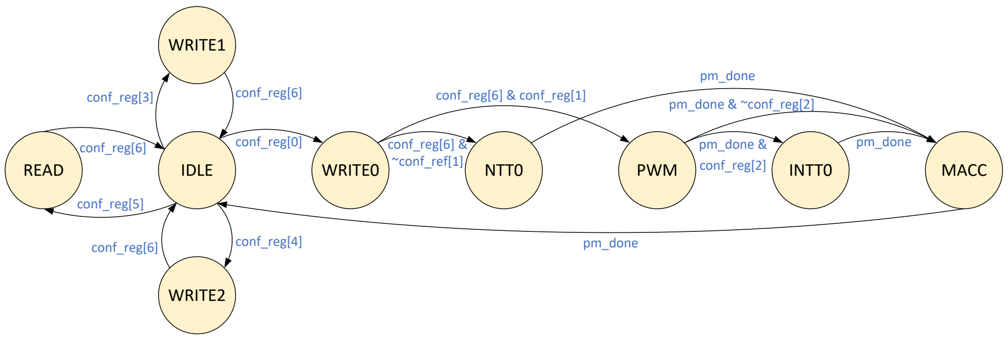

Figure 8: State Transition Diagram for Kyber

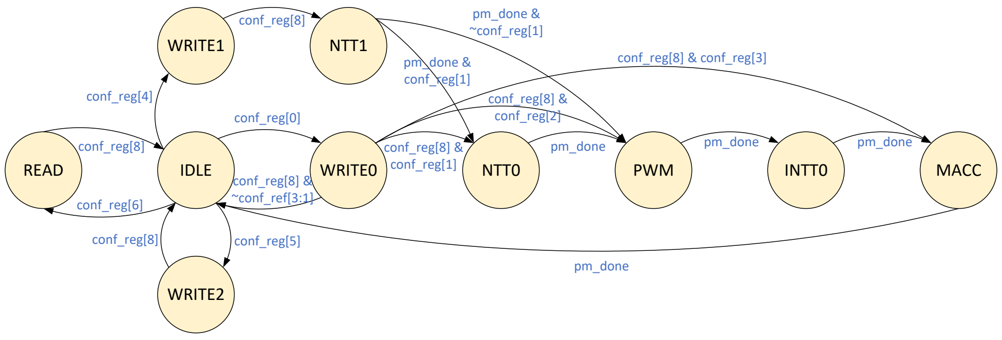

Figure 9: State Transition Diagram for Dilithium

4.4 Countermeasures against side-channel analysis

Besides efficiency, security against SCA is also an important factor to consider. Previous studies show that the SCA targeting NTT can access sensitive information in a single trace [PP19] or multiple traces [HHP$^+$21]. To protect the NTT, Ravi et al. [RPBC20] propose a range of generic shuffling and masking countermeasures. In this work, the countermeasures against SCA are reflected in three aspects.

Firstly, the modular arithmetic operations are executed in constant time, preventing simple power analysis (SPA) based on timing differences. This is an inherent feature of our modular arithmetic units and does not require any additional cost.

Secondly, the shuffling countermeasure for NTT is implemented to prevent single-trace template attacks. In our design, each set of four butterflies within any layer of the NTT is computed independently from the others. Each layer contains 32 independent sets. The execution order of these sets can be randomly shuffled as a countermeasure against SCA. Since the NTT is executed in a fully pipelined manner, the completely random orders in two adjacent layers may cause data hazards. To maintain high performance, a few limitations should be added to the shuffled orders of the latter layer. The limitations are that the first 9 sets of the latter layer have no data correlation with the last 9 sets of the former layer. We implement this countermeasure by modifying the *Addr_Gen* unit, as shown in Figure 7. If protection is considered, this unit generates shuffled addresses instead of in-place ones. The shuffled addresses are precomputed and stored in a read-only memory (ROM). The countermeasure cost is the additional resources for the ROM, while the performance is not affected. This countermeasure is sufficient to prevent single-trace template attacks. If a more robust countermeasure is required, an update scheme for the shuffled addresses should be introduced. As computational efficiency is the primary consideration in this work, we leave it for future research.

Thirdly, the masking countermeasure for polynomial arithmetic operations is implemented to prevent differential power attacks (DPA). It splits a secret polynomial into multiple parts called shares. These shares are processed individually to hide the power consumption that is correlated with the original secret. The operation being the same for each share, we can implement the masking countermeasure by utilizing the hardware accelerator to process the shares individually. The shares are generated in software, eliminating the need for any modifications to the hardware design. The masking polynomial arithmetic operations are described in Subsection 5.2.

5 Software Optimization

5.1 Memory Access

To ensure the hardware accelerator works as intended, the software needs to communicate with it. According to the PolarFire SoC MSS Technical Reference Manual [Mic22], the FPGA Fabric can be mapped to the address range from 0x60000000 to 0x7FFFFFFF. We define the starting address of the polynomial RAMs as 0x60010000, and the address of the *conf_reg* as 0x6001FFFC. The software can first write a configuration value to 0x6001FFFC, causing the accelerator to transition into a read or write state. Next, the polynomial coefficients can be transmitted between the main memory and the RAMs mapped at the starting address 0x60010000. The functions that implement such data transmissions are given below.

- **write_coeffs_0**(poly p, int c): Write a polynomial p from main memory to *RAM0*, with an integer c determining the subsequent state transitions.
- **write_coeffs_1**(poly p, int c): Write a polynomial p from main memory to *RAM1*, with an integer c determining the subsequent state transitions.
- **write_polyvec**(polyvec pv, int k): Write a polynomial vector pv from main memory to *RAM2*, with an integer k indicating the size of the vector.
- **read_polyvec**(polyvec pv, int k, int cls): Read a polynomial vector pv from *RAM2* to main memory, with an integer k indicating the size of the vector and an integer cls indicating whether *RAM2* should be cleared.

Implementing data transmission in these functions via the processor can be time-consuming. Fortunately, the PolarFire SoC FPGA is equipped with a DMA Engine that supports DMA transfers between the main memory and the FPGA Fabric. We employ two channels of the DMA Engine to accelerate the aforementioned functions. The first channel is used for **write_coeffs_0** and **write_coeffs_1** to transfer a single polynomial, while the second channel is used for **write_polyvec** and **read_polyvec** to transfer a polynomial vector. Additionally, the memory access speed can be further improved by configuring the L2 cache as a scratchpad. The scratchpad configuration allows for data stored in the scratchpad to be cached in a master's L1 data cache, resulting in faster access. The performance comparisons between processor transmission and DMA transmission in the context of Kyber and Dilithium are displayed in Figure 10.

In Figure 10, we show that the use of DMA and scratchpad significantly improves the memory access speed. The improvement becomes more evident as the vector size increases because the polynomials in a vector are stored in consecutive addresses. This fact implies

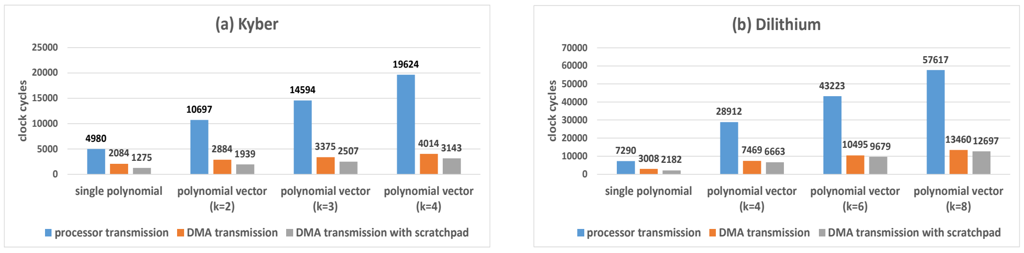

Figure 10: Clock Cycles for Processor Transmission and DMA Transmission

that if the software transfers the polynomials in a matrix or vector all at once, it would be faster than transferring them one by one. However, this idea requires large RAMs in the hardware accelerator. We make a trade-off between speed and hardware resources.

5.2 Polynomial Matrix and Vector Operations

The polynomial matrix and vector operations that our accelerators aim to speed up mainly include matrix-vector, vector-vector, and polynomial-vector multiplications and additions. Besides, NTT for a polynomial vector can also be performed separately according to practical needs. As mentioned above, the accelerator does not provide large RAMs for a polynomial matrix. Therefore, it requires a reasonable schedule for the software to calculate these operations. If the masking countermeasure against SCA is employed, the original sensitive polynomial vectors can be split into multiple shares, and the same operation can be performed on each share. Taking the typical operation $\mathbf{t} = \mathbf{A}\mathbf{s_1} + \mathbf{s_2}$ in Dilithium as an example, the procedure with masking countermeasures is described in Algorithm 4. This algorithm can be understood by referring to the state transition diagram shown in Figure 9. The first step is to write $\mathbf{s_2}[u]$ into *RAM2* so that it can be added to the result of $\mathbf{A}\mathbf{s_1}[u]$. The calculation of $\mathbf{A}\mathbf{s_1}[u]$ is implemented in steps 3-11 with the assistance of the hardware accelerator. In the outer loop, $\mathbf{A}[0][i]$ and $\mathbf{s_1}[u][i]$ are sent to *RAM0* and *RAM1*, respectively. The state transition corresponding to step 4 is IDLE→WRITE0→IDLE, while the one corresponding to step 5 is IDLE→WRITE1→NTT1→PWM→INTT0→MACC→IDLE. In this way, the accelerator calculates the product of $\mathbf{A}[0][i]$ and $\mathbf{s_1}[u][i]$. It should be noted that the state NTT0 is skipped because $\mathbf{A}$ is sampled in the NTT domain. The function **waitforhw**() causes the software to enter a sleep state until it receives the end signal from the accelerator. In the inner loop, the software only needs to send the polynomials in the i-th column of $\mathbf{A}$ to the accelerator one by one. These polynomials will be multiplied separately by $\mathbf{s_1}[u][i]$, which is already stored in *RAM1* using an NTT domain representation. The state transition corresponding to step 7 is IDLE→WRITE0→PWM→INTT0→MACC→IDLE. The result is stored in the j-th polynomial position of *RAM2*. In step 12, one share of the result vector is read from *RAM2*, while *RAM2* is cleared for subsequent operations.

The advantage of Algorithm 4 is that it avoids repetitive data transmissions and calculations, and keeps all intermediate results in hardware. There are other cases for polynomial matrix and vector operations in Kyber and Dilithium. The accelerator deals with them through different state transitions. The main cases are described as follows.

In Kyber, there are three main cases. The first case involves calculating NTT for a polynomial vector. This operation can be performed by using the **write_coeffs_0** function and controlling the state transition as IDLE→WRITE0→NTT0→MACC→IDLE. The second case involves matrix-vector multiplications where the resulting polynomials need to remain in the NTT domain. The third case involves matrix-vector and vector-vector

Algorithm 4 Matrix-Vector Multiplication and Addition Based on Hardware Accelerator

Require: $\mathbf{A} \in R_q^{k \times l}$, $\mathbf{s_1} \in R_q^{d \times l}$, $\mathbf{s_2} \in R_q^{d \times k}$, where k, l, q are parameters in Dilithium, d is the number of shares

Ensure: $\mathbf{t} = \mathbf{A}\mathbf{s_1} + \mathbf{s_2} \in R_q^{d \times k}$

```
1:  for ($u = 0;\ u < d;\ u = u + 1$) do
2:      write_polyvec($\mathbf{s_2}[u]$, $k$)
3:      for ($i = 0;\ i < l;\ i = i + 1$) do
4:          write_coeffs_0($\mathbf{A}[0][i]$, 0)
5:          write_coeffs_1($\mathbf{s_1}[u][i]$, 0)
6:          waitforhw()
7:          for ($j = 1;\ j < k;\ j = j + l$) do
8:              write_coeffs_0($\mathbf{A}[j][i]$, 2)
9:              waitforhw()
10:         end for
11:     end for
12:     read_polyvec($\mathbf{t}[u]$, $k$, 1)
13: end for
14: return t
```

multiplications, where the resulting polynomials need to be transformed back to the regular domain. For the latter two cases, the difference lies in whether the INTT0 state should be skipped. Since the operands have already been in NTT domain representation, the accelerator does not need to go through the NTT0 or NTT1 state.

In Dilithium, there are two main cases. The matrix-vector multiplication with a matrix in the NTT domain and a vector in the regular domain is the first case, which can be calculated using Algorithm 4. The second case involves the polynomial-vector multiplications with all operands in the regular domain. For this case, the scalar polynomial is written to *RAM1* after *RAM0* has received the first polynomial of the vector, with the state transition as IDLE→WRITE1→NTT1→NTT0→PWM→INTT0→MACC→IDLE. Next, the remaining polynomials in the vector are sent to *RAM0* one by one, with the state transition as IDLE→WRITE0→NTT0→PWM→INTT0→MACC→IDLE.

Besides, the functions **write_polyvec** and **read_polyvec** are optional in a specific scenario. For example, when calculating $\mathbf{w} = \mathbf{Ay}$ in the signing algorithm of Dilithium, the function **write_polyvec** should be omitted. The function **read_polyvec** can be omitted when a product is an intermediate value. For example, when computing $\mathbf{t} = \mathbf{As} + \mathbf{e}$ in the key generation algorithm of Kyber, where $\mathbf{e}$ should be first transformed into the NTT domain, it is not necessary to read the NTT result of $\mathbf{e}$ from the accelerator. Additionally, the function **read_polyvec** has a parameter *cls* to indicate whether *RAM2* should be cleared after reading. This parameter is set to zero when the intermediate value is used by both software and hardware. For example, when computing $c\mathbf{t_0}$ and $\mathbf{w} + c\mathbf{t_0}$ in the signing algorithm of Dilithium, the result of $c\mathbf{t_0}$ should be read without being cleared from *RAM2* for future use. Otherwise, *RAM2* should be overwritten with zeros.

5.3 SHA-3 Functions

The calculation of SHA-3 functions is another time-consuming task in Kyber and Dilithium. Previous works usually design a tightly coupled accelerator for SHA-3 [FBR+22, KSFS22], but the hardened processor does not allow us to make any changes to it. Designing a loosely coupled accelerator for SHA-3 would not be cost-efficient due to the significant communication overhead. There has been work using RISC-V assembly language to optimize the Keccak-*p* permutation of SHA-3 functions [Kos19]. However, according to our test, the performance of this work is not superior to that of a well-written C code compiled with "O3" level optimization. To achieve better efficiency, we make some modifications to this work. The optimization methods that we have employed are described as follows.

Firstly, as the RISC-V core supports the standard double-precision floating-point instruction-set extension named "D", we can use the FPR registers to store the state array. The state array of each SHA-3 function has a size of 25×64 bits, which can be held by 25 double-precision floating-point registers. The use of memory storing the state array is eliminated throughout the entire lifespan of each SHA-3 function. For example, the function SHAKE128 can be divided into two stages: absorbing and squeezing, which share f0-f24 registers for intermediate states. The floating-point registers are initialized with zeros during the absorbing stage and then moved to integer registers using the "fmv. x.d" instruction when they need to be processed. The values in f0-f20 will not be written to memory until the squeezing stage begins.

Listing 1: θ Step in [Kos19]

```
.macro CP _o,_a,_b,_c,_d,_e
  fmv.x.d  t0, \_a
  fmv.x.d  t1, \_b
  fmv.x.d  t2, \_c
  fmv.x.d  t3, \_d
  fmv.x.d  t4, \_e
  xor  t1, t1, t0
  xor  t1, t1, t2
  xor  t1, t1, t3
  xor  t1, t1, t4
  mv  \_o, t1
.endm

/* for i = 0 - 4, compute
   C[i]=A[i]^A[i+5]^A[i+10]
   ^A[i+15]^A[i+20] */

  CP  s1, 0, 5, 10, 15, 20
  CP  s2, 1, 6, 11, 16, 21
  CP  s3, 2, 7, 12, 17, 22
  CP  s4, 3, 8, 13, 18, 23
  CP  s5, 4, 9, 14, 19, 24
```

Listing 2: Optimized θ Step

```
.macro CP _a,_b,_c,_d,_e
  fmv.x.d  a3, f\_a
  fmv.x.d  a4, f\_b
  fmv.x.d  a5, f\_c
  fmv.x.d  t2, f\_d
  fmv.x.d  t3, f\_e
  c.xor  a0, a3
  c.xor  a1, a4
  c.xor  a2, a5
  xor  t0, t0, t2
  xor  t1, t1, t3
.endm

  fmv.x.d  a0, f0
  fmv.x.d  a1, f1
  fmv.x.d  a2, f2
  fmv.x.d  t0, f3
  fmv.x.d  t1, f4
  CP  5, 6, 7, 8, 9
  CP  10, 11, 12, 13, 14
  CP  15, 16, 17, 18, 19
  CP  20, 21, 22, 23, 24
```

Secondly, the execution order of assembly statements in the Keccak-*p* permutation is carefully arranged, aiming to fully utilize the pipeline feature of the RISC-V core. The 5-stage pipeline has a peak execution rate of one instruction per clock cycle. However, if there is a data hazard between adjacent instructions, the pipeline has to introduce a stall, resulting in performance losses. Taking the θ step as an example, it needs to compute the parity of each column. The original implementation of this step in [Kos19] is shown in

Listing 1. It can be observed that there are data hazards among the instructions in lines 7-11. As a contrast, our optimized implementation is shown in Listing 2. In this routine, the data dependency appears after at least 5 instructions. As a result, the 5-stage pipeline can be filled, allowing for the execution rate of one instruction per clock cycle. The other steps of the Keccak-*p* permutation can be optimized in the same way.

Thirdly, the RISC-V standard compressed instruction-set extension, named "C", is used to reduce the code size. The C extension replaces certain 32-bit instructions with 16-bit instructions without any loss in performance. As can be seen from Listing 2, the "xor" instructions in lines 7-9 are replaced with "c.xor" instructions. We employ registers a0 to a5 because they are required by the compressed instructions. The usage of registers s0 to s11 is abandoned to save time on stack operations. The compressed instructions are applied to all steps of the Keccak-*p* permutation, on condition that they do not compromise performance. As a result, the code size of the Keccak-*p* permutation is reduced by approximately 10%.

5.4 Multi-Core Acceleration

To the best of our knowledge, there is currently no work utilizing multi-core processors (either ARM or RISC-V) to enhance the implementation speed of Kyber or Dilithium, even though their mathematical structure is suitable for parallel computation. We have developed hardware accelerators for performing polynomial arithmetic operations in Kyber and Dilithium. However, this part of operations accounts for only around half of the total computation progress. The remaining operations executed by software mainly include hashing, sampling, packing/unpacking, encoding/decoding, and compression/decompression. The hashing operations are optimized using RISC-V assembly instructions, as described in Subsection 5.3. The other operations have something in common, that is, they all operate on polynomial matrices or vectors, and the processes for different polynomials do not have any data dependency. Based on this feature, we can easily assign computing tasks from a single core to multiple cores.

Taking the generation of the matrix $\mathbf{A}$ as an example, each polynomial in $\mathbf{A}$ is sampled uniformly from a pseudo-random string. The pseudo-random string is extended from a uniform seed and a nonce using the SHAKE128 function. In the context of single-core execution, the polynomials are sampled one by one. As the seed and nonce for each polynomial do not depend on the previous sampling result, it is possible to efficiently assign the task of sampling these polynomials to multiple cores. For one of the best cases, when the matrix $\mathbf{A}$ has 4×4 polynomials, each core is responsible for sampling four of them. In the practical context, several successive functions can be decomposed and distributed as evenly as possible across multiple cores to achieve a favorable balance.

The U54_1 to U54_4 cores are utilized for implementing the multi-core acceleration. The U54_1 controls the flow of the application program. It raises a synchronous software interrupt for each of the other U54 cores when a multi-core task is required. The cores U54_2 to U54_4 remain in the Wait for Interrupt (WFI) state until they receive the interrupt signals. The microprocessors communicate with each other through a shared memory area located in the L2 cache. The shared memory area stores data processed by multiple cores and variables used for synchronization. Based on this scheme, the target operations can be performed in parallel by multiple processors.

6 Experimental Results

We evaluate our hardware-software co-design on the PolarFire SoC FPGA Icicle Kit, which features the MPFS250T-FCVG484EES device. The hardware accelerators are simulated, synthesized, and implemented using Libero SoC v2022.2, while the software is developed and debugged using SoftConsole v2022.2. Kyber and Dilithium are implemented separately to achieve customized effects.

6.1 Effects of Hardware Acceleration

The synthesized and simulated results of our hardware accelerators for Kyber and Dilithium are listed in Table 5. There is no distinction between security levels because they share a common accelerator. The resources are measured by the utilized components available in Polarfire FPGA, which account for less than 5% of the total. The performance is measured by the clock cycles consumed by the major operations, which correspond to different states, as well as the maximum frequency. The accelerator for Dlithium has more resources and a lower frequency than the one for Kyber due to the disparity in prime fields. The clock cycles for both algorithms are generally in line with expectations, with a few additional cycles required for memory access. There are two versions of accelerators for each algorithm, depending on whether the shuffling countermeasure is implemented. By comparing the two versions, it can be observed that the countermeasure requires fewer than 300 additional 4LUT resources, with little impact on performance. This indicates that the shuffling countermeasures we implemented are cost-effective.

Table 5: Resources and Performance of the Accelerators for Kyber and Dilithium

Algorithm	**Resources**				**Clock Cycles**			Freq. (MHz)
	4LUT	DFF	Math	uSRAM	NTT/INTT	PWM	MACC	
Kyber[1]	6,035	1,896	4	38	236	140	76	165
Kyber[2]	6,325	1,877	4	38	236	140	76	165
Dilithium[1]	12,428	3,830	16	100	265	73	70	135
Dilithium[2]	12,673	3,822	16	100	265	73	70	131

[1] Without any countermeasure for the NTT/INTT implementation
[2] With a shuffling countermeasure for the NTT/INTT implementation

To demonstrate the effects of our hardware accelerators, we test the proportion of polynomial arithmetic operations that can be accelerated in the baseline implementations of Kyber and Dilithium. The baseline implementations use the official C code compiled by the RISC-V GCC toolchain with the "O3" optimization level. The average number of clock cycles is obtained by running the algorithm 1000 times. In Figure 11, we show the test results on a single U54 core for Kyber and Dilithium at NIST security level 5. The polynomial arithmetic operations displayed in blue are the components that would be accelerated by the accelerators. The test results of these algorithms, in which the polynomial arithmetic operations are accelerated by the hardware accelerators, are shown in Figure 12. As can be seen from the comparison of the two figures, the clock cycles of polynomial arithmetic operations are reduced by about 70%-90% with the assistance of hardware accelerators. As a result, the overall performance of these algorithms improves by approximately 20%-50%, depending on the proportion of time spent on polynomial arithmetic operations. With the performance of other operations remaining unchanged, the proportion of time spent on polynomial arithmetic operations drops to 7%-23%. The acceleration effects are similar for Kyber and Dilithium at other security levels.

The overhead of the masking countermeasure implemented in our design is also evaluated, as shown in Table 6. The clock cycles and code size are obtained from the accelerated implementations with and without first-order masking. It can be seen that the masking countermeasure has a significant impact on performance compared to code size. The signing algorithm is the most affected because it contains more sensitive variables than others. The

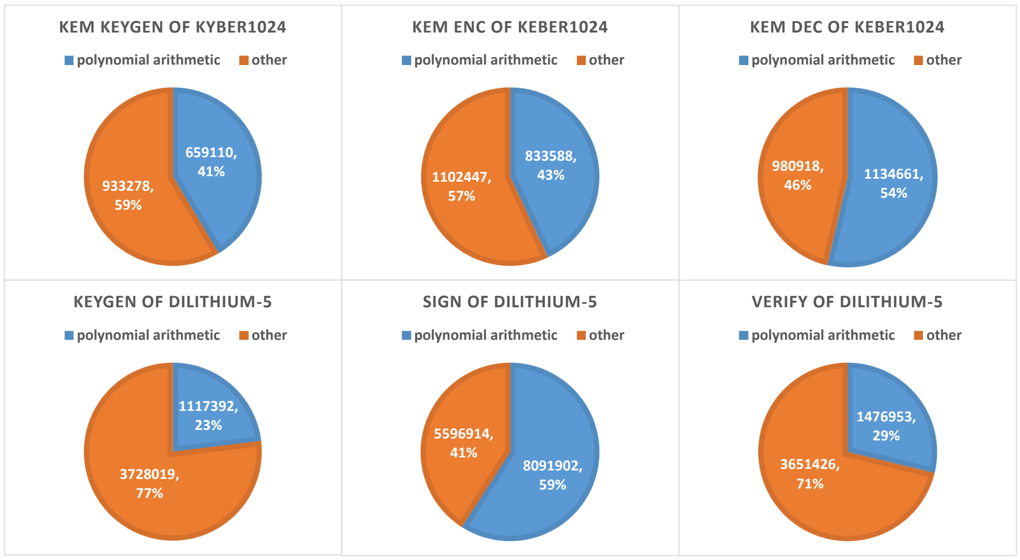

Figure 11: Clock Cycles of Baseline Implementations on a Single U51 Core

performance overhead is similar to that of the Generic-Random-Masked countermeasure presented in [RPBC20]. It achieves a balance between security and efficiency. The security of operations outside the hardware accelerator has not been considered in this work. It requires Boolean masking and corresponding secure conversions, which can lead to more overhead. We leave it for future research.

Table 6: Clock Cycles and Code Size for the Masked and Unmasked Implementations

Algorithm	**Kyber1024**				**Dilithium-5**		
	KeyGen	Encaps	Decaps	Code size	KeyGen	Sign	Code size
Unmasked	1,008,475	1,204,669	1,120,314	32,520	4,028,505	7,295,621	32,120
Masked	1,235,479	1,451,613	1,458,089	33,352	4,749,079	12,884,855	33,240
Overhead	22.5%	20.5%	30.2%	2.6%	17.9%	76.6%	3.5%

6.2 Effects of Code Optimization for SHA-3

The effects of code optimization for SHA-3 functions used in Kyber and Dilithium are depicted in Figure 13. We evaluate these functions individually. The results before optimization are obtained by compiling the official C code with the "O3" optimization level. The results after optimization involve the assembly code described in Subsection 5.3, which is compiled in the same manner. The performance is measured in terms of clock cycles per byte. For each function, we test the performance by setting its input to 10,000 random bytes. The output lengths of SHAKE128 and SHAKE256 are set to one squeezing block, depending on their rates. As shown in Figure 13, the performance of SHA-3 functions improves by approximately 15%-20%, while the code size decreases by about 40%-60%. The optimization of code size is more noticeable than that of performance because the official implementation adopts loop unrolling to achieve higher speed.

This increase in efficiency does not rely on any dedicated hardware or customized instruction sets. The flexibility advantage of the software is retained. It only requires

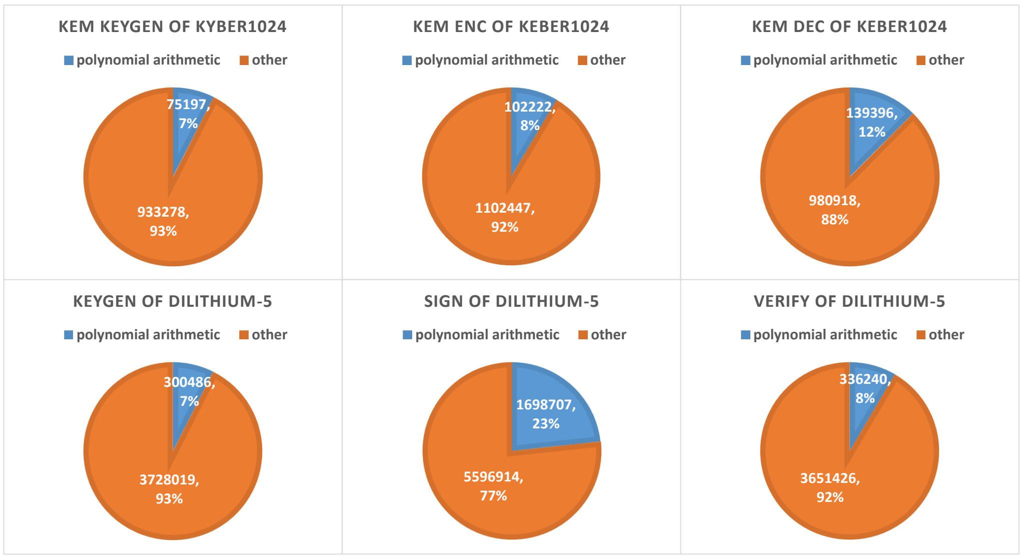

Figure 12: Clock Cycles of Accelerated Implementations on a Single U51 Core

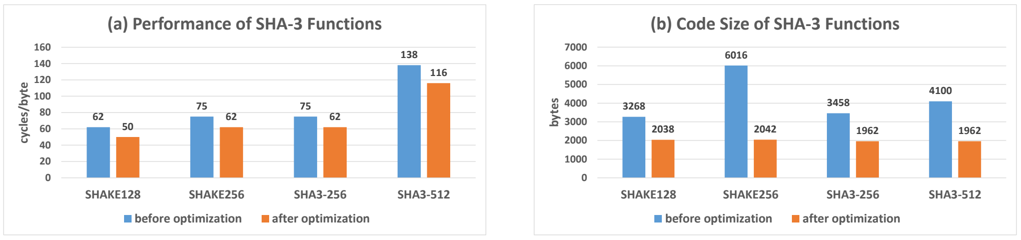

Figure 13: Performance and Code Size of SHA-3 Functions

the standard extensions "D" and "C", which are supported by most general-purpose RISC-V processors. The 64-bit registers and instructions are also necessary for storing and processing the 1600-bit state of SHA-3. Since SHA-3 functions account for a large proportion of calculations other than polynomial arithmetic operations, the optimization further improves the overall performance by about 10%-15% following the hardware acceleration. However, the software still takes 5 to 10 times longer than the hardware does, significantly impacting overall performance.

6.3 Effects of Multi-Core Acceleration

In Figure 14, we show the test results of different implementations of Kyber and Dilithium at NIST security level 5. The implementation methods include the baseline implementation, code optimization on a single core with an accelerator, and code optimization on multiple cores with an accelerator. As can be seen from the comparisons, the effects of acceleration and optimization are significant for all algorithms in Kyber and Dilithium. On top of the hardware acceleration and code optimization, the multi-core acceleration further enhances the performance by around 40%-60%. For the other security level, the multi-core acceleration exhibits similar effects.

The polynomial arithmetic operations can also be accelerated by the multiple processors,

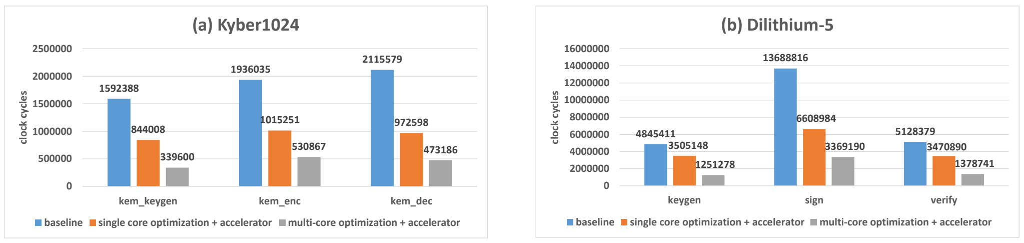

Figure 14: Clock Cycles of Different Implementations of Kyber and Dilithium

instead of the hardware accelerator. However, the multi-core acceleration can not achieve the speed-up ratio of 5-10 times as the hardware accelerator does. The operations executed by the multiple processors can also be accelerated by a more powerful hardware accelerator, but this comes at the cost of using more FPGA resources and sacrificing flexibility. We make a tradeoff between the two approaches. The polynomial arithmetic operations are independent of other operations, making it suitable to offload them to the hardware accelerator. The multiple processors are adept at handling other discrete operations, including the optimized SHA-3 functions, thereby ensuring efficient multi-core acceleration. The test results indicate that the multi-core acceleration and the hardware acceleration have a similar impact on the overall performance.

6.4 Comparison with Related Works

In Table 7 and Table 8, we present the performance results of our unmasked implementations for Kyber and Dilithium, respectively. For each algorithm, this work gives three versions of implementations. All versions are compiled by the RISC-V GCC toolchain with the "O3" optimization level and tested on U54 cores with a scratchpad memory configuration. The first version is the baseline software implementation which uses the official code directly. We provide the test results to demonstrate the performance of a single U54 core for reference. The second version contains the hardware acceleration using FPGA and the code optimization for SHA-3. The third version further introduces the multi-core acceleration. The frequency and the cycles listed here are both obtained from the processor's clock. The time in microseconds is calculated according to the frequency and cycles. The comparison of these three versions of implementations demonstrates the effects of our acceleration and optimization methods across different security levels. The overall performance can be improved by approximately 3 to 5 times.

Several related works that present outstanding results using hardware-software co-design are also listed for comparison. The related works published in the earlier years, such as [BUC19, FSS20, AEL+20, XHY+20], are not listed here. The reason is that they implement the older versions of Kyber and Dilithium with some parameter differences from ours. These differences can have an impact on performance. Since there have been enough state-of-the-art works for comparison, we omit the comparisons with earlier implementations for simplicity. The works listed here mainly involve solutions for customizing ISA extensions in open-source RISC-V cores, such as [NMZ+21, FBR+22, KSFS22, ZXXH22]. The designs of loosely-coupled accelerators can be found in [FBR+22, KSFS22, DMMM23, MCL+23]. The FPGA resources utilized by these works are presented in Table 9. The related works all evaluate their designs based on Xilinx FPGA, so we re-synthesis our hardware accelerators on Xilinx FPGA using Vivado 2022.1 to make them comparable. It should be noted that the data presented in the three tables does not include any countermeasures against SCA.

The performance comparisons are not entirely fair due to the differences in platforms and

Table 7: Comparison with Related Hardware-Software Co-Design for Kyber KEM

Work	Platform	Freq. (MHz)	KeyGen		Encapsulation		Decapsulation	
			Cycles	Time[μs]	Cycles	Time[μs]	Cycles	Time[μs]
Kyber512								
[NMZ+21]	CVA6	100	419,597	4,196	438,280	4,383	100,796	1,008
[FBR+22]	PULP	79.74	116,454	1,456	176,034	2,200	186,341	2,329
[ZXXH22]	Rocket	540	9,400	17	19,000	35	43,800	81
[DMMM23]	PULP	-	395,495	-	552,827	-	726,049	-
This[1]	PolarFire	600	623,342	1,039	834,370	1,391	956,454	1,594
This[2]	PolarFire	600	326,983	545	415,496	692	394,661	658
This[3]	PolarFire	600	151,593	253	260,923	435	233,616	389
Kyber768								
[NMZ+21]	CVA6	100	694,504	6,945	731,597	7,316	130,348	1,303
[FBR+22]	PULP	79.74	213,862	2,682	298,048	3,738	313,034	3,926
[ZXXH22]	Rocket	540	14,200	26	26,200	49	59,100	109
[DMMM23]	PULP	-	663,059	-	856,258	-	1,083,818	-
This[1]	PolarFire	600	1,031,670	1,719	1,329,313	2,216	1,480,350	2,467
This[2]	PolarFire	600	536,213	894	671,082	1,118	639,024	1,065
This[3]	PolarFire	600	231,334	386	378,792	631	334,569	558
Kyber1024								
[NMZ+21]	CVA6	100	1,090,458	10,905	1,126,462	11,265	159,639	1,596
[FBR+22]	PULP	79.74	266,209	3,338	368,409	4,620	392,873	4,927
[ZXXH22]	Rocket	540	18,500	34	33,700	62	77,500	144
[DMMM23]	PULP	-	1,001,350	-	1,247,565	-	1,523,411	-
This[1]	PolarFire	600	1,592,388	2,654	1,936,035	3,227	2,115,579	3,526
This[2]	PolarFire	600	844,008	1,407	1,015,251	1,692	972,598	1,621
This[3]	PolarFire	600	339,600	566	530,867	885	473,186	789

[1] Baseline software implementation on a single core
[2] Hardware-software co-design implementation on a single core
[3] Hardware-software co-design implementation on multiple cores

available resources. Several objective factors significantly impact the overall performance. Firstly, the PolarFire SoC FPGA platform requires us to use an L2 cache to store the program data, while other RISC-V-based works use on-chip memory like Block RAM (BRAM). According to our test, it takes about 7-9 clock cycles to read a 32-bit word from the L2 cache, while the on-chip memory only requires 2-3 clock cycles. Secondly, the hardened RISC-V processors we use do not allow us to make any changes to them, so the custom instruction extension solutions are not available in this work. Thirdly, our design uses separate clocks for software and hardware. The frequency of our FPGA accelerator, as shown in Table 5, is much lower than that of the RISC-V processor, while other works using ASIC can achieve a high frequency for their accelerators.

The work of [NMZ+21] requires more clock cycles compared to our accelerated implementations for both Kyber and Dilithium. The reason is that they employ only a small accelerator to speed up NTT/INTT. Although it takes fewer FPGA resources than ours, this accelerator can be rather inefficient if it is loosely coupled with the processor. Their

Table 8: Comparison with Related Hardware-Software Co-Design for Dilithium

Work	Platform	Freq. (MHz)	KeyGen		Sign		Verify	
			Cycles	Time[μs]	Cycles	Time[μs]	Cycles	Time[μs]
Dilithium-2								
[NMZ+21]	CVA6	100	1,592,325	15,923	5,884,266	58,843	1,700,679	17,017
[KSFS22]	PULP	800	593,403	742	1,905,872	2,382	651,217	814
[ZXXH22]	Rocket	540	45,800	85	175,100	324	89,800	166
[MCL+23]	Zynq	666	732,600	1,100	3,929,400	5,900	732,600	1,100
This[1]	PolarFire	600	1,658,593	2,764	6,754,172	11,257	1,862,610	3,104
This[2]	PolarFire	600	1,135,669	1,893	3,089,124	5,149	1,161,199	1,935
This[3]	PolarFire	600	438,698	731	1,697,566	2,829	542,212	904
Dilithium-3								
[NMZ+21]	CVA6	100	2,974,897	29,749	10,211,677	102,117	2,963,936	29,639
[KSFS22]	PULP	800	1,067,824	1,335	3,253,378	4,067	1,126,938	1,409
[ZXXH22]	Rocket	540	68,400	127	224,600	416	110,300	204
[MCL+23]	Zynq	666	999,000	1,500	5,794,200	8,700	1,065,600	1,600
This[1]	PolarFire	600	2,953,626	4,923	11,168,275	18,614	3,068,320	5,114
This[2]	PolarFire	600	2,087,190	3,479	5,002,513	8,338	1,986,265	3,310
This[3]	PolarFire	600	798,111	1,330	2,990,703	4,985	879,854	1,466
Dilithium-5								
[NMZ+21]	CVA6	100	5,001,302	50,013	13,339,255	133,393	5,132,776	51,328
[KSFS22]	PULP	800	1,784,767	2,231	4,357,249	5,447	1,848,324	2,310
[ZXXH22]	Rocket	540	94,900	176	313,200	580	160,000	296
[MCL+23]	Zynq	666	1,465,200	2,200	6,793,200	10,200	1,531,800	2,300
This[1]	PolarFire	600	4,845,411	8,076	13,688,816	22,814	5,128,379	8,547
This[2]	PolarFire	600	3,505,148	5,842	6,608,984	11,015	3,470,890	5,785
This[3]	PolarFire	600	1,251,278	2,085	3,369,190	5,615	1,378,741	2,298

[1] Baseline software implementation on a single core
[2] Hardware-software co-design implementation on a single core
[3] Hardware-software co-design implementation on multiple cores

tightly coupled accelerator achieves a speedup ratio of 2 to 4 times for NTT/INTT. In contrast, our loosely coupled accelerator achieves a speedup ratio of 3 to 10 times.

The work of [FBR+22] designs loosely coupled accelerators for NTT-based operations and tightly coupled accelerators for SHA-3 functions in Kyber. Our hardware-software co-design has a shorter execution time and uses fewer FPGA resources compared to their design. The number of clock cycles is not as small as theirs due to the objective factors explained above. The FPGA resources of their design are 2 to 3 times greater than ours, most of which are used for SHA-3. Their loosely coupled accelerator requires approximately 4,096 cycles to calculate NTT, which is 16 times longer than ours. The resources used for it consist of 2,454 LUTs and 1,917 FFs, which are similar to ours.

The work of [KSFS22] uses the same platform and a similar architecture as [FBR+22] to enhance the speed of Dilithium. Their work requires more clock cycles compared to the third version of our implementations, while also utilizing more FPGA resources. The execution time is short because of the high frequency of their ASIC clock. Their design

Table 9: Comparison of FPGA Resources for Accelerating Kyber and Dilithium

Work	**FPGA**	**Kyber**				**Dilithium**			
		LUT	FF	DSP	BRAM	LUT	FF	DSP	BRAM
[NMZ+21]	Zynq UltraScale+	3,173	58	5	0.5	3,506	71	10	0.5
[FBR+22]	Artix 7	7,687	3,515	7	4.5	-	-	-	-
[KSFS22]	Zynq UltraScale+	-	-	-	-	7,219	3,238	7	6
[ZXXH22]	Artix 7	25,674	3,137	64	6	25,674	3,137	64	6
[DMMM23]	Artix 7	7,128	3,798	-	-	-	-	-	-
[MCL+23]	Zynq 7000	-	-	-	-	9,365	6,811	4	5
This[1]	Zynq 7000	3,087	1,050	4	4	6,639	1,660	16	3

[1] We re-synthesis our hardware accelerators based on Xilinx FPGA using Vivado.

reduces the FPGA resources for NTT-based operations to 1,402 LUTs and 1,192 FFs. Our accelerator for Dilithium consumes much more LUTs due to the utilization of 1,351 LUTRAMs instead of BRAMs. Nonetheless, we achieve a speed-up ratio of 16 times for NTT at the cost of no more than 5 times increase in LUT resources.

The work of [ZXXH22] provides the most high-performance implementations for Kyber and Dilithium among the listed works. However, the high performance comes with a high cost. As can be seen from Table 9, their accelerator consumes much more FPGA resources than ours, particularly in terms of LUT and DSP. Their design heavily relies on the hardware crypto-core for most computing tasks, resulting in performance that is close to a pure hardware implementation. In contrast, our accelerators handle only about half of the computing tasks. Therefore, our design maintains a certain level of flexibility and requires fewer hardware resources.

The work of [DMMM23] proposes a loosely coupled Keccak accelerator to speed up SHA-3 functions for Kyber. It can be observed that the second version of our implementations surpasses theirs in terms of both performance and resource utilization. This comparison indicates that accelerating NTT-based operations is more efficient than accelerating SHA-3 in a loosely coupled hardware module for Kyber. Therefore, our decision to implement SHA-3 in software is reasonable, as it makes the accelerator more compact and efficient.

The work of [MCL+23] presents a software-hardware co-design for Dilithium. Although it is not implemented on a RISC-V processor, their work employs the ARM-based Zynq platform which shares similar features with PolarFire SoC FPGA. When compared to [MCL+23], our design does not suffer from the drawbacks of memory access, custom ISA extension, and clock frequencies. Therefore, the comparison is relatively fair. From Table 8 and Table 9, we can observe that the second version of our implementations has performance advantages in the signing algorithm, while resources other than DSP are also saved. The reason is that we put more effort into accelerating NTT-based operations, which account for a large proportion of the signing algorithm. Their work consumes more FPGA resources since it involves hardware accelerators for SHA-3 and sampling operations. The third version of our implementations has performance advantages in all algorithms of Dilithium, thanks to the utilization of multiple cores.

To be clear, our work is not merely an integration of existing technologies. As described in the previous sections, several innovative improvements are presented to make the computations more suitable for Kyber and Dilithium. Compared to related works, the biggest difference in our design is the collaboration between the customized FSM and the software scheduling. It enables continuous sequences of operations to be executed without interruption in the hardware while reducing the overhead of instruction interaction. This solution is designed based on the actual cases of operation sequences in Kyber and

Dilithium, ensuring that the computations are suitable for them.

6.5 Power and Energy Consumption

The consumption of power and energy is also an important metric for evaluating our implementations of Kyber and Dilithium. In Table 10, we present this metric for the three versions of our implementations. The power consumption is measured using SmartPower, which is a power-analysis tool that provides a detailed and accurate way to analyze designs for Microchip SoC FPGAs. We set the operating condition to be "Typical", which indicates that the measurement is conducted at a temperature of 25°C and a core voltage of 1.05V. Our baseline software implementation involves only a single working RISC-V core while other cores are on standby, so it has the lowest power consumption. The second version introduces a hardware accelerator, resulting in a slight increase in power consumption. The third version involves multiple cores working together, which consumes more power compared to the other versions that keep some cores in standby mode. The energy consumption is calculated by multiplying power and time. Since the increase in power consumption is insignificant compared to the improvement in performance, the energy consumption saved is similar to the time saved. This indicates that our hardware-software co-design achieves benefits in terms of energy consumption.

Table 10: Power and Energy Consumption for Implementations of Kyber and Dilithium

Algorithm	**Power (mW)**			**Energy (mJ)**		
	Baseline[1]	Accel.[2]	Accel.[3]	Baseline[1]	Accel.[2]	Accel.[3]
Kyber512	1,727	1,833	1,922	1.8/2.5/2.9	1.0/1.3/1.2	0.5/0.8/0.7
Kyber768	1,727	1,833	1,922	3.0/3.8/4.3	1.6/2.0/2.0	0.7/1.2/1.1
Kyber1024	1,727	1,833	1,922	4.6/5.6/6.1	2.6/3.1/3.0	1.1/1.7/1.5
Dilithium-2	1,727	1,847	1,936	4.8/19/5.4	3.5/9.5/3.6	1.4/5.5/1.8
Dilithium-3	1,727	1,847	1,936	8.5/32/8.8	6.4/15/6.1	2.6/9.7/2.8
Dilithium-5	1,727	1,847	1,936	14/39/15	11/20/11	4.0/11/4.4

[1] Baseline software implementation on a single core
[2] Hardware-software co-design implementation on a single core
[3] Hardware-software co-design implementation on multiple cores

For implementations with countermeasures against SCA, the average power consumption remains relatively unchanged while energy consumption increases due to the additional time required. However, the characteristic of runtime power consumption is significantly affected by the countermeasures. To evaluate the power leakage for both unprotected and protected hardware accelerators, we acquire the runtime power consumption on a SAKURA-X board that features a Kintex-7 FPGA through a LeCroy oscilloscope at a sampling frequency of 500MHz. Figure 15 shows a single power trace of the accelerator for Kyber, where the voltage value is used to represent the runtime power consumption. It corresponds to the state transition as IDLE→WRITE0→NTT0→MACC→IDLE. The states NTT0 and MACC can be easily observed from the peaks. We use 100,000 such traces as an example to evaluate the power leakage. The other calculation patterns exhibit similar characteristics.

We employ the Test Vector Leakage Assessment (TVLA) [GJJR11, SM15] methodology to evaluate power leakage from the traces. Given two sets of data with n_0 and n_1 power

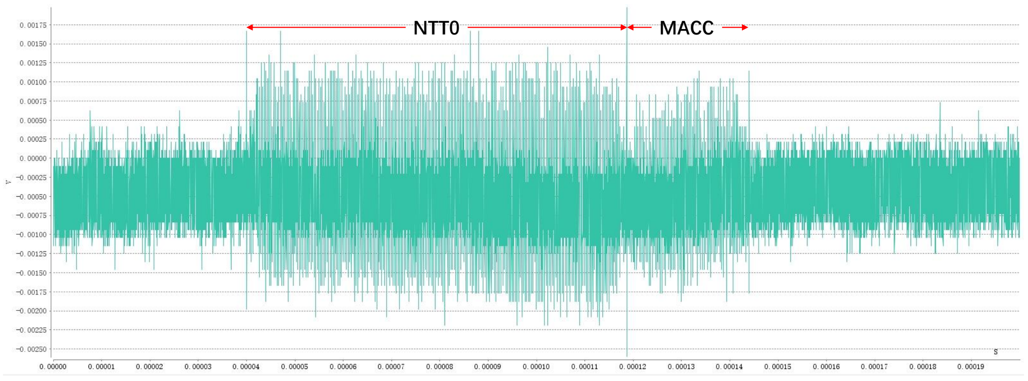

Figure 15: A Single Power Trace of the Accelerator for Kyber

traces, respectively, the TVLA metric is calculated as

$$t = \frac{\mu_0 - \mu_1}{\sqrt{\frac{s_0^2}{n_0} + \frac{s_1^2}{n_1}}}, \tag{12}$$

where μ_0, μ_1 are the means of the two sets, while s_0^2, s_1^2 are their respective variances. In literature, a threshold of $|t| > 4.5$ is usually defined to indicate that it is possible for an attacker to statistically distinguish between the two sets. In our measurement, the two sets are divided according to whether the inputs are random. One set contains 50,000 traces with random inputs while the other set contains 50,000 traces with fixed inputs. The calculation results of TVLA for power traces of unprotected and protected accelerators are depicted in Figure 16. It can be observed that there are significant differences between the two groups of results. The t values for the unprotected accelerator mostly exceed the threshold, which is marked by two red lines, especially during the first stage of NTT operation. On the contrary, the t values for the protected accelerator all fall within the threshold range. This indicates that obvious power leakages exist if the accelerator is not protected, while the countermeasures we take can mitigate such leakages. As SCA and countermeasures for Kyber and Dilithium are not the main focus of this paper, we leave more in-depth research on this field as future work.

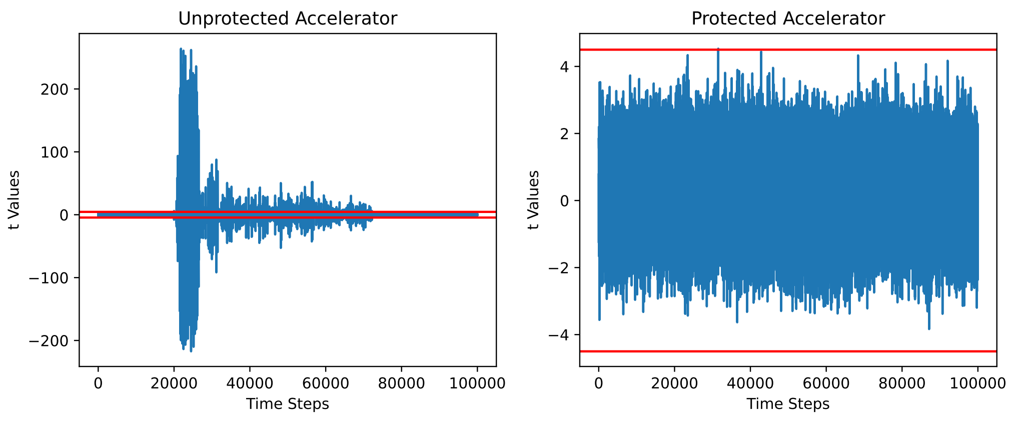

Figure 16: TVLA for Power Traces of Unprotected and Protected Accelerators

7 Conclusion

As Kyber and Dilithium have been selected for standardization in the third round of the NIST PQC standardization process, efficient implementations for them on various platforms become more valuable. In this work, we choose the industry's first RISC-V-based SoC FPGA as our experimental platform and present optimized hardware-software co-design for Kyber and Dilithium. Our design incorporates hardware accelerators to enhance the speed of polynomial arithmetic operations, optimized RISC-V assembly code for SHA-3 functions, and multi-core acceleration solutions for other matrix and vector-related operations. The experimental results show that it achieves a significant improvement in performance with reasonable resource utilization. The countermeasures against SCA for the hardware accelerator and a proper evaluation are also considered in this work.

Furthermore, our acceleration and optimization methods for Kyber and Dilithium are not limited to the platform we choose. The acceleration solutions, whether using FPGA or multiple cores, are irrelevant to ISA. They can be migrated to other platforms that contain FPGA or multi-core resources, such as Zynq-7000 SoC. The code optimization for SHA-3 functions does not need dedicated hardware. It can be implemented on various general-purpose RISC-V processors supporting the standard extensions "D" and "C". The RISC-V processors currently in use may not possess the same level of high performance as x86/x64 or ARM-based processors, but it is still worthwhile to implement and optimize the PQC algorithms on such platforms for a wider range of applications.

Acknowledgements

This work is supported in part by the National Natural Science Foundation of China under Grant No.U2336210, and in part by the Startup Fund for Young Faculty at SJTU (SFYF at SJTU) under Grant No.24X010500123, and in part by the Shanghai Science and Technology lnnovation Action Program under Grant No.22511101300.

References

[AAB+16] Krste Asanović, Rimas Avizienis, Jonathan Bachrach, Scott Beamer, David Biancolin, Christopher Celio, Henry Cook, Daniel Dabbelt, John Hauser, Adam Izraelevitz, Sagar Karandikar, Ben Keller, Donggyu Kim, John Koenig, Yunsup Lee, Eric Love, Martin Maas, Albert Magyar, Howard Mao, Miquel Moreto, Albert Ou, David A. Patterson, Brian Richards, Colin Schmidt, Stephen Twigg, Huy Vo, and Andrew Waterman. The rocket chip generator. Technical Report UCB/EECS-2016-17, EECS Department, University of California, Berkeley, Apr 2016.

[ABCG20] Erdem Alkim, Yusuf Alper Bilgin, Murat Cenk, and François Gérard. Cortex-m4 optimizations for {R, M} LWE schemes. *IACR Trans. Cryptogr. Hardw. Embed. Syst.*, 2020(3):336–357, 2020.

[ABD+a] Erdem Alkim, Joppe W. Bos, Léo Ducas, Patrick Longa, Ilya Mironov, Michael Naehrig, Valeria Nikolaenko, Chris Peikert, Ananth Raghunathan, and Douglas Stebila. Frodokem, algorithm specifications and supporting documentation. https://frodokem.org/.

[ABD+b] Roberto Avanzi, Joppe Bos, Léo Ducas, Eike Kiltz, Tancrède Lepoint, Vadim Lyubashevsky, John M. Schanck, Peter Schwabe, Gregor Seiler, and Damien

Stehlé. Crystals-kyber, algorithm specifications and supporting documentation. https://pq-crystals.org/kyber/resources.shtml.

[AEL+20] Erdem Alkim, Hülya Evkan, Norman Lahr, Ruben Niederhagen, and Richard Petri. ISA extensions for finite field arithmetic accelerating kyber and newhope on RISC-V. *IACR Trans. Cryptogr. Hardw. Embed. Syst.*, 2020(3):219–242, 2020.

[AY22] Latif Akçay and Berna Örs Yalçin. Analysing the potential of transport triggered architecture for lattice-based cryptography algorithms. *Int. J. Embed. Syst.*, 15(5):404–420, 2022.

[BAK21] Mojtaba Bisheh-Niasar, Reza Azarderakhsh, and Mehran Mozaffari Kermani. High-speed ntt-based polynomial multiplication accelerator for post-quantum cryptography. In *28th IEEE Symposium on Computer Arithmetic, ARITH 2021, Lyngby, Denmark, June 14-16, 2021*, pages 94–101. IEEE, 2021.

[Bar86] Paul Barrett. Implementing the rivest shamir and adleman public key encryption algorithm on a standard digital signal processor. In *Advances in Cryptology - CRYPTO '86, Santa Barbara, California, USA, 1986, Proceedings*, pages 311–323, 1986.

[BDK+] Shi Bai, Léo Ducas, Eike Kiltz, Tancrède Lepoint, Vadim Lyubashevsky, Peter SchwabePeter, Gregor Seiler, and Damien Stehlé. Crystals-dilithium, algorithm specifications and supporting documentation. https://pq-crystals.org/dilithium/resources.shtml.

[BDK+18] Joppe W. Bos, Léo Ducas, Eike Kiltz, Tancrède Lepoint, Vadim Lyubashevsky, John M. Schanck, Peter Schwabe, Gregor Seiler, and Damien Stehlé. CRYSTALS - kyber: A cca-secure module-lattice-based KEM. In *2018 IEEE European Symposium on Security and Privacy, EuroS&P 2018, London, United Kingdom, April 24-26, 2018*, pages 353–367, 2018.

[BUC19] Utsav Banerjee, Tenzin S. Ukyab, and Anantha P. Chandrakasan. Sapphire: A configurable crypto-processor for post-quantum lattice-based protocols. *IACR Trans. Cryptogr. Hardw. Embed. Syst.*, 2019(4):17–61, 2019.

[CYY+22] Xiangren Chen, Bohan Yang, Shouyi Yin, Shaojun Wei, and Leibo Liu. CFNTT: scalable radix-2/4 NTT multiplication architecture with an efficient conflict-free memory mapping scheme. *IACR Trans. Cryptogr. Hardw. Embed. Syst.*, 2022(1):94–126, 2022.

[DKL+18] Léo Ducas, Eike Kiltz, Tancrède Lepoint, Vadim Lyubashevsky, Peter Schwabe, Gregor Seiler, and Damien Stehlé. Crystals-dilithium: A lattice-based digital signature scheme. *IACR Trans. Cryptogr. Hardw. Embed. Syst.*, 2018(1):238–268, 2018.

[DMMM23] Alessandra Dolmeta, Mattia Mirigaldi, Maurizio Martina, and Guido Masera. Implementation and integration of keccak accelerator on RISC-V for crystals-kyber. In Andrea Bartolini, Kristian F. D. Rietveld, Catherine D. Schuman, and Jose Moreira, editors, *Proceedings of the 20th ACM International Conference on Computing Frontiers, CF 2023, Bologna, Italy, May 9-11, 2023*, pages 381–382. ACM, 2023.

[FBR+22] Tim Fritzmann, Michiel Van Beirendonck, Debapriya Basu Roy, Patrick Karl, Thomas Schamberger, Ingrid Verbauwhede, and Georg Sigl. Masked accelerators and instruction set extensions for post-quantum cryptography. *IACR Trans. Cryptogr. Hardw. Embed. Syst.*, 2022(1):414–460, 2022.

[FO99] Eiichiro Fujisaki and Tatsuaki Okamoto. Secure integration of asymmetric and symmetric encryption schemes. In Michael J. Wiener, editor, *Advances in Cryptology - CRYPTO '99, 19th Annual International Cryptology Conference, Santa Barbara, California, USA, August 15-19, 1999, Proceedings*, volume 1666 of *Lecture Notes in Computer Science*, pages 537–554. Springer, 1999.

[FSS20] Tim Fritzmann, Georg Sigl, and Johanna Sepúlveda. RISQ-V: tightly coupled RISC-V accelerators for post-quantum cryptography. *IACR Trans. Cryptogr. Hardw. Embed. Syst.*, 2020(4):239–280, 2020.

[GJJR11] Gilbert Goodwill, Benjamin Jun, Josh Jaffe, and Pankaj Rohatgi. A testing methodology for side-channel resistance validation. In *NIST non-invasive attack testing workshop*, volume 7, pages 115–136, 2011.

[GKS21] Denisa O. C. Greconici, Matthias J. Kannwischer, and Amber Sprenkels. Compact dilithium implementations on cortex-m3 and cortex-m4. *IACR Trans. Cryptogr. Hardw. Embed. Syst.*, 2021(1):1–24, 2021.

[HHP+21] Mike Hamburg, Julius Hermelink, Robert Primas, Simona Samardjiska, Thomas Schamberger, Silvan Streit, Emanuele Strieder, and Christine van Vredendaal. Chosen ciphertext k-trace attacks on masked CCA2 secure kyber. *IACR Trans. Cryptogr. Hardw. Embed. Syst.*, 2021(4):88–113, 2021.

[Int] RISC-V International. Risc-v developer boards. https://riscv.org/risc-v-developer-boards/.

[Kos19] Nick Kossifidis. rv_sha3, 2019. https://github.com/mickflemm/rv_sha3.

[KSFS22] Patrick Karl, Jonas Schupp, Tim Fritzmann, and Georg Sigl. Post-quantum signatures on RISC-V with hardware acceleration. *IACR Cryptol. ePrint Arch.*, page 538, 2022.

[LMP23] Huimin Li, Nele Mentens, and Stjepan Picek. Maximizing the potential of custom RISC-V vector extensions for speeding up SHA-3 hash functions. In *Design, Automation & Test in Europe Conference & Exhibition, DATE 2023, Antwerp, Belgium, April 17-19, 2023*, pages 1–6. IEEE, 2023.

[LN16] Patrick Longa and Michael Naehrig. Speeding up the number theoretic transform for faster ideal lattice-based cryptography. In Sara Foresti and Giuseppe Persiano, editors, *Cryptology and Network Security - 15th International Conference, CANS 2016, Milan, Italy, November 14-16, 2016, Proceedings*, volume 10052 of *Lecture Notes in Computer Science*, pages 124–139, 2016.

[LS15] Adeline Langlois and Damien Stehlé. Worst-case to average-case reductions for module lattices. *Des. Codes Cryptogr.*, 75(3):565–599, 2015.

[LSG21] Georg Land, Pascal Sasdrich, and Tim Güneysu. A hard crystal - implementing dilithium on reconfigurable hardware. In Vincent Grosso and Thomas Pöppelmann, editors, *Smart Card Research and Advanced Applications - 20th International Conference, CARDIS 2021, Lübeck, Germany, November 11-12, 2021, Revised Selected Papers*, volume 13173 of *Lecture Notes in Computer Science*, pages 210–230. Springer, 2021.

[Lyu09] Vadim Lyubashevsky. Fiat-shamir with aborts: Applications to lattice and factoring-based signatures. In Mitsuru Matsui, editor, *Advances in Cryptology - ASIACRYPT 2009, 15th International Conference on the Theory and Application of Cryptology and Information Security, Tokyo, Japan, December*

6-10, 2009. Proceedings, volume 5912 of *Lecture Notes in Computer Science*, pages 598–616. Springer, 2009.

[MCL+23] Gaoyu Mao, Donglong Chen, Guangyan Li, Wangchen Dai, Abdurrashid Ibrahim Sanka, Çetin Kaya Koç, and Ray C. C. Cheung. High-performance and configurable SW/HW co-design of post-quantum signature crystals-dilithium. *ACM Trans. Reconfigurable Technol. Syst.*, 16(3):44:1–44:28, 2023.

[Mic] Microchip. Polarfire soc fpgas. https://www.microchip.com/en-us/products/fpgas-and-plds/system-on-chip-fpgas/polarfire-soc-fpgas.

[Mic22] Microchip. *PolarFire SoC MSS Technical Reference Manual*, 2022. https://www.microsemi.com/document-portal/doc_view/1245725-polarfire-soc-fpga-mss-technical-reference-manual.

[MP85] Montgomery and L. Peter. Modular multiplication without trial division. *Mathematics of Computation*, 44(170):519–521, 1985.

[NIS15] NIST. Sha-3 standard: Permutation-based hash and extendable-output functions. NIST Technical Series Publications, 2015. https://nvlpubs.nist.gov/nistpubs/FIPS/NIST.FIPS.202.pdf.

[NIS16] NIST. Post-quantum cryptography standardization. NIST Computer Security Resource Center, 2016. https://csrc.nist.gov/projects/post-quantum-cryptography.

[NMZ+21] Pietro Nannipieri, Stefano Di Matteo, Luca Zulberti, Francesco Albicocchi, Sergio Saponara, and Luca Fanucci. A RISC-V post quantum cryptography instruction set extension for number theoretic transform to speed-up CRYSTALS algorithms. *IEEE Access*, 9:150798–150808, 2021.

[oST23a] National Institute of Standards and Technology. Module-lattice-based digital signature standard, 2023. https://csrc.nist.gov/pubs/fips/204/ipd.

[oST23b] National Institute of Standards and Technology. Module-lattice-based key-encapsulation mechanism standard, 2023. https://csrc.nist.gov/pubs/fips/203/ipd.

[PP19] Peter Pessl and Robert Primas. More practical single-trace attacks on the number theoretic transform. In Peter Schwabe and Nicolas Thériault, editors, *Progress in Cryptology - LATINCRYPT 2019 - 6th International Conference on Cryptology and Information Security in Latin America, Santiago de Chile, Chile, October 2-4, 2019, Proceedings*, volume 11774 of *Lecture Notes in Computer Science*, pages 130–149. Springer, 2019.

[RMJ+21] Sara Ricci, Lukas Malina, Petr Jedlicka, David Smékal, Jan Hajny, Peter Cíbik, Petr Dzurenda, and Patrik Dobias. Implementing crystals-dilithium signature scheme on fpgas. In Delphine Reinhardt and Tilo Müller, editors, *ARES 2021: The 16th International Conference on Availability, Reliability and Security, Vienna, Austria, August 17-20, 2021*, pages 1:1–1:11. ACM, 2021.

[RPBC20] Prasanna Ravi, Romain Poussier, Shivam Bhasin, and Anupam Chattopadhyay. On configurable SCA countermeasures against single trace attacks for the NTT - A performance evaluation study over kyber and dilithium on the

ARM cortex-m4. In Lejla Batina, Stjepan Picek, and Mainack Mondal, editors, *Security, Privacy, and Applied Cryptography Engineering - 10th International Conference, SPACE 2020, Kolkata, India, December 17-21, 2020, Proceedings*, volume 12586 of *Lecture Notes in Computer Science*, pages 123–146. Springer, 2020.

[Sho94] P.W. Shor. Algorithms for quantum computation: discrete logarithms and factoring. In *Proceedings 35th Annual Symposium on Foundations of Computer Science*, pages 124–134, 1994.

[SM15] Tobias Schneider and Amir Moradi. Leakage assessment methodology - A clear roadmap for side-channel evaluations. In Tim Güneysu and Helena Handschuh, editors, *Cryptographic Hardware and Embedded Systems - CHES 2015 - 17th International Workshop, Saint-Malo, France, September 13-16, 2015, Proceedings*, volume 9293 of *Lecture Notes in Computer Science*, pages 495–513. Springer, 2015.

[Tea17] CRYSTALS Team. Cryptographic suite for algebraic lattices, 2017. https://pq-crystals.org/index.shtml.

[WZCG22] Tengfei Wang, Chi Zhang, Pei Cao, and Dawu Gu. Efficient implementation of dilithium signature scheme on FPGA soc platform. *IEEE Trans. Very Large Scale Integr. Syst.*, 30(9):1158–1171, 2022.

[XHY+20] Guozhu Xin, Jun Han, Tianyu Yin, Yuchao Zhou, Jianwei Yang, Xu Cheng, and Xiaoyang Zeng. VPQC: A domain-specific vector processor for post-quantum cryptography based on RISC-V architecture. *IEEE Trans. Circuits Syst. I Regul. Pap.*, 67-I(8):2672–2684, 2020.

[XL21] Yufei Xing and Shuguo Li. A compact hardware implementation of cca-secure key exchange mechanism CRYSTALS-KYBER on FPGA. *IACR Trans. Cryptogr. Hardw. Embed. Syst.*, 2021(2):328–356, 2021.

[ZXXH22] Yifan Zhao, Ruiqi Xie, Guozhu Xin, and Jun Han. A high-performance domain-specific processor with matrix extension of RISC-V for module-lwe applications. *IEEE Trans. Circuits Syst. I Regul. Pap.*, 69(7):2871–2884, 2022.

[ZYC+20] Neng Zhang, Bohan Yang, Chen Chen, Shouyi Yin, Shaojun Wei, and Leibo Liu. Highly efficient architecture of newhope-nist on FPGA using low-complexity NTT/INTT. *IACR Trans. Cryptogr. Hardw. Embed. Syst.*, 2020(2):49–72, 2020.

[ZZW+22] Cankun Zhao, Neng Zhang, Hanning Wang, Bohan Yang, Wenping Zhu, Zhengdong Li, Min Zhu, Shouyi Yin, Shaojun Wei, and Leibo Liu. A compact and high-performance hardware architecture for crystals-dilithium. *IACR Trans. Cryptogr. Hardw. Embed. Syst.*, 2022(1):270–295, 2022.

[ZZZ+22] Yihong Zhu, Wenping Zhu, Min Zhu, Chongyang Li, Chenchen Deng, Chen Chen, Shuying Yin, Shouyi Yin, Shaojun Wei, and Leibo Liu. A 28nm 48kops 3.4μj/op agile crypto-processor for post-quantum cryptography on multi-mathematical problems. In *IEEE International Solid-State Circuits Conference, ISSCC 2022, San Francisco, CA, USA, February 20-26, 2022*, pages 514–516. IEEE, 2022.