High-performance and Configurable SW/HW Co-design of Post-quantum Signature CRYSTALS-Dilithium

GAOYU MAO, City University of Hong Kong, China and Zhejiang Lab, China
DONGLONG CHEN, BNU-HKBU United International College, China
GUANGYAN LI, City University of Hong Kong, China
WANGCHEN DAI, Zhejiang Lab, China
ABDURRASHID IBRAHIM SANKA, City University of Hong Kong, China
ÇETIN KAYA KOÇ, UC Santa Barbara, USA, NUAA, China, and Iğdır University, Turkey
RAY C. C. CHEUNG, City University of Hong Kong, China

CRYSTALS-Dilithium is a lattice-based post-quantum digital signature scheme that is resistant to attacks by quantum computers and has been selected to be standardized in the NIST post-quantum cryptography (PQC) standardization process. However, the speed performance and design flexibility of the Dilithium still need to be evaluated. This article presents a high-performance software/hardware co-design of CRYSTALS-Dilithium based on the NIST PQC round-3 parameters. High-speed pipelined hardware modules for NTT/INTT, point-wise multiplication/addition, and for SHAKE are included in the design to accelerate the time-consuming operations in Dilithium. All hardware modules are parameterized, thus allowing full support of runtime configuration to increase versatility. Moreover, the proposed software/hardware architecture and tight operating workflows reduce the data transmission overhead between the processor and other hardware modules. The hardware accelerator is implemented with a reconfigurable logic on FPGA and is integrated with the high-performance ARM Cortex-A9 processor in the Xilinx Zynq Architecture. We measure the performance of the software/hardware system for Dilithium in NIST security levels 2, 3, and 5. Compared to pure software implementations, we achieve 8.7–12.5 times speedup in Key generation, 6.3–7.3 times speedup in Sign, and 9.1–12.2 times speedup in Verify operations.

CCS Concepts: • **Security and privacy → Hardware security implementation**; **Digital signatures**; • **Hardware → Hardware accelerators;**

Additional Key Words and Phrases: Post-quantum cryptography, lattice-based cryptography, digital signature, CRYSTALS-Dilithium, software-hardware co-design

This work is supported by Hong Kong Innovation and Technology Commission (ITF Seed Fund ITS/216/19), City University of Hong Kong (Project Grant Nos. 9440242 and 9678187), National Natural Science Foundation of China (Nos. 62002023 and 62002239), the Guangdong Provincial Key Laboratory of Interdisciplinary Research and Application for Data Science, BNU-HKBU United International College (2022B1212010006), Guangdong Higher Education Upgrading Plan (2021-2025) (UIC R0400001-22), Guangdong Higher Education Key Platform and Research Project (No. 2020KQNCX100).
Authors' addresses: G. Mao, City University of Hong Kong, Hong Kong, China and Zhejiang Lab, Hangzhou, China; email: gaoyumao3-c@my.cityu.edu.hk; D. Chen (corresponding author), BNU-HKBU United International College, Zhuhai, China; email: donglongchen@uic.edu.cn; G. Li, A. I. Sanka, and R. C. C. Cheung, City University of Hong Kong, Hong Kong, China; emails: {guangyali5-c, iasanka2-c}@my.cityu.edu.hk, r.cheung@cityu.edu.hk; W. Dai, Zhejiang Lab, Hangzhou, China; email: w.dai@my.cityu.edu.hk; Ç. K. Koç, UC Santa Barbara, USA and NUAA, China and Iğdır University, Turkey; email: cetinkoc@ucsb.edu.

1936-7406/2023/06-ART44 $15.00
https://doi.org/10.1145/3569456

ACM Reference format:
Gaoyu Mao, Donglong Chen, Guangyan Li, Wangchen Dai, Abdurrashid Ibrahim SANKA, Çetin Kaya Koç, and Ray C. C. Cheung. 2023. High-performance and Configurable SW/HW Co-design of Post-quantum Signature CRYSTALS-Dilithium. *ACM Trans. Reconfig. Technol. Syst.* 16, 3, Article 44 (June 2023), 28 pages.
https://doi.org/10.1145/3569456

1 INTRODUCTION

Public key cryptography provides data confidentiality and authenticity in modern digital communication systems. However, the most widely used public-key algorithms, including the RSA and ECC, can be efficiently broken by running the Shor's algorithm [40] on a quantum computer with a few thousand qubits. Hence, it has become necessary to find suitable alternative cryptosystems before the practical deployment of quantum computers. **Post-quantum cryptography (PQC)** is a term to describe the set of cryptographic algorithms that are secure against quantum attacks [7, 8]. PQC algorithms are divided into different variants, namely, lattice-based cryptography [30], code-based cryptography [33], multivariate cryptography [13], hash-based cryptography [9], and supersingular elliptic curve isogeny cryptography [20]. The **National Institute of Standards and Technology (NIST)** has initiated a process of PQC standardization since 2016. Sixty-nine schemes were selected for the first round of evaluation after 2017. Seven of these schemes have advanced to the third round of evaluation after July 2020. In July 2022, NIST identified four candidate algorithms for standardization, including three lattice-based cryptography: CRYSTALS-KYBER, CRYSTALS-Dilithium, FALCON, and a hash-based signature: SPHINCS+.

Lattice-based cryptography is based on the difficulty of computational lattice problems that cannot be solved efficiently on a conventional digital computer. Examples of such problems include the **shortest vector problem (SVP)** [1], **short integer solution problem (SIS)** [1], and the **learning with error problem (LWE)** [36]. The SIS problem is to find a short vector s such that $A \cdot s = 0$, given the matrix A. The LWE is to find the vector s from $b = A \cdot s + e$, given the matrix A and the vector b, where e is the hidden error vector. The Ring-SIS and Ring-LWE problems [28] define the matrix A over a polynomial ring so it can be obtained under the rotational shift operation of a vector a. This design provides more compactness and efficacy, because there is no need to store the large matrix A and the calculation of $A \cdot s$ can be accelerated by using the **number theoretic transforms (NTT)**. The **Module-SIS (MSIS)** and **Module-LWE (MLWE)** [24] replace the single ring elements (a and s) with the module elements over the same ring. Therefore, there exist tradeoffs between security and efficiency in MSIS and MLWE.

Based on the hardness of the MSIS and MLWE lattice problems, CRYSTALS-Dilithium [3] is designed using the Fiat-Shamir with Aborts technique [26]. It is a digital signature scheme that has been proved secure under the chosen message attacks. The most time-consuming operations in the Dilithium scheme are the **extendable-output function (XOF)** and the matrix/vector multiplication in the polynomial ring. The parameters of polynomial ring and XOF are the same in different security levels but only involve fewer or more operations. The officially submitted Dilithium implementation is given in the C language, and there is an AVX2 optimized version.

The deployment of different software and hardware platforms significantly impacts the performance of the cryptosystems. There are many software and hardware design explorations to evaluate the NIST PQC algorithm standardization process. Software implementation owns the merits of easy portability and short development time, thus normally becomes the first performance evaluation choice. Greconici et al. [18] implemented Dilithium on ARM Cortex-M3 and ARM Cortex-M4 to explore the tradeoff between speed and memory usage strategy. Hardware implementation (e.g., FPGA and ASIC) can easily outperform software implementation in speed and power, although a relatively longer development cycle is required. Soni et al. [41] made a hardware comparison of the NIST PQC signature algorithm Dilithium and qTESLA in FPGA using an HLS-based design methodology. Ricci et al. [37] proposed the first VHDL implementation of Dilithium on FPGA and was a high-speed design.

Software/hardware co-design, however, is a **System on Chip (SoC)** design involving both the software design in a microprocessor such as ARM and RISC-V and the hardware design on an FPGA or ASIC. The software/hardware co-design possesses the advantages of pure software and hardware platforms. Specifically, a parallel and pipelined architecture could be explored to speed up the critical part of the system on the hardware while the remaining non-critical part be implemented in the software in a short development time. Hence, the software/hardware co-design has less time to market than pure hardware [21] and achieves better performance than pure software designs.

Furthermore, the limited hardware resources in FPGAs make the software/hardware co-design a good choice for efficient system implementations. Systems that cannot fit into the desired FPGA could be implemented as a software/hardware co-design with reduced costs (than buying bigger or higher-end FPGA) and optimum performance [2]. Other software/hardware co-design advantages are easier controllability and higher flexibility. The control and configuration section could be put on the software [21]. More flexibility could be achieved by placing the non-critical part susceptible to modifications on the software side while the fixed critical part achieved in the hardware. Hence, by using a software/hardware co-design, the hardware side could easily be subjected to several tests from the software side code. Aysu et al. [2] presented an SW/HW co-design for lattice-based signature to reduce resource utilization and optimize the latency in offline and online application scenarios. Dang et al. [12] implemented three lattice-based PQC algorithms using SW/HW co-design and demonstrated 7–30× speedup over pure software. Mera et al. [29] designed a polynomial multiplication co-processor for the saber and achieved a 6× speedup over the software at a small area cost.

Several software/hardware co-designs exist on NIST round-2 Dilithium, including [46] on ZYNQ-7020 platform with ARM Cortex-A9 processor, and [4] with software on RISC-V processor and hardware on ASIC. In Reference [46], the authors designed hardware architecture for NTT, point-wise addition/multiplication, and SHA-3 Keccak functions. However, the speed improvement was insignificant. The point-wise operations only considered two polynomials, not particularly acceleration for the polynomial matrix and vectors. Meanwhile, the Keccak-related samplers were not implemented in the hardware, which resulted in a large data transmission overhead. In Reference [4], hardware accelerators, including sampling with SHA-3 based **Pseudo-Random Number Generation (PRNG)** and NTT, were designed to adapt the computation of several lattice-based cryptosystems. However, the authors did not provide a dedicated acceleration for the time-consuming polynomial matrix-vector multiplication, resulting in a longer Dilithium computation time.

To further shorten the data transmission overhead and increase the speed of the Dilithium cryptosystem, we propose a high-speed hardware accelerator and integrate it into a flexible SoC architecture. To further benefit the society, our code is open-source and available at https://github.com/CALAS-CityU/SW-HW-Co-design-of-Dilithium. The major contributions of this work are as follows:

- In pursuit of configurability, a flexible SoC architecture is designed for both the software and the hardware computation. A fully parameterized versatile hardware accelerator design enables a runtime configuration to adjust the computation for Dilithium of different security levels.
- To maintain a good speed-area tradeoff, a hybrid NTT/INTT module is designed for both NTT and INTT. The separated NTT and INTT algorithms are combined, and the hybrid

ALGORITHM 1: Dilithium's Key Generation [3]

```
Output: Public key $pk$, Secret key $sk$
 1: $\zeta \leftarrow \{0,1\}^{256}$
 2: $(\rho, \varsigma, K) \in \{0,1\}^{256\times 3} := H_{256}(\zeta)$
 3: $(\mathbf{s}_1, \mathbf{s}_2) \in S_\eta^l \times S_\eta^k := H_{128}(\varsigma)$
 4: $\hat{\mathbf{A}} \in R_q^{k\times l} := H_{128}(\rho)$
 5: $\hat{\mathbf{s}_1}$ =NTT($\mathbf{s}_1$)
 6: $\hat{\mathbf{m}}_1 = \hat{\mathbf{A}} \cdot \hat{\mathbf{s}_1}$    // Polynomial Matrix-Vector Multiplication (Point-wise Multiplication + Point-wise Addition)
 7: $\mathbf{m}_2$ =INTT($\hat{\mathbf{m}}_1$)
 8: $\mathbf{t} := \mathbf{m}_2 + \mathbf{s}_2$    // Point-wise Addition
 9: $(\mathbf{t}_1, \mathbf{t}_0) :=$ Power2Round$_q(\mathbf{t}, d)$
10: $tr \in \{0,1\}^{384} := H_{256}(\rho || \mathbf{t}_1)$
11: Pack $pk = (\rho, \mathbf{t}_1)$, pack $sk = (\rho, K, tr, \mathbf{s}_1, \mathbf{s}_2, \mathbf{t}_0)$
```

architecture can now reuse hardware resources for NTT/INTT computation. Hardware accelerators for the time-consuming SHAKE and point-wise addition/multiplication are designed to speed up the whole Dilithium system. Moreover, all hardware modules are implemented to run in constant time to avoid simple timing attacks.

- To reduce the data transmission overhead, vectorized point-wise adder and multiplier are designed to accommodate different lengths of polynomial matrix-vector multiplication and polynomial vector multiplication/addition/subtraction. This design effectively reduces the data transfer between the software and the hardware. Furthermore, a unified pipeline architecture, which tightly integrates Keccak core with samplers, is designed for the SHAKE. The tightly coupled architecture can effectively reduce the intermediate data transmission between the software and the hardware.
- The proposed design is implemented on Xilinx ZedBoard and evaluates the Dilithium Key generation, Sign, and Verify algorithms performance under three different security levels. Implementation results show that the proposed system could compute Dilithium security level 2 Key generation, Sign, and Verify in 1.10ms, 5.93ms, and 1.17ms, respectively. Compared with the pure software implementation, the proposed software/hardware co-design achieves a speedup of 6.3–33.2×.

The rest of this article is organized as follows: Section 2 briefly introduces the Dilithium algorithms and provides a software profiling of the algorithms. Section 3 presents the hardware design details, including software/hardware co-design architecture, hybrid NTT/INTT module, point-wise multiplication module, point-wise addition module, and SHA-3-based PRNG module. Section 4 discusses the experiment results, including the design and test of hardware and software, operational analysis, and the overall performance of SW/HW co-design. Section 5 compares the design of polynomial operations and complete signature schemes with other related works. Section 6 concludes this article.

2 PRELIMINARIES

2.1 CRYSTALS-Dilithium Signature Algorithm

The Dilithium signature cryptosystem comprises three parts: Key generation, Sign, and Verify. Specifically, Key generation generates public and private keys. Sign uses the private key to sign the message, while the Verify uses the public key to verify the validity of the signature. For ease of understanding from the computation perspective, we enrich the content of these algorithms by adding computational details and describing them in Algorithms 1, 2, and 3, respectively. Table 1 describes the notations and operations in the above algorithms used in the NIST Dilithium

Table 1. Algorithm Notations and Operations Description

Notation or operation	Description
Superscript	Data size.
Subscript	Serial number or parameter.
Normal letter	Scalar.
Bold lower-case letters (e.g., $\mathbf{s}$)	Polynomial vector.
Bold upper-case letters (e.g., $\mathbf{A}$)	Polynomial matrix.
$\|\|$	Concatenate numbers.
$\hat{}$	The number is in NTT domain.
H_{128}	SHAKE128 XOF in SHA-3.
H_{256}	SHAKE258 XOF in SHA-3.
S_η	Rejection eta sampling with coefficient in $[-\eta, \eta]$.
R_q	Rejection uniform sampling with coefficient in $[-q, q]$.
S_{γ_1}	Bit-pack to get number in$[-\gamma_1, \gamma_1)$.
NTT	Transform polynomial to NTT domain.
INTT	Convert the polynomial from NTT domain to normal domain.
Power2 Round	Power of two rounding.
HighBits	Decompose to get high-order bits.
LowBits	Decompose to get low-order bits.
MakeHint	Compute hint for overflow bits.
UseHint	Use hint to correct overflow bits.
Sample InBall	Sample polynomial with τ nonzero coefficients in {−1,1}.

Table 2. Parameters of Dilithium [3]

NIST Security Level	2	3	5
Parameters			
q [modulus]	8,380,417	8,380,417	8,380,417
d [dropped bits from t]	13	13	13
τ [# of ±1's in c]	39	40	60
γ_1 [y coefficient range]	2^{17}	2^{19}	2^{19}
γ_2 [lower rounding range]	$(q-1)/88$	$(q-1)/32$	$(q-1)/32$
(k,l) [dimensions of A]	$(4,4)$	$(6,5)$	$(8,7)$
η [secrete key range]	2	4	2
β [$\tau \cdot \eta$]	78	196	120
ω [max # of 1's in h]	80	55	75

reference C code [3]. Note that Dilithium has three different security levels, which provides a tradeoff in security and performance. Table 2 lists the parameter values in different security levels.

In Algorithm 1, the ζ is a 256-bit true random number (i.e., Step 1) and is expanded by the SHAKE256 to get the ρ, ς, K (i.e., Step 2). The ς is extended by the SHAKE128, which generates short vectors $\mathbf{s}_1, \mathbf{s}_2$ after rejection sampling (i.e., Step 3). The ρ is extended by the SHAKE128 to generate the polynomial matrix $\mathbf{A}$ after rejection sampling (i.e., Step 4). Because Dilithium is based on the MLWE problem, $\mathbf{A}$ is a polynomial matrix, not a vector. NTT is used in polynomial matrix-vector multiplication (i.e., Steps 5–7). Note that $\mathbf{A}$ is sampled in the NTT domain, no further transformation is needed. The Power2Round breaks up high and low bits to shrink the key size (i.e., Step 9). The outputs pk and sk are packed and stored for Sign and Verify (i.e., Step 11).

ALGORITHM 2: Dilithium's Sign [3]

```
Input: Secret key $sk$, Message $M$
Output: Signature $\sigma$
 1: Unpack $sk = (\rho, K, tr, \mathbf{s}_1, \mathbf{s}_2, \mathbf{t}_0)$
 2: $\mu \in \{0,1\}^{384} := H_{256}(tr||M)$
 3: $\rho' \in \{0,1\}^{384} := H_{256}(K||\mu)$
 4: $\hat{\mathbf{A}} \in R_q^{k\times l} := H_{128}(\rho)$
 5: $\hat{\mathbf{s_1}}$=NTT($\mathbf{s}_1$), $\hat{\mathbf{s_2}}$ =NTT($\mathbf{s}_2$), $\hat{\mathbf{t}_0}$ =NTT($\mathbf{t}_0$)
 6: $\kappa := 0$, $(\mathbf{z}, \mathbf{h}) := \perp$
 7: while $(\mathbf{z}, \mathbf{h}) = \perp$ do
 8:     $\mathbf{y} \in \widetilde{S}_{\gamma_1}^l := H_{256}(\rho', \kappa)$
 9:     $\hat{\mathbf{y}}$ =NTT($\mathbf{y}$)
10:     $\hat{\mathbf{w}} := \hat{\mathbf{A}} \cdot \hat{\mathbf{y}}$        // Polynomial Matrix-Vector Multiplication (Point-wise Multiplication + Point-wise Addition)
11:     $\mathbf{w}$ =INTT($\hat{\mathbf{w}}$)
12:     $\mathbf{w}_1 := \text{HighBits}_q(\mathbf{w}, 2\gamma_2)$
13:     $\widetilde{c} \in \{0,1\}^{256} := H_{256}(\mu||\mathbf{w}_1)$
14:     $c \in B_\tau$ :=SampleInBall($\widetilde{c}$)
15:     $\hat{c}$ =NTT($c$)
16:     $\mathbf{v}_1$ =INTT($\hat{c} \cdot \hat{\mathbf{s_1}}$)        // Polynomial Vector Multiplication (Point-wise Multiplication)
17:     $\mathbf{z} := \mathbf{y} + \mathbf{v}_1$        // Point-wise Addition
18:     $\mathbf{v}_2$ =INTT($\hat{c} \cdot \hat{\mathbf{s_2}}$)        // Polynomial Vector Multiplication (Point-wise Multiplication)
19:     $\mathbf{v}_3 = \mathbf{w} - \mathbf{v}_2$        // Point-wise Subtraction
20:     $\mathbf{r}_0$ :=LowBits$_q(\mathbf{v}_3, 2\gamma_2)$
21:     if $||\mathbf{z}||_\infty \geq \gamma_1 - \beta$ or $||\mathbf{r}_0||_\infty \geq \gamma_2 - \beta$ then
22:         $(\mathbf{z}, \mathbf{h}) := \perp$
23:     else
24:         $\mathbf{v}_4$ =INTT($\hat{c} \cdot \hat{\mathbf{t}_0}$)        // Polynomial Vector Multiplication (Point-wise Multiplication)
25:         $h := \text{MakeHint}_q(-\mathbf{v}_4, \mathbf{v}_3 + \mathbf{v}_4, 2\gamma_2)$        // Point-wise Addition
26:         if $||\mathbf{v}_4||_\infty \geq \gamma_2$ or the # of 1's in $h \geq \omega$ then
27:             $(\mathbf{z}, \mathbf{h}) = \perp$
28:         end if
29:     end if
30:     $\kappa := \kappa + l$
31: end while
32: Pack $\sigma = (\mathbf{z}, \mathbf{h}, \widetilde{c})$
```

In Algorithm 2, the sk is unpacked for Sign (i.e., Step 1). The SHAKE256 is used for hashing input messages and keys (i.e., Steps 2–3). The masking vector $\mathbf{y}$ is expanded from ρ', κ by using the SHAKE256, and its coefficients are within the range $[-\gamma_1, \gamma_1)$ (i.e., Step 8). The polynomial matrix-vector multiplication $\mathbf{A} \cdot \mathbf{y}$ is calculated and the HighBits is used to get the high-order bits $\mathbf{w}_1$ (i.e., Steps 9–12). The challenge c is obtained by hashing the $tr, M, \mathbf{w}_1$ with the SHAKE256, then sampled with τ random positions to be ± 1 and the others be 0 (i.e., Steps 13–14). The c is used to generate the potential signature $\mathbf{z}$ (i.e., Steps 15–17). Note that fewer bits are used to store the signature; it needs to generate the hints $\mathbf{h}$ before compression to ensure the correctness in Verify (i.e., Step 25). There are four conditions to check whether $\mathbf{z}$ will leak information (i.e., Steps 21, 26). If yes, then the signature will be rejected and then generated again.

In Algorithm 3, public key pk and signature σ are unpacked for Verify (i.e., Steps 1–2). The message M and public key are hashed with the SHAKE256 (i.e., Step 3). The NTT is used to calculate $\mathbf{Az} - c\mathbf{t}$ (i.e., Steps 5–10). The hint $\mathbf{h}$ is used to correct calculation errors in data compression (i.e., Step 10). There are three conditions to check whether the obtained signature can meet the security requirements (i.e., Step 12). If the security requirements are not satisfied simultaneously, then the signature will be rejected.

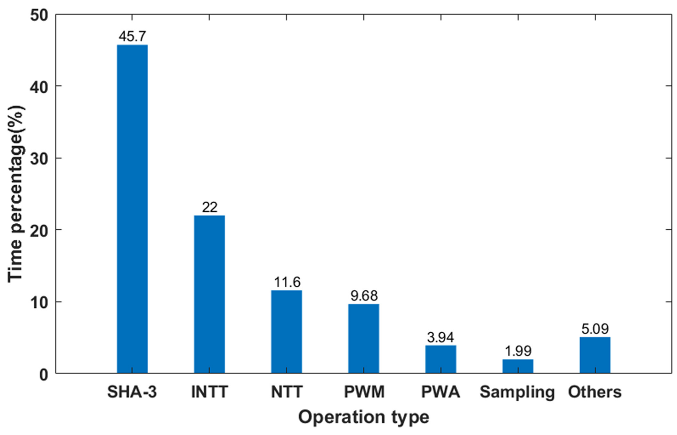


Fig. 1. Software profiling results of Dilithium.

ALGORITHM 3: Dilithium's Verify [3]

```
Input: Public key $sk$, Message $M$, signature $\sigma$
Output: The validity of the signature
 1: Unpack $\sigma = (\mathbf{z}, \mathbf{h}, \widetilde{c})$
 2: Unpack $pk = (\rho, \mathbf{t}_1)$
 3: $\mu \in \{0,1\}^{384} := H_{256}(H_{256}(\rho||\mathbf{t}_1)||M)$
 4: $\hat{\mathbf{A}} \in R_q^{k\times l} := H_{128}(\rho)$
 5: $\hat{\mathbf{z}}$ =NTT($\mathbf{z}$)
 6: $\hat{\mathbf{w}}_1 := \hat{\mathbf{A}} \cdot \hat{\mathbf{z}}$    // Polynomial Matrix-vector Multiplication (Point-wise Multiplication + Point-wise Addition)
 7: $c$ :=SampleInBall($\widetilde{c}$)
 8: $\hat{c}$ =NTT($c$), $\hat{\mathbf{t}_1}$ =NTT($\mathbf{t}_1 \cdot 2^d$)
 9: $\mathbf{w}_2 = \hat{c} \cdot \hat{\mathbf{t}_1}$    // Polynomial Vector Multiplication (Point-wise Multiplication)
10: $\mathbf{w}'$ :=UseHint$_q(\mathbf{h}, \mathbf{w}_1 - \mathbf{w}_2, 2\gamma_2)$    // Point-wise Subtraction
11: $c_2 = H_{256}(\mu||\mathbf{w}')$
12: Return[$||\mathbf{z}||_\infty < \gamma_1 - \beta$] and [$\widetilde{c} = c_2$] and [# of 1's in $\mathbf{h}$ is $\leq \omega$]
```

2.2 Software Profiling and Design Analysis

Before dividing the workload between the software and the hardware, it is important to first analyze the schedule and data dependency of the algorithm, conduct profiling, and identify the time-consuming functions in the system. We choose the Dilithium reference C code implementation [3] in NIST Security Level 3 and profile it with the TCF profiler on Xilinx Vitis Platform. We execute the Key generation, Sign and Verify algorithms 1,000 times on the ARM Cortex-A9 processor (with cache on) at 666 MHz and obtain the time percentage of each operation, as summarized in Figure 1.

As shown in Figure 1, the most time-consuming part is the SHA-3 related operation, including the SHAKE128/SHAKE256 permutation, input absorb, and output store functions. The second is the INTT operation, and the third is the NTT operation. Both NTT and INTT operations include modular multiplication and occupy around 34% of the computing time. The fourth is the **point-wise multiplication (PWM)** operation, which takes 9.68% of the time. The fifth is the **point-wise addition (PWA)** operation with the subsequent modular operations (the PWA also includes the point-wise subtraction operation). Sampling operation occupies around 2% of the total time, which includes the rejection eta sampling and rejection uniform sampling. There are 5% remaining operations are listed as Others in Figure 1, such as signature pack operation for 0.57%, signature

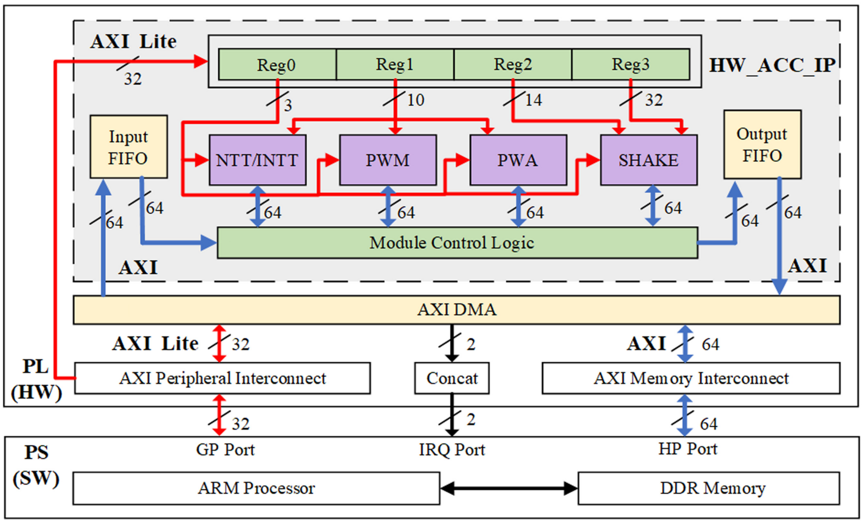


Fig. 2. The top-level SW/HW co-design SoC architecture.

unpack operation for 1.39%, the decompose operation in make/use hint for 1.08%, check norm operation in signature checking for 0.56%, the SampleInBall operation for 0.06%, and so on.

Based on the Profiling results, hardware modules are designed to accelerate the six most timing-consuming operations, which add up to 94.91% of the total execution time. They are the hybrid NTT/INTT module for both NTT and INTT operation, the PWM module for point-wise multiplication operation, the PWA module for point-wise addition/subtraction operation, and the SHAKE module for SHA-3 operation and sampling operation. For the other operations, they are neither time-consuming nor friendly to hardware design, so we keep them running in the software. To increase the design flexibility to support Dilithium computation for all the security levels, we parameterize the hardware modules to support runtime configuration.

3 HARDWARE ACCELERATION ARCHITECTURE

3.1 Run-time Configurable SW/HW Co-Design Architecture

The top-level software/hardware co-design architecture is shown in Figure 2. The proposed system is designed according to the Xilinx Zynq SoC architecture, which includes the **Processing System (PS)** and the **Programmable Logic (PL)**. The **Advanced eXtensible Interface (AXI)** standard is used to interconnect the PS and PL. The software runs on the ARM processor on the PS, while the designed hardware accelerator runs on the reconfigurable logic on the PL.

On the PS side, the processor accesses the data in the DDR for computation. The processor includes a cache to store temporary data for acceleration. The IRQ port is used to answer the interrupt request from the PL. The HP port is a high-performance interface that connects to the DDR controller. It could read and write a large amount of data in memory through the AXI protocol. The GP port is a general-purpose low-performance interface that can read and write registers on the PL through the AXI-Lite protocol.

On the PL side, DMA is the intermedium for data communication with DDR and is connected to the HP port using AXI stream protocol. The DMA interacts with the hardware accelerator through input and output FIFOs. The read and write interrupt signals of the DMA pass to the IRQ port through the concat IP. The processor controls the DMA data transfer and passes configured parameters via the GP ports using AXI lite protocol. The AXI memory interconnect and AXI

Table 3. Control Register Definition

Register	Control signal	Width	Description
Reg0	start_module	3	Initiate the start/stop of the corresponding modules.
Reg1	ntt_sel	1	Select the NTT or INTT function of the hybrid NTT/INTT module.
	pwm_vector_len	4	Determine the polynomial vector length in the PWM module.
	pwa_add_sub_sel	1	Determine the polynomial vector length in the PWA module.
	pwa_vector_len	4	Select the addition or subtraction operation in the PWA module.
Reg2	shake_mode	2	Decide the types of function in the SHA-3 family.
	sampler_sel	1	Choose the sampler type is uniform rejection sampling or eta rejection sampling.
	sampler_eta	1	Set the parameter in the eta rejection sampler.
	shake_write_len	10	Define the number of output bytes writing to the output FIFO in the SHAKE module.
Reg3	shake_read_len	32	Define the number of bytes that the SHAKE module accept.

peripheral interconnect are the intermediate medium between the endpoint IPs and the PS. Their main tasks include memory mapping, bit width conversion, and clock conversion. The AXI stream data transmission in this design uses a 64-bit bus, while the AXI lite control signal uses a 32-bit bus.

The HW_ACC_IP consists of input and output FIFOs, a hardware accelerator, control registers, and the module control logic. The hardware accelerator contains four modules, including the hybrid NTT/INTT, PWM, PWA, and SHAKE. Each module works independently. All modules work with the same input and output FIFOs. The module control logic is an arbiter designed to convey control information between the PS and different acceleration modules. The design configurability is achieved through control registers, which are used to convey control signals and design parameters. The four control registers are defined as shown in Table 3. The register0 is to use 3-bit to control the startup of four modules, and the other registers are used to convey parameters settings of different modules.

3.2 Hybrid NTT/INTT Hardware Architecture

NTT is generally a **Discrete Fourier Transform (DFT)** over an integer field or ring [25, 35, 39]. NTT is commonly used to accelerate the multiplication of two polynomials: The classical schoolbook polynomial multiplication has a complexity of $O(n^2)$, while the NTT can reduce it to $O(n \log n)$. The NTT is defined over polynomial ring $R_q = Z_q[x]/(x^n + 1)$, where the coefficient is under modulo q and the polynomial degree is smaller than n. The normal domain polynomial coefficients are denoted as a_i, and the NTT domain polynomial coefficients are represented as $\hat{a}_i$, where $i = 0, 1, \ldots, n-1$. The NTT is computed as $\hat{a}_i = \sum_{j=0}^{n-1} a_j \omega^{ij}$ in Z_q, while the INTT calculates $a_i = n^{-1} \sum_{j=0}^{n-1} \hat{a}_j \omega^{-ij}$ in Z_q, where ω is the primitive root of unity in R_q.

When directly applying NTT in the polynomial multiplication, it requires n zeros appended to each input, which doubles the length of the inputs and requires additional reduction to the ring R_q. To address these issues, the **negative wrapped convolution (NWC)** method [27] can be explored. By applying NWC in polynomial multiplication, one needs to first perform point-wise multiplications of a_i and γ^i, where the γ is the square root of ω, then transform two polynomials $a(x)$ and $b(x)$ into NTT domain to get $\hat{a}(x)$ and $\hat{b}(x)$. Next, point-wise multiply these two polynomials and get $\hat{c}(x)$. After that, use INTT to transform the results back to the normal domain and get the results $c(x)$. Last step is to perform point-wise multiplication of c_i and γ^{-i}.

Applying NWC requires the multiplication of γ^i before NTT and the multiplication of γ^{-i} after INTT. The previous work in Reference [38] showed multiplication of γ^i can be merged inside the decimation-in-time NTT based on the **Cooley-Tukey (CT)** [11] butterfly and the work in

Reference [34] showed the multiplication of γ^{-i} can be merged into the decimation-in-frequency INTT, which is based on **Gentle-Sande (GS)** [17] butterfly. For the CT structure, the multiplication takes place before the add/subtract operation (i.e., $a_1 + a_2 \cdot \omega$, $a_1 - a_2 \cdot \omega$). For the GS structure, the multiplication takes places only after subtract operation (i.e., $a_1 + a_2$, $(a_1 - a_2) \cdot \omega$).

When deploying NTT into hardware implementation, speed performance and resource consumption are important factors. By applying CT structure in NTT and GS structure in INTT, the speed can be increased but the resources consumption is doubled. Based on the work in References [23, 42, 44], a hybrid NTT/INTT algorithm is proposed for hardware implementation, which combines control logic and butterfly computation unit for both NTT and INTT. By designing a combined hardware module, the resources are saved and some multiplexers are introduced for function selection. The twiddle factors for both NTT and INTT are pre-computed and stored in the ROM memory. It is worth noting that in NTT, the pre-computed twiddle factor is obtained by first calculating zeta$[i] = \gamma^i, i = 0, 1, \ldots, n-1$ and then switching the coefficient order through the bit reverse function. INTT first calculates zeta$[i] = \gamma^i, i = n, n+1, \ldots, 2n-1$ and then performs bit reverse operation. Since $\gamma^n \equiv -1 \bmod q$, one could deduce the pre-computed twiddle factors of INTT from NTT by flipping the sign bit. Using this method, we could reduce storage space for the twiddle factors by half compared with the traditional method.

ALGORITHM 4: Hybrid NTT/INTT Algorithm

```
Input: $a(x)$ with coefficients $\{a_1, a_2 \cdots a_n\}$, or $\hat{a}(x)$ with coefficients $\{\hat{a}_1, \hat{a}_2 \cdots \hat{a}_n\}$
Input: Pre-computed twiddle factor zeta$[i] = \gamma^{Bit\,Reverse[i]}$;
Output: NTT($a(x)$) or INTT($\hat{a}(x)$)
 1: Initialization $k \leftarrow 0$ or $n$
 2: for m = 0; m < $log_2 n$ ; m++ do
 3:     len ← $(\frac{n}{2} >> m)$ or $(1 << m)$
 4:     for i = 0; i < n; i = j + len do
 5:         $\omega \leftarrow$ zeta[++$k$] or q - zeta[- -$k$]
 6:         for j = i; j < i + len; j++ do
 7:             $r_1 \leftarrow a_{j+len}$ or $(\hat{a}_j - \hat{a}_{j+len})/2$
 8:             $u_1 \leftarrow r_1 \cdot \omega$        // Modular Multiplication (Multiplication + Modular Reduction)
 9:             $r_2 \leftarrow u_1$ or $\hat{a}_{j+len}$
10:             $u_2 \leftarrow a_j + r_2$ or $\hat{a}_j + r_2$
11:             $t_1 \leftarrow u_2$ or $\frac{u_2}{2}$
12:             $t_2 \leftarrow (a_j - u_1)$ or $u_1$
13:             $a_j$ or $\hat{a}_j \leftarrow t_1$
14:             $a_{j+len}$ or $\hat{a}_{j+len} \leftarrow t_2$
15:         end for
16:     end for
17: end for
```

In Algorithm 4, the polynomial length n is 256, and the primitive $2n$-th root of unity γ is 1,753 in $\mathbb{Z}_q$. The arithmetic is performed under modulus q, which is the prime number 8,380,417 = $2^{23} - 2^{13} + 1$. The unified butterfly structure takes a_1, a_2, ω as inputs, calculates $a_1 + a_2 \cdot \omega$, $a_1 - a_2 \cdot \omega$ in NTT, and calculates $(\hat{a}_1 + \hat{a}_2)/2$, $(\hat{a}_1 - \hat{a}_2) \cdot \omega/2$ in INTT (i.e., Steps 7–14). The multiplication of n^{-1} in INTT is integrated into the butterfly structure by the multiplication of 1/2, which is achieved by $(u_2 >> 1)$ when u_2 is even or $(u_2 >> 1) + (q+1)/2$ when u_2 is odd. The modular reduction after the multiplication of r_1 and ω (i.e., Step 8) is expensive in hardware. The commonly used Barrett reduction [5] and Montgomery modular multiplication [31] algorithms require additional multiplication and conditional subtraction. We utilize the property of ($2^{23} \equiv 2^{13} - 1 \bmod q$) and obtain an efficient modular reduction algorithm by splitting large-bit operation into small-bit operation without additional multiplication, as shown in Algorithm 5.

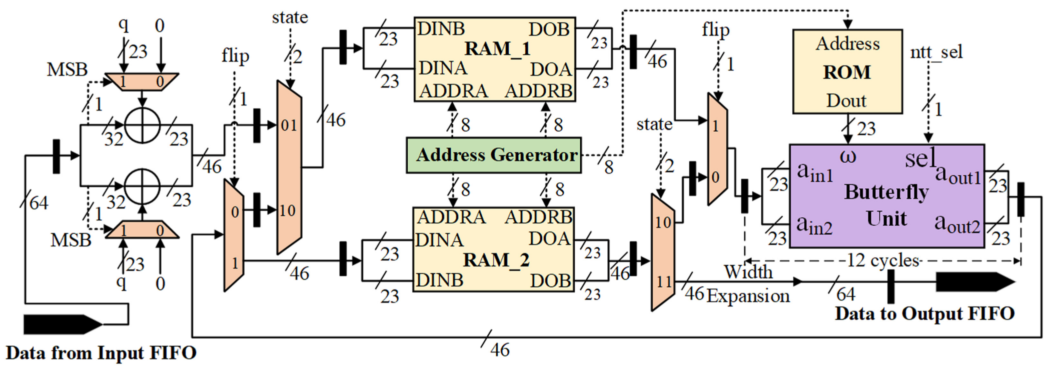


Fig. 3. Hybrid NTT/INTT hardware design architecture.

ALGORITHM 5: Proposed Efficient Modular Reduction in $\mathbb{Z}_{2^{23}-2^{13}+1}$

```
Input: $a[45:0]$
Input: $q[22:0] = 2^{23} - 2^{13} + 1$
Output: $r = a \bmod q$
 1: $c[13:0] = a[45:33] + a[32:23]$
 2: $e[10:0] = c[13:10] + c[9:0]$
 3: $f[22:0] = 2^{13} \cdot (e[10] + e[9:0]) - (e[10] + c[13:10])$
 4: $x[23:0] = f[22:0] + a[22:0]$
 5: if $x \geq q$ then
 6:     $x[22:0] = x[23:0] - q[22:0]$
 7: end if
 8: $d[22:0] = a[45:33] + a[45:23]$
 9: $r[23:0] = x[22:0] - d[22:0]$
10: if $r \geq q$ then
11:     $r[23:0] = r[23:0] - q[22:0]$
12: end if
13: if $r < 0$ then
14:     $r[22:0] = r[23:0] + q[22:0]$
15: end if
16: Return $r$
```

The designed NTT hardware architecture is shown in Figure 3. First, data is read from the 64-bit input FIFO and stored in the RAM_1. The polynomial coefficients are stored as 32-bit integers in the processor, so each 64-bit input data contains two polynomial coefficients. The two 32-bit polynomial coefficients are transformed under the modular q to cut the bit length to 23-bit. Second, the data is read from one RAM to the butterfly unit for computation. At the same time, the butterfly unit writes the computed data into the other RAM. The different operations in NTT and INTT are selected by the multiplexer in the Butterfly unit, controlled by the 1-bit ntt_sel signal. Two RAMs take turns to read and write, as controlled by the 1-bit flip signal. Finally, the 46-bit output data from the RAM_2 is expanded to 64-bit, then written into the output FIFO. The single-port ROM stores the pre-computed twiddle factor ω. The width of the ROM is 23-bit, and the depth is $n = 256$. Two RAMs are dual-port with a width of 23-bit and a depth of $\frac{n}{2} = 128$. A **finite-state machine (FSM)** is designed to manage these three working states. Counters are designed to control each clock cycle, so each state consumes a fixed number of cycles. The modular reduction unit designed according to Algorithm 5 contains in the butterfly unit, which consumes five clock cycles during the reduction in pipelined mode.

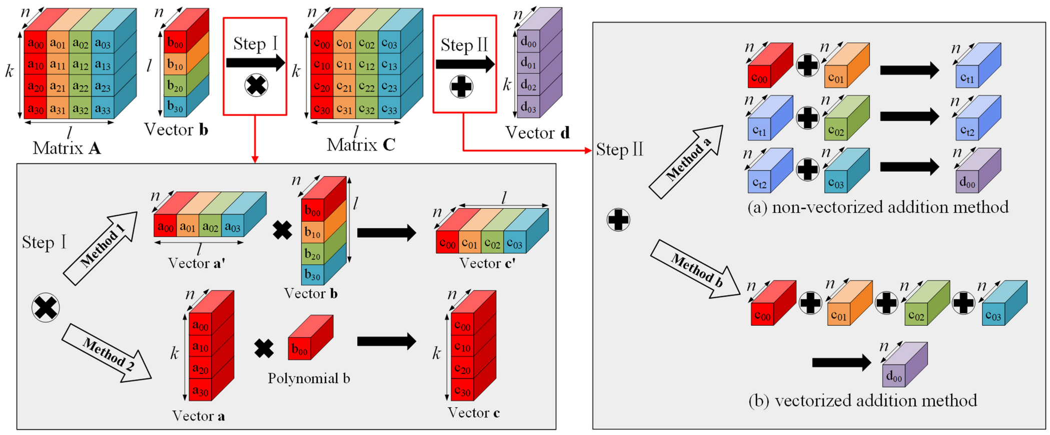


Fig. 4. Polynomial matrix-vector multiplication.

3.3 Vectorized PWM and PWA Hardware Architecture

In Dilithium, a large number of polynomial matrices and polynomial vectors require the calculation of **point-wise multiplication (PWM)** and **point-wise addition (PWA)**. Hence, an efficient hardware design to accelerate these computations is essential to a high-speed Dilithium system.

We use the array $a[n]$ to represent all the n coefficients and $a[i]$ to be one of the coefficient from polynomial $a(x)$, where $a(x) = \sum_{i=0}^{n-1} a[i]x^i$. We let $\mathbf{a}$ be a polynomial column vector and the coefficients of $\mathbf{a}$ are stored in a two-dimensional array $a[l][n]$, where l is the column length of the vector. Set $\mathbf{A}$ to be a polynomial matrix, and its coefficients are stored in a three-dimensional array $a[k][l][n]$. Assume the input polynomial coefficients are $a[n]$ and $b[n]$, and the output polynomial coefficients are $c[n]$. Then, we need to compute $c[i] = a[i] \cdot b[i] \bmod q$ in PWM, while in PWA, we compute $c[i] = a[i] + b[i] \bmod q$.

In the Dilithium software reference design, each function only completes one PWM/PWA of two polynomials, ensuring flexibility in software. However, we could explore a parallel architecture to accelerate these computations in hardware. Take the polynomial matrix-vector multiplication shown in Figure 4 as an example. Let $\mathbf{A}$ be a $k \times l$ polynomial matrix, and a_{ij} represent polynomial with length n. $\mathbf{b}$ is a polynomial vector, and b_{ij} is the polynomial with length n. The polynomial matrix-vector multiplication is divided into two steps. In step I (multiplication), each row of the polynomial matrix $\mathbf{A}$ is multiplied by the polynomial vector $\mathbf{b}$ to get a row of polynomial vector in polynomial matrix $\mathbf{C}$. In step II (addition), the polynomial vectors of each row in matrix $\mathbf{C}$ are added correspondingly to obtain the polynomial column vector $\mathbf{d}$.

There are two methods to compute the multiplication of step I. In method 1, one row of the matrix $\mathbf{A}$ is taken and multiplied by the column vector $\mathbf{b}$; in method 2, one column of matrix $\mathbf{A}$ is taken and multiplied by one polynomial in the column vector $\mathbf{b}$. Both methods need to transmit $k \times l \times n$ coefficients of the matrix $\mathbf{A}$. However, for vector $\mathbf{b}$, method 1 needs to transmit $k \times l \times n$ coefficients, while method 2 only needs to transmit $l \times n$ coefficients. In method 2, the polynomial b is reused to multiply with the column vector of length k, so the data transmission overhead of vector $\mathbf{b}$ is only $1/k$ times of method 1. Therefore, we designed the hardware modules for PWM according to method 2, which significantly reduces the number of data transfers.

The PWM algorithm is designed as shown in Algorithm 6. The vector length k is configurable: When $k = 1$, it is used to accelerate the point-wise multiplication of two polynomials. When k is greater than 1, it is used to compute the point-wise multiplication of the polynomial column vector and the polynomial. In our design, the transmitted polynomial a is reused to multiply with

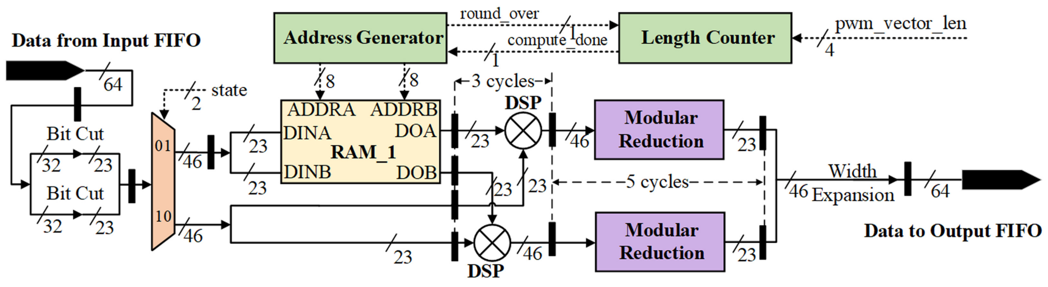


Fig. 5. Point-wise multiplication design architecture.

all the polynomials in vector **b**, thus reducing the transmission of polynomial a from k to only 1 time. Based on the aforementioned method, we design the hardware architecture of PWM as shown in Figure 5. The length counter is a 4-bit counter to count the length of a polynomial vector (corresponding to Step 1 in Algorithm 6). The address generator is an 8-bit counter to generate the read/write address of the RAM_1 (corresponding to Step 2 in Algorithm 6). Note that the coefficients of polynomial a are stored in the RAM_1, while the coefficients of **b** are read directly from the input FIFO. There are three control states in the PWM module. First, the polynomial a is read from the input FIFO to the RAM_1 in the read state. Second, two multipliers receive data from the Input FIFO and the RAM_1. The multipliers complete the point-wise multiplication and modular reduction operations in the multiplication state. At the same time, the multiplication results are written to the output FIFO. Reading, computing, and writing the data are carried out in a pipelined manner. The main computation is to perform multiplication with DSPs and do modular reduction afterwards (corresponding to Step 3 in Algorithm 6). For the two modular multiplication units, one is shared with the NTT module, and the other is designed in this module.

ALGORITHM 6: Vectorized Point-wise Multiplication

```
Input: Polynomial $a$ with coefficient array $a[n]$
Input: Polynomial vector $\mathbf{b}$ with coefficient array $b[k][n]$
Output: Polynomial vector $\mathbf{r} = \mathbf{b} \cdot a$
1: for $i = 0; i < k; i{+}{+}$      // Vectorized read counter design in hardware. The $k$ is configurable using software.
2:     for $j = 0; j < n; j{+}{+}$      // Single polynomial read counter design in hardware. The $n$ is fixed to 256 in hardware.
3:         $r[i][j] = a[j] \cdot b[i][j] \bmod q$      // Parallel multiplication unit design in hardware (two modular multipliers).
4:     end for
5: end for
6: Return $\mathbf{r}$ with coefficient array $r[k][n]$
```

There are two methods to compute the polynomial addition (i.e., step II) in Figure 4. In method a, two polynomials c_{00} and c_{01} are transmitted from the software to the hardware for computation first. Then, the temporary result c_{t1} is transmitted back to software. Next, the polynomials c_{t1} and c_{02} are transmitted and computed following the above process repeatedly until the end of the computation. This method is flexible for parameterized design but requires a $3(l-1)\times n$ coefficients transmission overhead.

To reduce the transmission workload, we propose method b, in which the temporary results are kept in hardware for further reuse. Only the polynomials in the same row and the final results are transmitted. Both methods need to transmit $l \times n$ input coefficients and n output coefficients. However, method a needs to additionally transmit $(l-2) \times n$ intermediate input coefficients and $(l--2) \times n$ intermediate output coefficients. Therefore, the vectorized PWA is designed according to method b to reduce the number of data transfers.

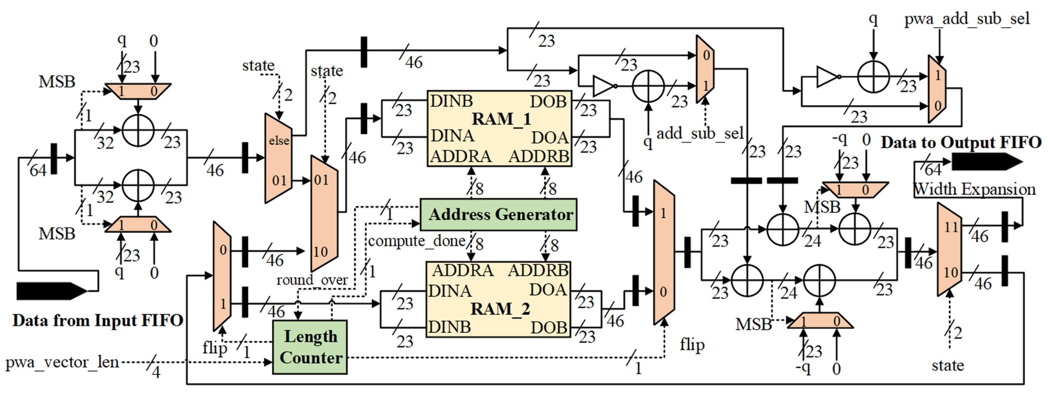
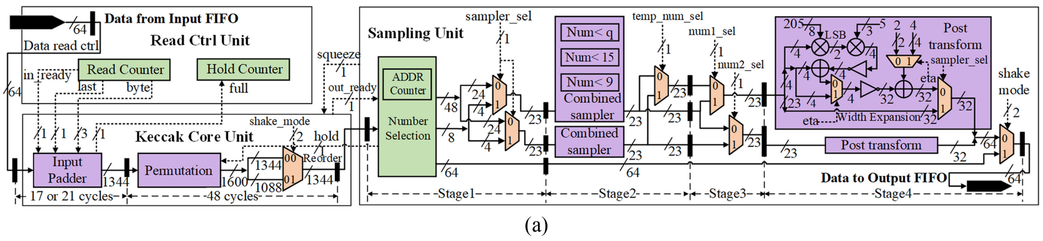


Fig. 6. Point-wise addition design architecture.

ALGORITHM 7: Vectorized Point-wise Addition

```
Input: Polynomial $a$ with coefficient array $a[n]$
Input: Polynomial vector b with coefficient array $b[l-1][n]$
Output: Polynomial $r$ with coefficient array $r[n]$
 1: for $i = 0; i < l-1; i{+}{+}$          // Vectorized read counter design in hardware. The $l$ is configurable using software.
 2:     for $j = 0; j < n; j{+}{+}$      // Single polynomial read counter design in hardware. The $n$ is fixed to 256 in hardware.
 3:         if $i == 0$
 4:             $r[j] = a[j] \pm b[i][j] \bmod q$          // Parallel addition unit design in hardware (two modular adders).
 5:         else
 6:             $r[j] = r[j] + b[i][j] \bmod q$                  // Two modular adders (reuse the two adders in Step 4).
 7:         end if
 8:     end for
 9: end for
10: Return $r$
```

The PWA algorithm is shown in Algorithm 7. The PWA algorithm could perform different computations: When $l = 2$ and configured as addition/subtraction, point-wise addition/subtraction of two polynomials is computed; when l is greater than 2, point-wise addition of polynomial vector of length l is conducted. From the above analysis, our vectorized addition method can reduce the data transmission from $3(l-1) \times n$ coefficients to $(l+1) \times n$ coefficients. Based on Algorithm 7, we design the hardware architecture of PWA as shown in Figure 6. The length counter is a 4-bit counter to count the length of a polynomial vector (corresponding to Step 1 in Algorithm 7). The address generator is an 8-bit counter to generate the read/write address of the RAM_1 and RAM_2 (corresponding to Step 2 in Algorithm 7). There are three computation states in hardware. The polynomial is read from the input FIFO into the RAM_1 in the read state. If $l = 2$, then it will directly enter the final state to complete point-wise addition/subtraction and write data into the output FIFO (corresponding to Step 4 in Algorithm 7). Subtraction is achieved by taking the negative of the input FIFO data and then by addition. If l is greater than 2, it will first enter the second add state after the first read state, perform point-wise additions (corresponding to the Step 5 in Algorithm 7), and then enter the final state to complete the last set of point-wise addition and write data to the output FIFO at the same time (corresponding to the Step 4 in Algorithm 7). Two adders receive data from the RAM and the input FIFO when performing point-wise addition, and then another two adders are used to compute modular reduction over q. The RAM_1 and RAM_2 take turns sending and receiving data in add_state, controlled by the 1-bit flip signal.



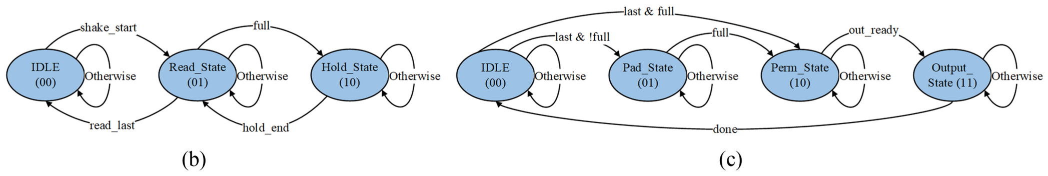


Fig. 7. SHAKE design diagram. (a) Hardware architecture and pipeline design of the SHAKE module. (b) SHAKE input control state diagram. (c) SHAKE output control state diagram.

3.4 SHAKE Module with Unified Sampling

SHAKE functions include SHAKE128 and SHAKE256. They are extendable-output functions based on the Keccak algorithm in SHA-3 family, which takes any input size and generates any output length. Based on the profiling results in Figure 1, it is the most time-consuming function, and accelerating this function would significantly improve the overall system performance. In the Dilithium algorithm, the SHAKE256 generates random seeds, and its outputs can be used by other operations directly. However, the SHAKE128 is used to generate numbers such as the polynomial matrix A, short vectors s_1, and s_2, which should satisfy some specific requirements. In this case, the outputs of the SHAKE128 need to be sampled to meet the corresponding requirements. The Keccak function and the samplers in SHAKE are implemented separately in the software implementation. First, the Keccak function generates a certain number of random seeds. Then the seeds pass through the samplers for sampling. If the output cannot meet the requirements after sampling, then the aforementioned operations need to be performed again. However, in the software/hardware co-design, if the Keccak function and the sampler are implemented separately, then the data transmission overhead would be non-negligible. In addition, extra control logic and space for restoration are required. Therefore, the proposed design tightly combines the Keccak function and samplers into one module to save transmission time and design space.

The hardware design of SHAKE is shown in Figure 7. Figure 7(a) shows the data flow of the SHAKE module. This module mainly consists of three units, including the read ctrl unit, the Keecak core unit, and the sampling unit. The read ctrl unit controls the read data flow, which is shown in Figure 7(b). In the read_state, data is read from the input FIFO to the Keccak core unit. When the Keccak core unit is full and cannot receive new input data, a 1-bit full signal is sent. The control state then transfers to the hold state, where a hold counter is used to count until the end of the hold state.

The Keccak core unit is adjusted and improved based on the design in Reference [32]. The newly designed Keccak core contains additional registers to hold the state in the permutation block. The 1-bit hold signal is to control the hold state, so the permutation process can be paused to wait for the end of the sampling process. The input padder accepts 64 bits of input data every cycle and gets 1,344 bits with padding after 17 or 21 cycles. The valid output bits of the input padder are 1,088 or 1,344, depending on the 2-bit shake_mode signal. The final output is obtained through repeated permutation, and all the processes cost 48 cycles. The valid final output bits are 1,088 or 1,344, depending on the 2-bit shake_mode signal. The intermediate 1,600-bit data XOR with 1,344-bit from input padder until all the input bits are absorbed. The 1-bit last signal indicates the last input, and the 3-bit byte signal is the valid input bytes. The final 1,344-bit can go back into a new round of permutation with 48 cycles until no more output bits are required. The 1-bit squeeze signal controls the output bits' continued generation.

The sampling unit has four stages, and each stage consumes one hardware cycle. An FSM is designed to indicate the computing state of the Keccak core unit in Figure 7(c). If the Keccak core unit produces a valid output, then the state will go to the output_state. In the output_state, when the 1-bit output_ready signal is high, the 1,344-bit output from the Keccak core unit is saved, and the sampling process starts. In stage 1, the number selection block selects the bits of the sampled number, and address counters are used to generate the number address. The sampled numbers are selected according to the sampling type. Two 4-bit sampled numbers in rej_eta sampling are generated from the selected 8-bit number, and two 23-bit numbers in rej_uniform sampling are generated from the selected 48-bit number. However, for the SHAKE256 output, the 64-bit number is sent to the output FIFO directly without sampling. In stage 2, two combined samplers are used for sampling. One 23-bit temporary number is the previously saved sampled number, while the other two 23-bit numbers are the current sampled number. In stage 3, two 23-bit numbers will be selected to output if these numbers meet the requirements. In stage 4, the post transform computation is performed for the two selected numbers in rej_eta sampling. Since the two samplers accept two 4-bit numbers in one cycle during the rej_eta sampling process, they need at least 168 cycles to complete the sampling, which is far more than 48 cycles in permutation. Therefore, in rej_eta sampling, we extend the output cycles to wait for the completion of the sampling process, since, if we enlarge the bit width of the sampler unit, then the used logic resources would be increased significantly. As long as the size of the short vector that needs to be sampled is relatively small, the extension of the clock cycle is a better tradeoff.

4 SOFTWARE/HARDWARE CO-DESIGN ANALYSIS

4.1 Hardware Design and Test

The hardware is designed using Verilog HDL. The designed hardware is synthesized and implemented in Xilinx Vivado 2020.2. The selected device is Zynq-7000 XC7Z020-1. Our design has four main modules: NTT, PWM, PWA, and SHAKE. The four hardware modules are integrated and connected to PS for function acceleration. Each module can be parameterized to accelerate different kinds of functions. Configurability has the following costs in hardware: The first is the need to combine different computing units; the second is to provide an external configuration interface; the third is that it takes extra time to configure. Our design uses the lightweight AXI-Lite bus for configuration. It has a simple structure and consumes very few cycles by configuring 32-bit registers, so the cost of configuration is low. To analyze the performance of function acceleration, each of these modules is tested individually. Note that during the individual module test, the 64-bit width input and output FIFOs are also included and configured as read and write interfaces. The implementation results in terms of hardware resources are shown in Table 4, while the cycle counts for different parameter settings are shown in Table 5.

4.1.1 *Hybrid NTT/INTT Module.*

The Hybrid NTT/INTT module performs both the NTT and INTT having the same polynomial length n and modulus q in Dilithium. The configurability of this module is by adding multiplexers and some arithmetic units based on Algorithm 4. The selection of NTT and INTT functions only needs the 1-bit ntt_sel signal. The module contains only one butterfly unit, which consumes two DSPs. The cycle counts of length n NTT/INTT mainly includes $\frac{n}{2} \cdot 2$ cycles for FIFOs reading and writing, $\frac{n}{2} \cdot \log_2 n$ cycles for NTT calculation, and $15 \cdot \log_2 n$ cycles

Table 4. Hardware Resource of Individual Modules and System Integration

HW Module	LUT	Slice	FF	DSP	BRAM	Fmax (MHz)
NTT/INTT	799	328	971	2	4.5	172
PWM	561	257	796	4	3	178
PWA	527	209	645	0	4	238
SHAKE	8,472	2,411	5,035	0	2	169
HW_ACC_IP	9,365	2,826	6,811	4	5	161
PL_HW_system	13,128 (24%)	4,379 (32%)	11,556 (10%)	4 (1.8%)	14 (10%)	150

Table 5. Hardware Cycles of Each Function under Different Parameter Settings

HW Module	Function	HW cycles
NTT/INTT	ntt ($n = 256$)	1,405
	intt ($n = 256$)	1,405
PWM	point_wise_mul ($k = 1$)	269
	point_wise_mul ($k = 6$)	911
PWA	point_wise_add ($l = 2$)	265
	point_wise_sub ($l = 2$)	265
	point_wise_add ($l = 5$)	665
SHAKE	$H_{256}(32, 96)$	81
	$H_{256}(1{,}952, 48)$	761
	H_{128}+rej_uniform ($n = 256$)	284
	H_{128}+rej_eta_4 ($n = 256$)	302
	H_{128}+rej_eta_2 ($n = 256$)	214

for pipeline delay in different NTT stages. The hybrid structure uses the same cycles for both the NTT and INTT computation, which is 1,405 cycles in Dilithium of $n = 256$ and $q = 8{,}380{,}417$. The critical path in this module lies in the modular reduction unit.

4.1.2 *PWM Module.* The PWM module realizes the point-wise multiplication of two polynomials. The module could also multiply a variable-length polynomial vector. The configurability is achieved by setting the length counter through the 4-bit pwm_vector_len signal. Two modular multiplication units in the PWM module are used to match the transmission speed of input and output FIFOs. The PWM module needs $\frac{n}{2}$ cycles to read the first polynomial. We bury the reading time of the later polynomials into the pipeline computation. There are $\frac{n}{2} \cdot k$ cycles for point-wise multiplication and eight cycles for modular multiplication in the pipeline. When the polynomial vector length k under test is set to six (i.e., the length of NIST security level 3), the cycle cost is 911.

4.1.3 *PWA Module.* The PWA module computes point-wise addition and subtraction of two polynomials. The negation of the numbers for subtraction is hidden in the pipeline. The selection of addition or subtraction function is to use the 1-bit pwa_add_sub signal to depend whether the negative sign is used in the pipeline computation. The PWA module can also be configured to compute the point-wise addition of polynomial vectors. The polynomial vector length is configured by setting the length counter through the 4-bit pwa_vector_len signal. The modular addition unit number is set to two to match the data transmission speed of FIFOs. The computing time mainly includes $\frac{n}{2}$ cycles for data reading of the first polynomials and $\frac{n}{2} \cdot (l - 1)$ cycles of point-wise addition. The vector length is parameter configurable. When setting the tested length $l = 5$ (i.e., the length of NIST security level 3), the cycle cost is 665.

4.1.4 SHAKE Module. The SHAKE module completes the SHA-3 related operations and generates the outputs of the SHAKE256 and the sampled results of the SHAKE128. This module consumes the highest portion of hardware resources, because a relatively high-speed Keccak core would not become the performance bottleneck of the whole system. The configurability of the SHAKE module is achieved by length-configurable input and output counters, the multifunctional Keccak Core, and versatile samplers. The signals defined in Reg2 and Reg3 in Table 3 are all used to configure the parameters in the SHAE module. The first tested SHAKE256 function works as a PRNG, which requires 32-byte inputs and obtains 96-byte outputs. The second tested SHAKE256 function works as a **collision-resistant hash (CRH)** function, which requires 1,952-bytes inputs and obtains 48 bytes outputs. For the other three SHAKE128-related functions, the inputs are 34 bytes, and the outputs are polynomials with a length of 256. All three functions complete the sampling process in the interval between two rounds of Keccak output (each round consumes 48 cycles for permutation). In rej_uniform sampling, at least five rounds of Keccak permutations are required, since two samplers receive 48 bits each cycle, and the sampling acceptance rate is 99%. In rej_eta sampling, two samplers require 8 bits each cycle. For each round (1,344-bit output), 168 cycles are consumed for sampling, which is more than 48 cycles. Therefore, the hold signal is pulled high to extend two round interval cycles from 48 to 168 to wait for the end of the sampling process.

4.1.5 HW System Integration. The HW_ACC_IP integrates the four modules, while the PL_HW system integrates all hardware modules on the PL, including the HW_ACC_IP, AXI-DMA, AXI interconnection, system clock, and the concat module. The maximum working frequency of the PL_HW_system reaches 150 MHz, which is lower than the individual modules. This is because the logic congestion during place and route introduces longer wiring paths. Note that the integrated HW_ACC_IP uses approximately 6.7% less LUT than the sum of the individual modules, because the hardware resource reuse technique is applied during the system integration. More specifically, the modular reduction units are shared between the hybrid NTT/INTT module and the PWM module; thus, four DSPs instead of six are used in the HW_ACC_IP. The BRAMs used by the hybrid NTT/INTT, PWM, and PWA modules are also shared; thus, only five BRAMs are deployed in the HW_ACC_IP.

4.2 Firmware Design and Test

The hardware system in PL is connected with an on-chip ARM Cortex-A9 processor and a 512 MB DDR memory. The processor is working under 666 MHz. After interconnection, the hardware bitstream is generated and exported to Vitis 2020.2. The Vitis is used for software design and system verification. The designed software is written in C/C++. The software includes the original Dilithium reference implementation, the firmware program to initiate the hardware accelerator, and the testing program to evaluate the design performance. The Dilithium reference implementation provides well-optimized algorithms for software execution, so we choose the reference implementation program as the software benchmark and use some of the functions directly in our design.

The firmware program is to provide the control, monitoring, and data manipulation of the hardware accelerator. It manages the data read and write process by sending the start address and length of the data to the DMA controller. It controls the start and parameter configuration of each hardware module by configuring the AIX DMA and registers Reg0-Reg3 in Figure 2. The firmware program also flushes the data to maintain the data consistency between the hardware accelerator and host processor when the cache is ON. We configure different parameters through the

Table 6. Function Acceleration Results with Different Parameter Configurations

	Processor cache turn on			Processor cache turn off		
Function	SW time(μs)	SW/HW time(μs)	Speedup	SW time(μs)	SW/HW time(μs)	Speedup
ntt (n = 256)	177.6	15.6	11.3	2,239.3	14.2	157.0
intt (n = 256)	227.4	15.6	14.5	3,113.0	14.2	218.4
point_wise_mul (k = 1)	50.7	12.7	3.9	731.4	11.2	65.0
point_wise_mul (k = 6)	305.9	36.7	8.3	4,381.1	16.4	266.3
point_wise_add (l = 2)	24.8	12.6	2.0	457.9	11.2	40.8
point_wise_sub (l = 2)	24.7	12.5	2.0	457.8	11.2	40.6
point_wise_add (l = 5)	52.4	16.6	3.1	994.3	11.6	85.2
$H_{256}(32, 96)$	63.3	4.0	15.5	1,007.2	6.3	159.6
$H_{256}(1{,}952, 48)$	954.0	9.8	96.4	14,908.5	9.5	1,553.2
H_{128}+rej_uniform (n = 256)	341.6	11.3	30.0	5,499.2	11.7	467.8
H_{128}+rej_eta_4 (n = 256)	145.5	11.6	12.4	2,333.1	11.9	194.8
H_{128}+rej_eta_2 (n = 256)	81.5	11.0	7.4	1,293.3	11.1	115.6
matrix_mul ($k = 6, l = 5$) (Before SW adjustment)	1,194.4	298.8	4.0	17,910.0	281.4	63.6
matrix_mul ($k = 6, l = 5$) (After SW adjustment)		229.2	5.2		212.6	84.2

firmware program to test the acceleration results of each function. The test results are as shown in Table 6. Due to the time difference of the software execution, all the time indices are the average of 1,000 measurements. To evaluate the speed performance of each function, we compare the computing time of the pure software and software/hardware co-design for each function. The tested software/hardware co-design is to use the designed firmware program to call the hardware accelerator. In terms of pure software implementation, the ARM Cortex-A9 processor is used by turning the processor's cache ON and OFF. This is because, in the **Internet of Things (IoT)** application scenarios, the energy-efficient processors might not have cache support. To demonstrate the use of accelerator in different embedded devices, the firmware program is tested with the cache turned ON and OFF. The speedup is the ratio of the SW time to SW/HW time, which indicates the software/hardware acceleration improvement over the pure software.

As shown in Table 6, the pure software with cache turn ON has around 12–18× speedup compared to the cache turn OFF time. However, the performance improvement of cache is insignificant when compared to the SW/HW time. This is because the cache could significantly accelerate the arithmetic computation in pure software design. While the firmware program mainly completes interface calls and is rarely affected by the cache. In addition, when the cache is ON, data flush functions are required, so this factor increases the SW/HW time. For example, due to the long data flush time, the tested function point_wise_mul (k = 6) consumes significantly more time when the cache is ON. However, DMA preparation time is shorter if the cache is ON, decreasing the total execution time. This helps to explain why the time of the $H_{256}(32, 96)$ function is shorter when the cache is on. The proposed software/hardware system has 2–96× speedup compared to the pure software implementation. The SW/HW acceleration of point_wise_mul and point_wise_add function increase with the parameter k and l, respectively, because the vectorized method is applied to reduce the data transmission amounts. The tested point_wise_mul function is directly used for the polynomial vector multiplication acceleration in Algorithms 2,3. Thanks to the high-performance architecture design of the SHAKE module, the $H_{256}(1{,}952, 48)$ function achieves a speedup of 96 when compared with the pure software time. This result demonstrates that the high-speed architecture design for the time-consuming functions gives a good tradeoff.

4.3 Software Adjustment and Operational Analysis

The polynomial matrix-vector multiplication is the main algebraic operation in the scheme. According to Figure 4, the polynomial matrix-vector multiplication first performs multiplication operations and then performs addition operations. We design multiplication and addition units to accelerate this computation according to method 2 and method b, respectively. However, the original software algorithm is designed according to method 1 and method a, so directly applying our designed hardware for acceleration is not efficient enough. As shown in Algorithm 1, matrix A is obtained with SHAKE128 and the sampling function (i.e., Step 4) and then performs the polynomial matrix-vector multiplication (i.e., Step 6). The sampling function of step 4 is to sample by row, but in method 2 of Figure 4, we need to transfer the data of matrix A by column. The mismatched arrangement of rows and columns between sampling and multiplication of matrix A will lead to inefficient DMA transfers. Because DMA is most efficient when data is continuously transmitted, and the inconsistency of rows and columns requires data to be transmitted in segments, thereby reducing transmission efficiency. Here, we adjust the sampling order of matrix A, from sampling by row to sampling by column. Since different rows and columns are sampled independently, after we modify the sampling order, we can ensure that the obtained matrix A is consistent with the original one without affecting the subsequent calculation results. In addition, after adjustment, no additional memory is needed to store matrix A, since the row and column ordering of sampling and computation is consistent. Otherwise, we need to create an additional array to store matrix A after the multiplication calculation, increasing memory usage. The matrix_mul ($k = 6, l = 5$) is calculated by first using point_wise_mul ($k = 6$) five times to obtain an intermediate matrix and then using point_wise_add ($l = 5$) six times to obtain the final output, as shown in the two steps of Figure 4. We compare the computational efficiency before and after the adjustment in Table 6. We can calculate that after adjustment, the efficiency is improved by $(1-\frac{229.2}{298.8}) = 23.3\%$ with cache ON and $(1-\frac{212.6}{281.4}) = 24.4\%$ with cache OFF.

The SW/HW co-design execution flow of each accelerated function in Table 6 is summarized as shown in the Figure 8. Figure 8(a) shows the common execution order to call the hardware accelerator. This process includes the register configuration of the design parameters and the start signal, cache flush to keep data consistency between cache and DDR, DMA read to send the read data address and length to the DMA module, HW compute for computation, and DMA write to transfer the results back to DDR memory. The configurability of the whole system is achieved by the register configuration process. For the NTT and SHAKE computation, as shown in Figure 8(b), hardware accelerators are working in pipelined with DMA read and write; thus, parts of the DMA read and write time are buried into the hardware computation. For the point-wise multiplication in polynomial vector multiplication and polynomial matrix-vector multiplication, as shown in Figure 8(c), DMA read, hardware computation, and DMA write are deeply pipelined; thus, the data transmission overhead is reduced significantly. In terms of the point-wise addition, as shown in Figure 8(d), the addresses of the vectors may be discontinuous; thus, multiple read data transfers are initiated. However, with the help of FIFO to buffer the data, the data transfer time can still be deeply pipelined with hardware calculations. The whole system can work efficiently by using the above three types of execution flow.

4.4 Overall Software/Hardware Co-design Speedup

After the system integration, we evaluated and compared the Dilithium signature algorithms on both pure software and hardware-software co-design. The transmission interface is configured

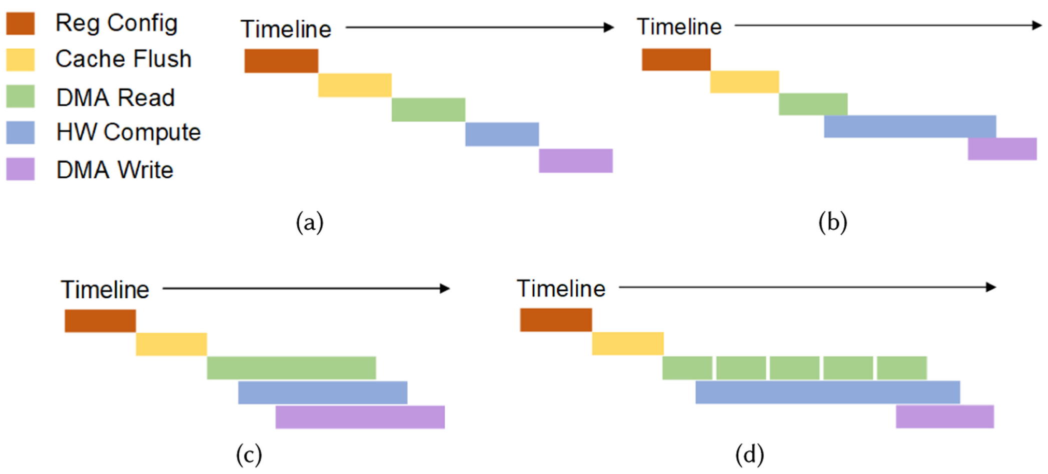


Fig. 8. Execution flow of hardware accelerator. (a) Common interface calling flow. (b) Execution flow with pipelined hardware design in NTT and SHAKE. (c) Execution flow with deeply pipelined hardware design in vectorized multiplication. (d) Execution flow with deeply pipelined hardware design in vectorized addition.

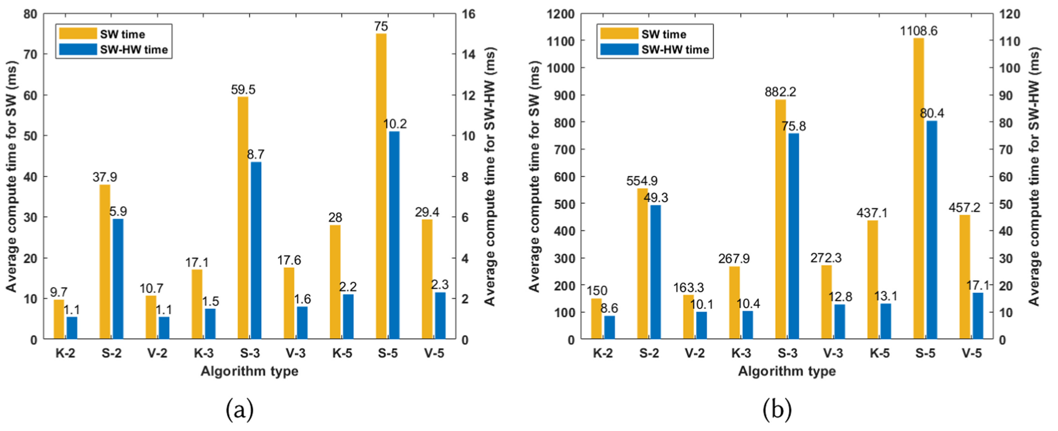


Fig. 9. Dilithium average compute time. (a)compute with processor cache ON. (b)compute with processor cache OFF.

according to the parameters of Dilithium. Moreover, the hardware accelerator is designed to be fully parameter configurable; there is no need to modify the hardware design and transmission interface to adapt to different security levels. The Dilithium algorithms are tested 1,000 times, and the average running time is recorded in Figure 9. The speedup of the software-hardware co-design to pure software is calculated accordingly and illustrated in Figure 10. In Figures 9 and 10, *K* refers to Key generation, *S* refers to Sign, *V* refers to Verify, and 2, 3, and 5 are the corresponding NIST security levels.

As shown in Figure 9, for the same security level, the Key generation and Verify algorithms take similar computing time, while the Sign algorithm consumes 3–5 more time. This is because, during the Sign process, the signature rejection would introduce a re-computation of Sign, thus increasing the computing time. For different security levels, the computing time increases with the expansion of the corresponding parameters.

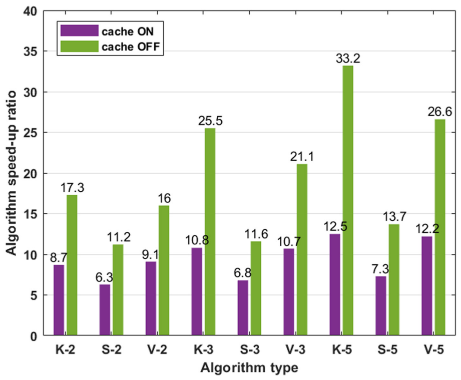


Fig. 10. Dilithium speedup for hardware acceleration.

Table 7. Comparison of NTT Hardware Accelerator

Design	Modulus	Device	cycles	Freq.(MHz)	LUT/Slice/DSP	LUT/Slice/DSP $\times 10^3$ cycles
[46]	8,380,417	Xilinx Artix-7	1,170	216	2,044/N/A/16	2,391/N/A/18.7
[23]	8,380,417	Xilinx Virtex-7	533	N/A	444/N/A/17	241/N/A/9.06
[16]	1,049,089	Xilinx Spatan-6	220	235	N/A/14K/128	N/A/3,080/28.1
[14]	N/A	Xilinx Spatan-6	4,066	233	533/214/1	2,167/870/4.06
This work	8,380,417	Xilinx Artix-7	1,405	172	799/328/2	1,122/460/2.8

Considering the speedup in Figure 10, the Key generation algorithm has the highest acceleration, while the Sign algorithm has the lowest index. This is because the Sign algorithm needs to unpack the generated key and pack the generated signature. These operations have no parallel property; thus, their computation would be serial in the hardware. To alleviate the usage of hardware resources for other operations, the pack and unpack operations are calculated in the software. In summary, when the cache is ON, our hardware-software co-design system could accelerate the Dilithium algorithms by 6.3–12.5×. When the cache is OFF, it could accelerate the algorithms by 11.2–33.2×.

5 COMPARISON WITH RELATED WORKS

5.1 Comparison of Polynomial Operations

Polynomial operations are the most important operations in lattice-based cryptography. Many designs explore different NTT architectures to optimize polynomial operations. The designs in References [10, 14, 16] are specially optimized for the NTT operation, and the designs in References [22, 42–44, 46] are optimized for the NTT design in post-quantum cryptography schemes. Table 7 compares the performance and resource consumption of the NTT hardware accelerator in different designs. The performance is to measure the consumed hardware cycles. The resource consumption compares the used LUT, Slice, and DSPs. The **area-time product (ATP)** is calculated for comparison by multiplying the consumed hardware cycles by the number of LUT, Slice, and DSPs. For a fair comparison, the polynomial length n is 256 in all compared designs.

Zhou et al. [46] designed a hardware architecture for polynomial vector operations, including NTT, INTT, point-wise multiplication, and point-wise addition. In Reference [46], the twiddle factor generation is on the fly and works simultaneously with butterfly operations. Different computation flows were designed for CT-NTT and GS-INTT. The computation of NTT and INTT required an additional stage point-wise multiplication. In contrast, we design a hybrid architecture for both

NTT and INTT computation. We explore the symmetry of the twiddle factor in NTT and INTT and store the precomputed twiddle factor in ROM. Meanwhile, we achieve a more compact design in hybrid architecture by combining control logic and resuing computing resources. Furthermore, we do not need an extra point-wise multiplication stage, because we have incorporated the additional required point-wise multiplication with the precomputed twiddle factor and the butterfly unit computation. Our design consumes longer cycles, because we additionally include the data read and write process between the FIFO and RAM. Our NTT module does not include the point-wise multiplication and point-wise addition, because we provide dedicated acceleration in other modules by fully exploiting the computational properties of MLWE computation. In comparison, our design is more compact and achieves a better ATP.

Land et al. [23] designed a high-speed NTT module, which contained two independent BFUs for parallel computation, and used DSP for all arithmetic operations to achieve low area, high-frequency design. Their design consumed fewer LUTs and hardware cycles, because it did not need to connect to the processor, thus reducing control logic and read and write cycles. Additionally, the design contains two butterfly units for higher parallelism, while we only have one butterfly unit. In comparison, our design consumes fewer DSPs and is more suitable for resource-constrained embedded devices to save DSP resources. Feng et al. [16] proposed a hardware architecture for the multi-lane NTT algorithm, where d ($d = 16$) lanes of butterfly units worked in parallel to complete the NTT calculation. Their design is highly parallel, while our module contains only one computing unit. Compared with Reference [16], our design achieves better ATP in both slice and DSP usage. Du et al. [14] designed an efficient polynomial multiplier to multiply two polynomials. The designed architecture transformed two polynomials into the NTT domain, completed point-wise multiplication, and then transformed back to the normal domain. In our accelerated algorithm, most data is sampled in the NTT domain. It only requires a small amount of NTT calculations while a larger amount of INTT calculations. Therefore, our modules perform NTT and INTT calculations, individually. To sum up, our polynomial operation accelerator achieves a good balance between performance and resource usage, which is suitable for the software/hardware co-design acceleration.

5.2 Comparison of Complete Signature Scheme

Our SW/HW co-design put the software design in an embedded processor and deployed the most time-consuming tasks in an FPGA for acceleration. Many works used similar design principles to speed up the Dilithium algorithm and other signature algorithms. Table 8 makes a detailed comparison of our design with other related SW/HW co-design. However, some works used different design principles, such as deploying the complete algorithm in an FPGA. For complete comparisons, we list the results of pure hardware design using FPGA. Table 9 compares the results with the related pure hardware designs. When comparing the performance of the sign (S), we use the best-case scenario, where the signature is successfully generated after only one loop without rejection in Algorithm 2. According to the reference performance of Skylake CPU in Reference [3] and our analysis results in ARM Cortex-A9, the average cycle of Sign (S) is 4× of the cycle Key Gen (K). While in the best case, the cycle of Sign (S) is 1.5× of the cycle of Key Gen (K). Since most designs provided the best case results, we used the best case in Sign (S) for fair comparisons.

Some of the compared Dilithium designs are based on NIST PQC round-2 parameters. There are some differences between the round-2 and round-3 designs. In round-3 design, the used public key matrix size is changed from 4×3 and 5×4, to 4×4 in security level 2. The non-zero coefficients output in the SampleInBall function changed from 60 to 39 and 49 for security level 2 and 3. The dropped bits of the public key are reduced from 14 to 13. The sampling range of masking polynomial $\mathbf{y}$ is changed to power-of-2 for simplicity. According to the Dilithium-Algorithm

Table 8. Comparison with Related SW/HW Co-design

Scheme[a]		Platform	Freq. (MHz)	LUT	Time (ms) (K/S/V)	LUT × Time $\times 10^3$ (K/S/V)	OPT[b] (ms)	Thro.[c] (OP/s)
[46]	Dilithium-2[d]	Xilinx Zynq 7000 XC7Z020 2CLG400I	533(SW) 100(HW)	2,620[I]	N/A/15.6/11.8	N/A/40.8/30.9	27.4[e]	36.4
				6,253[II]	N/A/15.1/11.5	N/A/94.4/71.9	26.6[e]	37.5
				2,620[III]	N/A/12.6/9.94	N/A/33.0/26.0	22.5[e]	44.3
		Altera Cyclone IV	100(SW)	3,908[IV]	N/A/2,030/465	N/A/7,933/1,817	2,495[e]	0.4
			100(HW)	8,568[V]	N/A/391/128	N/A/3,350/1,096	519[e]	1.9
[4]	Dilithium-2[d]	TSMC 40nm LP CMOS ASIC	72(SW)	N/A	2.3/8.8/1.1	N/A	12.2	81.9
	Dilithium-3[d]		72(SW)	N/A	3.1/11.3/3.8	N/A	18.2	54.9
[19]	Dilithium-5	Xilinx Zynq 7000 XC7Z010-1	667(SW) 163(HW)	N/A	0.50/0.60/0.54	N/A	1.64	609
[45]	SM2[f]	Xilinx Zynq 7000	666(SW) 50(HW)	13,821	9.7/9.3/19	134/128/262	20.9	47.8
This work	Dilithium-2	Xilinx Zynq7000 XC7Z020 CLG484-1	666(SW) 150(HW)	13,128	1.1/2.3/1.1	14.4/30.1/14.4	4.5	222
	Dilithium-3				1.5/3.1/1.6	19.6/40.6/21.0	6.2	161
	Dilithium-5				2.2/4.5/2.3	28.8/59.0/30.1	9.0	111

[a]2,3,5 refer to NIST security level. [b]The operation (OP) includes all the procedures of Dilithium (K+S+V). The operation time (OPT) is the time of each operation. [c]The throughput (Thro.) is the number of operations per second (OP/s). [d]NIST PQC round-2 implementation. [e] The operation here only includes S and V, but not K. [f]SM2 is an elliptic curve-based public key standard, which cannot resist attacks by quantum computers. [I]ARM Cortex-A9 + polynominal(HW). [II]ARM Cortex-A9 + polynominal (HW) + Keccak (HW). [III]ARM Cortex-A9 + polynominal(HW) + Keccak(NEON instructions). [IV]Nios II + polynominal(HW). [V]Nios II + polynominal(HW) + Keccak(HW).

specifications [3, 15], compared with the round-2 security level 2 algorithm, the round-3 security level 2 algorithm consumes cycles of 81% (K), 66% (S), 87% (V), while the round-3 security level 3 algorithm consumes cycles of 146% (K), 115% (S), 129% (V) running on the Skylake CPU. At the same security level, the round-3 algorithm performance is better due to simplifying the above operations and optimizing the software code. However, the parts we do hardware acceleration, including NTT, PWM, PWA, and SHAKE operations, have not changed. The changed parts only affect a small part of the software implementation and have no significant impact on the design performance comparison. Our design has achieved acceleration for all security levels. Therefore, the acceleration performance could be fairly compared with other works.

Zhou et al. [46] presented an SW/HW co-design of Dilithium with a tradeoff between speed and resource utilization. The author designed hardware accelerators for polynomial multiplication and the SHA-3 Keccak function. The author implemented the design on Xilinx Zynq 7000 XC7Z020, which is most comparable to our design. The author provided three hardware acceleration methods in this platform: polynomial (I), polynomial + Keccak (II), and polynomial + NEON vector instruction acceleration for Keccak (III). However, these three methods can only provide no more than 2× speedup, while our design achieves 7–12× speedup. Our speedup is higher because we consider the computational characteristic of MLWE to accelerate point multiplication and point-wise addition. Meanwhile, we combine the Keccak function with different sampling functions to speed up SHA-3 related RPNG functions. The author obtained lower speed and lower throughput on the Altera Cyclone IV platform because of the lower performance of the Nios II processor; therefore, the overall system performance decreases in this case. Overall, our design has higher speedup and higher throughput and achieves a better balance between performance and resource consumption.

Banerjee et al. [4] presented a cryptography processor with NTT acceleration architecture and a Keccak-based PRNG core with a discrete distribution sampler architecture. Their design used a low-power RISC-V microprocessor for software computations and custom hardware architecture for time-consuming operations. This work demonstrated several NIST round-2 algorithms, including NewHope, qTESLA, CRYSTALS-Dilithium, and so on. The processor was fabricated in TSMC

Table 9. Comparison with Other Pure HW Design

Scheme		Platform	Freq. (MHz)	LUT/FF/DSP/BRAM	Time (ms) (K/S/V)	OPT (ms)	Thro. (OP/s)
[41]	Dilithium-2[b]	Xilinx Artix-7 HLS implementation	114	86,646/17,674/N/A/N/A, 90,567/21,160/N/A/N/A, 65,274/15,169/N/A/N/A	2.0/14.2/2.5	18.7	53.4
[37]	Dilithium-2[b]	Xilinx Virtex-7 UltraScale+	350, 333, 158	54,183/25,236/182/15, 68,461/86,295/965/145, 61,738/34,963/316/18	0.05/0.06/0.09	0.21	4,758
[6]	Dilithium-2	Xilinx Artix-7	256	53,187/28,318/16/29	0.02/0.12/0.03	0.17	5,882
	Dilithium-3				0.03/0.19/0.04	0.26	3,846
	Dilithium-5				0.05/0.21/0.05	0.31	3,325
[19]	Dilithium-5	Xilinx Artix-7	163	13,975/6,845/35/4	0.38/0.69/0.41	1.48	675
This work	Dilithium-2	Xilinx Zynq7000 XC7Z020 CLG484-1	666 (SW) 150 (HW)	13,128/11,556/4/14[g]	1.1/2.3/1.1	4.5	222
	Dilithium-3				1.5/3.1/1.6	6.2	161
	Dilithium-5				2.2/4.5/2.3	9.0	111

[b]NIST PQC round-2 implementation. [g]The consumed resources include all modules in PL (the designed accelerator, AXI DMA, DDR interface, etc.).

40 nm low-power CMOS process. Our design further accelerates the point-wise operations for MLWE computation. Because it is an ASIC-based design targeted for energy efficiency, the computing system worked under a low working frequency of 75 MHz. Therefore, our proposed system still surpasses their design in terms of throughput.

Gupta et al. [19] presented a pure hardware design of Dilithium on the Xilinx Artix-7 platform. The authors also evaluated the design as a co-processor and fitted it into the Xilinx Zynq 7000 XC7Z010-1 platform. This work made all calculations in hardware, and the ARM Cortex-A9 processor was only used to complete the interface calls without participating in the calculation. However, our design accelerates each function individually, requiring more data transfers in hardware and software. Our design can be configured to run Key Gen, Sign, and Verify algorithms at all security levels. Hence, our design provides higher flexibility.

Zheng et al. [45] present an SW/HW co-design of the SM2 digital signature algorithm on the Xilinx Zynq-7000 SoC platform. The author implemented point and modular operations in hardware and the signature/verification algorithm flow in software to achieve performance balance. Our design uses a similar platform to achieve a comparable speedup. However, since SM2 is an elliptic curve-based signature algorithm, it cannot resist the attack of quantum computers. Our design shows that Dilithium can provide long-term security guarantees and achieve higher throughput by using similar hardware resource consumption in SW/HW co-design.

Table 9 lists the results for some pure hardware designs. Soni et al. [41] mapped the C source code of Dilithium into FPGA by using **the High-Level Synthesis (HLS)**. The authors implemented the Key generation, Sign, and Verify algorithms separately. Due to the low efficiency of HLS, their hardware design used a relatively large amount of logic but still obtained lower throughput compared with ours. Ricci et al. [37] presented the first hardware design of Dilithium on Virtex-7 UltraScale+ FPGAs. Beckwith et al. [6] presented a high-performance implementation of Dilithium combining three major operations at all security levels. Gupta et al. [19] presented a lightweight hardware implementation of Dilithium at security level 5. The design in Reference [19] is lightweight because each module used a lightweight implementation. For example, the Keccak module in their design consumed 4,200 LUTs, while ours consumed about 8,000 LUTs.

The pure hardware designs have a higher degree of parallelism and do not need to transfer data between software and hardware. Hence, pure hardware implementation can achieve higher throughput compared with SW/HW co-design. However, SW/HW co-design has some advantages.

First, SW/HW co-design only focuses on hardware designs of the computationally intensive parts, which can reduce the system development time. Second, SW/HW co-design can reduce hardware resource usage by realizing module reuse, allowing the system to apply more functions and algorithms. Otherwise, deploying a single algorithm may occupy the resources of the entire board. Third, SW/HW co-design has a higher flexibility. The proposed accelerator has already integrated into the processor and it is easy to apply in real application scenarios. The communication interface can be shared with other accelerators. Thus, it is easier to construct a more versatile system. In addition, The deployment of algorithms may be different from the original algorithms when considering different application scenarios. In the SW/HW co-design, the software in the processor can be easily upgraded and flexibly adjust the parameters in hardware, which can help algorithms better integrate into different scenarios. If a more compact design is expected, then the SHA-3 core in the design could be upgraded to be more lightweight, as we now use a relatively high-performance core to maintain the performance.

6 CONCLUSIONS

This article presents a software/hardware co-design of the NIST PQC round-3 digital signature scheme, CRYSTALS-Dilithium. For high speed, our accelerator designed and implemented hardware modules, including a hybrid NTT/INTT, point-wise multiplier and adder, and SHAKE PRNG with tightly coupled samplers. For flexibility, the ARM processor is cooperated with the aforementioned hardware accelerator to compute Dilithium for different security levels. The hardware is fully pipelined and parameterized and thus could perform different calculations according to the configured parameters. We tested the hardware modules, analyzed individual function performances, and verified the hardware acceleration results of the Dilithium algorithms. According to the implementation results, our design consumes a reasonable amount of hardware resources and obtains high acceleration results. Our software/hardware co-design achieves a good balance in speed, resources, and flexibility compared with existing pure software and hardware designs.

A potential future work is integrating the hardware accelerator into the RISC-V processor and targeting low-power IoT applications. It is worth exploring the design space for a more compact design or designs with a higher degree of parallelism. It would also be interesting to analyze side-channel attacks and explore approaches that would make our hardware accelerator resistant to more advanced attacks.

ACKNOWLEDGMENTS

The authors would like to thank the anonymous reviewers for their valuable comments.

REFERENCES

[1] M. Ajtai. 1996. Generating hard instances of lattice problems (extended abstract). In *Proceedings of the 28th Annual ACM Symposium on Theory of Computing (STOC'96)*. Association for Computing Machinery, New York, NY, 99–108. DOI: https://doi.org/10.1145/237814.237838

[2] Aydin Aysu, Bilgiday Yuce, and Patrick Schaumont. 2015. The future of real-time security: Latency-optimized lattice-based digital signatures. *ACM Trans. Embed. Comput. Syst.* 14, 3 (Apr. 2015). DOI: https://doi.org/10.1145/2724714

[3] S. Bai, L. Ducas, E. Kiltz, T. Lepoint, V. Lyubashevsky, P. Schwabe, G. Seiler, and D. Stehle. 2020. CRYSTALS-Dilithium–algorithm specifications and supporting documentation. *NIST Post-quant. Cryptog. Standardiz. Round 3* (2020).

[4] Utsav Banerjee, Tenzin S. Ukyab, and Anantha P. Chandrakasan. 2019. Sapphire: A configurable crypto-processor for post-quantum lattice-based protocols. *IACR Trans. Cryptog. Hardw. Embed. Syste.* 2019, 4 (2019), 17–61. DOI: https://doi.org/10.13154/tches.v2019.i4.17-61

[5] Paul Barrett. 1986. Implementing the Rivest Shamir and Adleman public key encryption algorithm on a standard digital signal processor. In *Proceedings of the Conference on the Theory and Application of Cryptographic Techniques*. Springer, 311–323. DOI: https://doi.org/10.1007/3-540-47721-7_24

[6] Luke Beckwith, Duc Tri Nguyen, and Kris Gaj. 2021. High-performance hardware implementation of CRYSTALS-Dilithium. In *Proceedings of the International Conference on Field-Programmable Technology (ICFPT'21).* 1–10. DOI: https://doi.org/10.1109/ICFPT52863.2021.9609917

[7] Daniel J. Bernstein. 2009. *Introduction to Post-quantum Cryptography.* Springer, Berlin, 1–14. DOI: https://doi.org/10.1007/978-3-540-88702-7_1

[8] Daniel J. Bernstein and Tanja Lange. 2017. Post-quantum cryptography. *Nature* 549, 7671 (2017), 188–194. DOI: https://doi.org/10.1038/nature23461

[9] Denis Butin. 2017. Hash-based signatures: State of play. *IEEE Secur. Priv.* 15, 4 (2017), 37–43. DOI: https://doi.org/10.1109/MSP.2017.3151334

[10] Donald Donglong Chen, Nele Mentens, Frederik Vercauteren, Sujoy Sinha Roy, Ray C. C. Cheung, Derek Pao, and Ingrid Verbauwhede. 2015. High-speed polynomial multiplication architecture for ring-LWE and SHE cryptosystems. *IEEE Trans. Circ. Syst. I: Reg. Papers* 62, 1 (2015), 157–166. DOI: https://doi.org/10.1109/TCSI.2014.2350431

[11] James W. Cooley and John W. Tukey. 1965. An algorithm for the machine calculation of complex Fourier series. *Math. Comput.* 19 (1965), 297–301. DOI: https://doi.org/10.1090/S0025-5718-1965-0178586-1

[12] Viet B. Dang, Farnoud Farahmand, Michal Andrzejczak, and Kris Gaj. 2019. Implementing and benchmarking three lattice-based post-quantum cryptography algorithms using software/hardware codesign. In *Proceedings of the International Conference on Field-Programmable Technology (ICFPT'19).* 206–214. DOI: https://doi.org/10.1109/ICFPT47387.2019.00032

[13] Jintai Ding and Bo-Yin Yang. 2009. *Multivariate Public Key Cryptography.* Springer, 193–241. DOI: https://doi.org/"10.1007/978-3-540-88702-7_6

[14] Chaohui Du and Guoqiang Bai. 2016. Towards efficient polynomial multiplication for lattice-based cryptography. In *Proceedings of the IEEE International Symposium on Circuits and Systems.* 1178–1181. DOI: https://doi.org/10.1109/ISCAS.2016.7527456

[15] L. Ducas, E. Kiltz, T. Lepoint, V. Lyubashevsky, P. Schwabe, G. Seiler, and D. Stehle. 2019. CRYSTALS-Dilithium–Algorithm specifications and supporting documentation. *NIST Post-quant. Cryptog. Standardiz. Round* 2 (2019).

[16] Xiang Feng, Shuguo Li, and Sufen Xu. 2020. RLWE-oriented high-speed polynomial multiplier utilizing multi-lane stockham NTT algorithm. *IEEE Trans. Circ. Syst. II: Express Briefs* 67, 3 (2020), 556–559. DOI: https://doi.org/10.1109/TCSII.2019.2917621

[17] W. M. Gentleman and G. Sande. 1966. Fast fourier transforms: For fun and profit. In *Proceedings of the Fall Joint Computer Conference (AFIPS'66 (Fall)).* Association for Computing Machinery, New York, NY, 563–578. DOI: https://doi.org/10.1145/1464291.1464352

[18] Denisa O. C. Greconici, Matthias J. Kannwischer, and Daan Sprenkels. 2020. Compact Dilithium implementations on Cortex-M3 and Cortex-M4. *IACR Trans. Cryptog. Hardw. Embed. Syste.* 2021, 1 (Dec. 2020), 1–24. DOI: https://doi.org/10.46586/tches.v2021.i1.1-24

[19] Naina Gupta, Arpan Jati, Anupam Chattopadhyay, and Gautam Jha. 2022. Lightweight Hardware Accelerator for Post-quantum Digital Signature CRYSTALS-Dilithium. Cryptology ePrint Archive, Paper 2022/496. Retrieved from https://eprint.iacr.org/2022/496.

[20] David Jao and Luca De Feo. 2011. Towards quantum-resistant cryptosystems from supersingular elliptic curve isogenies. In *Post-quantum Cryptography.* Springer, Berlin, 19–34. DOI: https://doi.org/10.1007/978-3-642-25405-5_2

[21] J. Kokila, N. Ramasubramanian, and S. Indrajeet. 2016. A survey of hardware and software co-design issues for system on chip design. In *Advanced Computing and Communication Technologies.* Springer, Singapore, 41–49. DOI: https://doi.org/10.1007/978-981-10-1023-1_4

[22] Dur-e-Shahwar Kundi, Yuqing Zhang, Chenghua Wang, Ayesha Khalid, Maire O'Neill, and Weiqiang Liu. 2022. Ultra high-speed polynomial multiplications for lattice-based cryptography on FPGAs. *IEEE Trans. Emerg. Topics Comput.* (2022). DOI: https://doi.org/10.1109/TETC.2022.3144101

[23] Georg Land, Pascal Sasdrich, and Tim Güneysu. 2022. A hard crystal—Implementing Dilithium on reconfigurable hardware. In *Smart Card Research and Advanced Applications.* Springer International Publishing, Cham, 210–230. DOI: https://doi.org/10.1007/978-3-030-97348-3_12

[24] Adeline Langlois and Damien Stehlé. 2015. Worst-case to average-case reductions for module lattices. *Des., Codes Cryptog.* 75, 3 (2015), 565–599. DOI: https://doi.org/10.1007/s10623-014-9938-4

[25] Patrick Longa and Michael Naehrig. 2016. Speeding up the number theoretic transform for faster ideal lattice-based cryptography. In *Cryptology and Network Security.* Springer International Publishing, Cham, 124–139. DOI: https://doi.org/10.1007/978-3-319-48965-0_8

[26] Vadim Lyubashevsky. 2009. Fiat-Shamir with Aborts: Applications to lattice and factoring-based signatures. In *Proceedings of the Advances in CryptologyConference.* Springer, Berlin, 598–616. DOI: https://doi.org/10.1007/978-3-642-10366-7_35

[27] Vadim Lyubashevsky, Daniele Micciancio, Chris Peikert, and Alon Rosen. 2008. SWIFFT: A modest proposal for FFT hashing. In *Fast Software Encryption*. Springer, Berlin, 54–72. DOI: https://doi.org/10.1007/978-3-540-71039-4_4

[28] Vadim Lyubashevsky, Chris Peikert, and Oded Regev. 2010. On ideal lattices and learning with errors over rings. In *Proceedings of the Advances in Cryptology Conference*. Springer, Berlin, 1–23. DOI: https://doi.org/10.1007/978-3-642-13190-5_1

[29] Jose Maria Bermudo Mera, Furkan Turan, Angshuman Karmakar, Sujoy Sinha Roy, and Ingrid Verbauwhede. 2020. Compact domain-specific co-processor for accelerating module lattice-based KEM. In *Proceedings of the 57th ACM/IEEE Design Automation Conference (DAC'20)*. 1–6. DOI: https://doi.org/10.1109/DAC18072.2020.9218727

[30] Daniele Micciancio and Oded Regev. 2009. *Lattice-based Cryptography*. Springer, 147–191. DOI: https://doi.org/10.1007/978-3-540-88702-7_5

[31] Peter L. Montgomery. 1985. Modular multiplication without trial division. *Math. Computat.* 44, 170 (1985), 519–521. DOI: https://doi.org/10.1090/S0025-5718-1985-0777282-X

[32] OpenCores. 2018. SHA3 (KECCAK). Retrieved from https://opencores.org/projects/sha3.

[33] Raphael Overbeck and Nicolas Sendrier. 2009. *Code-based Cryptography*. Springer, Berlin, 95–145. DOI: https://doi.org/"10.1007/978-3-540-88702-7_4

[34] Thomas Pöppelmann, Tobias Oder, and Tim Güneysu. 2015. High-performance ideal lattice-based cryptography on 8-bit ATxmega microcontrollers. In *Proceedings of the Progress in Cryptology Conference*. Springer International Publishing, Cham, 346–365. DOI: https://doi.org/10.1007/978-3-319-22174-8_19

[35] I. Reed and Treiu-Kien Truong. 1975. The use of finite fields to compute convolutions. *IEEE Trans. Inf. Theor.* 21, 2 (1975), 208–213. DOI: https://doi.org/10.1109/TIT.1975.1055352

[36] Oded Regev. 2005. On lattices, learning with errors, random linear codes, and cryptography. In *Proceedings of the 37th Annual ACM Symposium on Theory of Computing (STOC'05)*. Association for Computing Machinery, New York, NY, 84–93. DOI: https://doi.org/10.1145/1060590.1060603

[37] Sara Ricci, Lukas Malina, Petr Jedlicka, David Smékal, Jan Hajny, Peter Cibik, Petr Dzurenda, and Patrik Dobias. 2021. Implementing CRYSTALS-Dilithium signature scheme on FPGAs. In *Proceedings of the 16th International Conference on Availability, Reliability and Security (ARES'21)*. Association for Computing Machinery, New York, NY. DOI: https://doi.org/10.1145/3465481.3465756

[38] Sujoy Sinha Roy, Frederik Vercauteren, Nele Mentens, Donald Donglong Chen, and Ingrid Verbauwhede. 2014. Compact ring-LWE cryptoprocessor. In *Proceedings of the Conference on Cryptographic Hardware and Embedded Systems*. Springer, Berlin, 371–391. DOI: https://doi.org/10.1007/978-3-662-44709-3_21

[39] Michael Scott. 2017. A note on the implementation of the number theoretic transform. In *Cryptography and Coding*. Springer International Publishing, Cham, 247–258. DOI: https://doi.org/10.1007/978-3-319-71045-7_13

[40] P. W. Shor. 1994. Algorithms for quantum computation: Discrete logarithms and factoring. In *Proceedings of the 35th Annual Symposium on Foundations of Computer Science*. IEEE, 124–134. DOI: https://doi.org/10.1109/SFCS.1994.365700

[41] Deepraj Soni, Kanad Basu, Mohammed Nabeel, and Ramesh Karri. 2019. A hardware evaluation study of NIST post-quantum cryptographic signature schemes. In *Proceedings of the 2nd PQC Standardization Conference*. NIST. Retrieved from https://csrc.nist.gov/CSRC/media/Events/Second-PQC-Standardization-Conference/documents/accepted-papers/soni-hardware-evaluation.pdf.

[42] Wen Wang, Shanquan Tian, Bernhard Jungk, Nina Bindel, Patrick Longa, and Jakub Szefer. 2020. Parameterized hardware accelerators for lattice-based cryptography and their application to the HW/SW co-design of qTESLA. *IACR Trans. Cryptog. Hardw. Embed. Syste.* 2020, 3 (June 2020), 269–306. DOI: https://doi.org/10.13154/tches.v2020.i3.269-306

[43] Kan Yao, Dur-E-Shahwar Kundi, Chenghua Wang, Maire O'Neill, and Weiqiang Liu. 2021. Towards CRYSTALS-Kyber: A M-LWE cryptoprocessor with area-time tradeoff. In *Proceedings of the IEEE International Symposium on Circuits and Systems*. 1–5. DOI: https://doi.org/10.1109/ISCAS51556.2021.9401253

[44] Neng Zhang, Bohan Yang, Chen Chen, Shouyi Yin, Shaojun Wei, and Leibo Liu. 2020. Highly efficient architecture of NewHope-NIST on FPGA using low-complexity NTT/INTT. *IACR Trans. Cryptog. Hardw. Embed. Syste.* 2020, 2 (Mar. 2020), 49–72. DOI: https://doi.org/10.13154/tches.v2020.i2.49-72

[45] Xin Zheng, Chongyao Xu, Xianghong Hu, Yun Zhang, and Xiaoming Xiong. 2020. The software/hardware co-design and implementation of SM2/3/4 encryption/decryption and digital signature system. *IEEE Trans. Comput.-aid. Des. Integ. Circ. Syst.* 39, 10 (2020), 2055–2066. DOI: https://doi.org/10.1109/TCAD.2019.2939330

[46] Zhen Zhou, Debiao He, Zhe Liu, Min Luo, and Kim-Kwang Raymond Choo. 2021. A software/hardware co-design of CRYSTALS-Dilithium signature scheme. *ACM Trans. Reconfig. Technol. Syst.* 14, 2 (June 2021). DOI: https://doi.org/10.1145/3447812

Received 17 June 2022; revised 7 October 2022; accepted 9 October 2022