IEEE Access
Multidisciplinary ⋮ Rapid Review ⋮ Open Access Journal

Received 4 February 2024, accepted 15 February 2024, date of publication 26 February 2024, date of current version 5 March 2024.

Digital Object Identifier 10.1109/ACCESS.2024.3370470

RESEARCH ARTICLE

Efficient Low-Latency Hardware Architecture for Module-Lattice-Based Digital Signature Standard

QUANG DANG TRUONG[1], (Graduate Student Member, IEEE), PHAP NGOC DUONG[1,2], (Member, IEEE), AND HANHO LEE[1], (Senior Member, IEEE)

[1]Department of Electrical and Computer Engineering, Inha University, Incheon 22212, South Korea
[2]Faculty of Computer Engineering and Electronics, The University of Danang—Vietnam-Korea University of Information and Communication Technology, Da Nang 50000, Vietnam

Corresponding author: Hanho Lee (hhlee@inha.ac.kr)

This work was supported in part by the MSIT (Ministry of Science and ICT), South Korea, through the ITRC (Information Technology Research Center) Support Program, supervised by the Institute for Information and Communications Technology Planning and Evaluation (IITP), under Grant IITP-2021-0-02052; in part by the National Research Foundation of Korea (NRF) Grant funded by the Korea Government through MSIT under Grant 2021R1A2C1011232; and in part by the IITP Grant funded by the Korea Government through MSIT under Grant 20210007790012003.

ABSTRACT The rapid advancement of powerful quantum computers poses a significant security risk to current public-key cryptosystems, which heavily rely on the computational complexity of problems such as discrete logarithms and integer factorization. As a result, CRYSTALS-Dilithium, a lattice-based digital signature scheme with the potential to be an alternative algorithm that can withstand both quantum and classical attacks, has been standardized as ML-DSA after NIST Post-Quantum Cryptography competition. While prior studies have proposed hardware designs to accelerate this cryptosystem, there is room for further optimization in the tradeoff between performance and hardware consumption. This paper addresses these limitations by presenting an efficient low-latency hardware architecture for ML-DSA, leveraging optimized timing schedules for its three main algorithms. The hardware implementation enables runtime switching main operations in ML-DSA with various security levels. We design flexible arithmetic and hash modules tailored for ML-DSA, the most time-consuming submodules and key determinants of the scheme implementation. Combined with efficient operation scheduling to maximize the utilized time of submodules, our design achieves the best latency among FPGA-based implementations, outperforming state-of-the-art works by 1.27∼2.58× in terms of the area-time tradeoff metric. Therefore, the proposed hardware architecture demonstrates its practical applicability for digital signature cryptosystems in post-quantum era.

INDEX TERMS Post-quantum cryptography (PQC), module-lattice-based digital signature standard (ML-DSA), crystals-Dilithium, lattice-based cryptography (LBC), number theoretic transform (NTT).

I. INTRODUCTION

In recent years, the rise of quantum computing has posed a significant threat to traditional public-key cryptographic schemes. Shor's algorithm [1], when executed on a sufficiently powerful quantum computer, has the capability to efficiently solve complex mathematical problems that form the basis of widely used cryptographic algorithms such as RSA and Elliptic curve cryptography. This realization prompted the National Institute of Standards and Technology (NIST) to launch the Post-Quantum Cryptography (PQC) standardization process in 2016, with the aim of developing new public-key standards that are believed to be secure even against adversaries in possession of a large-scale quantum computer. After three rounds of evaluation and review, NIST has selected CRYSTALS-Dilithium for digital signature standardization to the next round of PQC standardization process [2].

The associate editor coordinating the review of this manuscript and approving it for publication was Mario Donato Marino.

NIST has recently released the initial public drafts of the Module-Lattice-Based Digital Signature Standard (ML-DSA) [3], which is based on the Dilithium submission. Dilithium relies on the worst-case hardness of module lattice problems [4]. It has the potential to resist both quantum and classical attacks and offers advantages such as fast arithmetic operations and compact keys, ciphertext and signature sizes. Given the potential of ML-DSA to become a standard and replace other digital signature schemes specified in NIST Federal Information Processing Standards Publication (FIPS) and Special Publications (e.g., FIPS 186-5 [5]), which is ready to be used for obtaining assurances for digital signature applications. There is a growing interest in developing efficient implementations of this scheme in both hardware and software. Most existing works on implementing and evaluating Dilithium have used pure software methods [6], [7] or hardware-software co-design methods [8]. Developing and benchmarking software implementations of cryptographic algorithms like PQC is relatively straightforward. However, implementing them in hardware platforms, particularly on Field-Programmable Gate Arrays (FPGAs), requires significant time and design effort to assess their efficiency in terms of speed, area utilization, power consumption and energy efficiency. By exploring FPGA implementations, researchers can realize various ideas and apply the tradeoff method to optimize the approach for hardware designs to achieve desired performance goals. FPGA implementations are a crucial and efficient tool for gaining valuable insights into the cost and performance characteristics of PQC algorithms when deployed in hardware environments.

Several prior researches have focused on improving the hardware designs for Round 3 Dilithium on FPGA platforms. In these designs, two main aspects were typically considered. The first is developing hardware architectures that could support all security levels as illustrated in [9], [10] or specifying to a particular security level as proposed in [11], [12], [13], [14]. We recognize that adopting a unified architecture with a flexible mode signal to accommodate all security levels is a more effective approach. This strategy can enhance algorithm performance in hardware, providing flexibility and enabling the architecture to address a broader spectrum of applications. The second consideration is the goal of these implementations. Some studies aimed at maximizing performance while others focused on minimizing power or area consumption or achieving a balance in area-time tradeoff. As a result, these implementations were broadly categorized as high-performance [9], [10] or lightweight [12], [15] although the distinction is not always clear-cut as some designs may aim for specific tradeoff directions. For instance, a high-efficiency design [11] might primarily focus on achieving high performance while keeping resource utilization in comparable level.

Lightweight implementations aim to minimize hardware utilization, allowing them to fit on even the smallest FPGA platforms. However, current FPGA platforms support a large amount of hardware resources, as evidenced by the unified architecture for CRYSTALS-Kyber and Dilithium on the UltraScale+ ZCU102 [15]. This cryptoprocessor utilizes only a small portion of the platform's hardware resources, utilizing less than 10% of available resources. When implementing Dilithium on hardware, one of the most important goals is accelerating execution time to improve application performance. Therefore, a high-performance approach is more suitable for Dilithium hardware implementation, which is still affordable with the available hardware resources on current FPGA platforms. Among the high-performance designs that fully support three security levels, the compact and efficient hardware architecture presented in [10] achieves the best results in resource consumption, while the architecture proposed in [9] achieves the lowest latency. The difference between these two architectures lies in the number of arithmetic and hash modules used. The architecture in [10] has one hash and arithmetic modules, that means it is slower but more compact than the work in [9] which uses multiple modules.

Consequently, we realized that designing efficient hash and arithmetic modules is key factor for achieving high-performance architecture. The latency of hardware implementations for Dilithium is mainly dominated by hashing and polynomial arithmetic tasks, which should be continuously executed as much as possible. Besides, the interconnection between individual submodules in the whole architecture should be carefully considered to avoid critical paths, which can reduce the working frequency of the hardware implementation. The balance between on-chip resource utilization and performance must also be considered to maximize the efficiency of combined architecture for all phases. To implement a more compact architecture, [10] does not support packing/unpacking and the final results are stored in BRAMs. However, in public-key cryptosystems, packing is essential for defining the data layout of the keys and signature before transmission. This is necessary as these applications may be implemented on diverse platforms with varying architectures. Additionally, although the proposed on-the-fly calculation for matrix $\mathbf{A}$ in [10] aims at minimizing storage, it leads to a longer critical path as well as restricts the design to operate at higher frequency. Conversely, [9] provided practical support with packing/unpacking modules featuring 64-bit ports. However, their hash modules support multiple instances of SHA-3 and the complexity of iterative NTT structure leads to a longer critical path. Additionally, utilizing two arithmetic modules with only one used in Keygen and Verify algorithms of ML-DSA, might result in inefficient hardware utilization.

This study approaches a unified hardware design for executing three main algorithms in NIST ML-DSA [3]. Our design improves submodules and employs appropriate operation scheduling tailored to ML-DSA hardware implementation. Additionally, this study aims at a high-performance implementation and achieving an optimal balance between execution time and on-chip resource utilization.

In summary, we make the following main contributions:

1) We propose a unified hardware architecture that supports three algorithms of ML-DSA and can be configured for NIST security levels in run-time. The combined hardware design improves the submodules and optimizes on-chip resource utilization. Furthermore, the proposed architecture supports packing/unpacking modules that enables efficient data exchange with other platforms, facilitating accelerated multiple computational tasks in ML-DSA.
2) The hash and arithmetic modules are designed to match the specific needs of ML-DSA. Specifically, we use independent hash modules for SHAKE-128 or SHAKE-256 and a fully pipelined NTT architecture in the arithmetic module, resulting in an improved critical path. Optimizing these two modules, which have the most substantial impact on the overall execution time of the architecture, leads to performance enhancement for the entire cryptosystem.
3) By using flexible submodules for entire architecture, we propose an optimal timing diagram and proper scheduling to achieve area-time balance when executing ML-DSA processes. Comparison results show that our design achieves the lowest latency among FPGA-based implementations and outperforms state-of-the-art studies in terms of area-time product (ATP) metric.

The remainder of this paper is organized as follows. Section II provides a brief background of ML-DSA and number theoretic transform. Section III proposes our design decisions and details our hardware architecture as well as the optimized modules. The implementation results and comparison with the state-of-the-art works are given in Section IV. Finally, Section V summarizes and concludes the paper.

II. PRELIMINARIES

A. MODULE-LATTICE-BASED DIGITAL SIGNATURE STANDARD OVERVIEW

ML-DSA is a digital signature scheme based on CRYSTALS-Dilithium, a member of the Cryptographic Suite for Algebraic Lattices (CRYSTALS) and a digital signature scheme based on the "Fiat-Shamir with Aborts" approach [16]. Different from other lattice-based algorithms recommended after round 3: Kyber and Falcon, Dilithium uses uniform sampling rather than discrete Gaussian distribution for secret randomness generation. This approach greatly simplifies polynomial generation and is easily implemented in constant time. The hard problems underlying the security of ML-DSA schemes are Module Learning with Errors (M-LWE) and Module Shortest Integer Solution (M-SIS) formally proved in [4].

M-LWE problem needed to protect against key-recovery [17] and M-SIS problem for strong unforgeability [18]. The M-LWE problem can be briefly described as follows: Let $\mathbf{A} \in R_q^{k\times l}$ be uniformly chosen $\mathbf{s_1} \in R_q^l$, $\mathbf{s_2} \in R_q^k$. Then, the standard M-LWE problem is the public key ($\mathbf{A}$, $\mathbf{t} = \mathbf{As_1}+\mathbf{s_2}$) is indistinguishable from ($\mathbf{A}$, $\mathbf{t}$) where $\mathbf{t}$ is chosen uniformly at random. The M-SIS problem in ML-DSA is expressed through: from a uniformly chosen matrix $\mathbf{A} \in R_q^{k\times l}$, that there exists $\mathbf{z}$ and $\mathbf{u}$ such that $\mathbf{Az}+\mathbf{u} = 0$, where $\|\mathbf{z}\|_\infty \leq 2(\gamma_1 - \beta)$, $\|\mathbf{u}\|_\infty \leq 4\gamma_2 + 2$ (parameters can be chosen to match NIST security levels outlined in Table 1). In addition to 3 proposed NIST security levels, the security of ML-DSA can be adjusted by changing the parameters inside. The most straightforward way to enhance or reduce security is by modifying the values of (k, l) and then adjusting η, β and ω accordingly. Another approach is to lower the values of γ_1 and/or γ_2, which makes forging signatures more challenging since it relies on the underlying SIS problem. Increasing (k, l) significantly strengthens security, while adjusting γ_i is more suitable for minor security tweaks. One of the most notable updates in ML-DSA from Dilithium version 3.1 is that it contains two versions of the signature generation algorithm: "hedged" and "deterministic" which are selected by the value *rnd*, as seen in Algorithm 2. The 'hedged' version is a new improvement aimed at facilitating countermeasures against side-channel attacks and fault attacks on deterministic signatures [19].

Key generation (Keygen), Signature generation (Sign) and Verification (Verify), three core algorithms of ML-DSA, are shown in Algorithms 1-3 [3].

KeyGen(): Key generation algorithm generates a keypair consisting of a private signing key (sk) and public verification key (pk) used for signature generation and verification. In this algorithm, two uniform seeds: public seed (ρ) and noise seed (ρ') are mapped to generate polynomial matrix $\mathbf{A}$ and vectors $\mathbf{s_1}$, $\mathbf{s_2}$. It then computes $\mathbf{t} = A\mathbf{s_1} + \mathbf{s_2}$ to generate the final keypair. To keep size small, the public key includes the seed (ρ) instead of matrix $\mathbf{A}$ and the upper bit of $\mathbf{t}$. The secret key packs the lower d bits of $\mathbf{t}$, seed ρ and two byte-arrays K, tr.

TABLE 1. Parameters sets of ML-DSA.

Parameters	NIST security level		
	2	3	5
q [modulus]	8380417	8380417	8380417
d [dropped bits from t]	13	13	13
τ [# of $\pm$1's in c]	39	49	60
γ_1 [y coefficient range]	2^{17}	2^{19}	2^{19}
γ_2 [low-order rounding range]	(q-1)/88	(q-1)/32	(q-1)/32
(k, l) [dimension of $\mathbf{A}$]	(4,4)	(6,5)	(8,7)
η [secret key range]	2	4	2
β [$\tau \cdot \eta$]	78	196	120
ω [max # of 1's in hint]	80	55	75
λ [collision strength of $\tilde{c}$]	128	192	256

Algorithm 1 Key Generation KeyGen() [3]

Output: $pk = (\rho, \mathbf{t_1})$, $sk = (\rho, K, tr, \mathbf{s_1}, \mathbf{s_2}, \mathbf{t_0})$

1: $\zeta \leftarrow \{0, 1\}^{256}$
2: $(\rho, \rho', K) \in \{0, 1\}^{256} \times \{0, 1\}^{512} \times \{0, 1\}^{256} \leftarrow \mathrm{H}(\zeta)$
3: $\mathbf{A} \in R_q^{k\times l} := \mathrm{ExpandA}(\rho)$
4: $(\mathbf{s_1}, \mathbf{s_2}) \in S_\eta^l \times S_\eta^k := \mathrm{ExpandS}(\rho')$
5: $\mathbf{t} := \mathbf{As_1} + \mathbf{s_2}$
6: $(\mathbf{t_1}, \mathbf{t_0}) := \mathrm{Power2Round_q}(\mathbf{t}, d)$
7: $tr \in \{0, 1\}^{512} := \mathrm{H}(\rho \parallel \mathbf{t_1})$
8: **return** (pk, sk)

Algorithm 2 Signature Generation Sign(sk, M) [3]

Input: $sk = (\rho, K, tr, \mathbf{s_1}, \mathbf{s_2}, \mathbf{t_0}), M \in \{0,1\}^*$
Output: $\sigma = (\tilde{c}, \mathbf{z}, \mathbf{h})$

$\mathbf{A} \in R_q^{k \times l}$:=ExpandA(ρ)
$\mu \leftarrow$ H($tr \parallel M$), $rnd \leftarrow \{0,1\}^{256}$ *or* $\{0\}^{256}$
$\rho' \leftarrow$ H($K \parallel rnd \parallel \mu$)
$\kappa := 0$, $(\mathbf{z}, \mathbf{h}) := \perp$
while $(\mathbf{z}, \mathbf{h}) := \perp$ **do**
  $\mathbf{y} \in \tilde{S}_{\gamma_1}^l$:= ExpandMask(ρ', κ)
  $\mathbf{w} := \mathbf{Ay}$
  $\mathbf{w_1} :=$ HighBits$_q(\mathbf{w}, 2\gamma_2)$
  $\tilde{c} \in \{0,1\}^{2\lambda} \leftarrow$ H($\mu \parallel \mathbf{w_1}$)
  $(\tilde{c_1}, \tilde{c_2}) \in \{0,1\}^{256} \times \{0,1\}^{2\lambda-256}$
  $c :=$ SampleInBall($\tilde{c_1}$)
  $\mathbf{z} := \mathbf{y} + c\mathbf{s_1}$
  $\mathbf{r_0} :=$ LowBits$_q(\mathbf{w} - c\mathbf{s_2}, 2\gamma_2)$
  if $\|\mathbf{z}\|_\infty \geq \gamma_1 - \beta$ or $\|\mathbf{r_0}\|_\infty \geq \gamma_2 - \beta$ **then**
    $(\mathbf{z}, \mathbf{h}) := \perp$
  else
    $\mathbf{h} :=$ MakeHint$_q(-c\mathbf{t_0}, \mathbf{w} - c\mathbf{s_2} + c\mathbf{t_0}, 2\gamma_2)$
    if $\|c\mathbf{t_0}\|_\infty \geq \gamma_2$ or $\sum \mathbf{h_i} > \omega$ **then**
      $(\mathbf{z}, \mathbf{h}) := \perp$
    end if
  end if
  $\kappa := \kappa + l$
end while
return $\sigma = (\tilde{c}, \mathbf{z}, \mathbf{h})$

Algorithm 3 Verification Verify(pk, M, σ) [3]

Input: $pk = (\rho, \mathbf{t_1}), M \in \{0,1\}^*, \sigma = (\tilde{c}, \mathbf{z}, \mathbf{h})$
Output: Valid or Invalid

$\mathbf{A} \in R_q^{k \times l}$:= ExpandA(ρ)
$\mu \leftarrow$ H(H($\rho \parallel \mathbf{t_1}$) $\parallel M$)
$(\tilde{c_1}, \tilde{c_2}) \in \{0,1\}^{256} \times \{0,1\}^{2\lambda-256} \leftarrow \tilde{c}$
$c :=$ SampleInBall($\tilde{c_1}$)
$\mathbf{w_1'} :=$ UseHint$_q(\mathbf{h}, \mathbf{Az} - c\mathbf{t_1} \cdot 2^d, 2\gamma_2)$
if $\|\mathbf{z}\|_\infty < \gamma_1 - \beta$ & $\sum \mathbf{h_i} \leq \omega$ & $\tilde{c} =$ H($\mu \parallel \mathbf{w_1'}$) **then**
  return Valid
end if
return Invalid

Sign(): This algorithm takes message M and secret key to generate the signature attached with the message before sending to verifier. Signature generation begins by generating masking vector $\mathbf{y}$ and matrix $\mathbf{A}$ and then compute $\mathbf{w} = \mathbf{Ay}$. It is based on generating challenge c by hashing the message with the high order bit of $\mathbf{w}$, signature as $\mathbf{z} = y + c\mathbf{s_1}$ then and also the hints $\mathbf{h}$ for the verifier. The potential signature σ consisting of $(c, \mathbf{z}, \mathbf{h})$ is checked by adapting the condition of the polynomial max norms are within an acceptable predefined range and the maximum number of 1's in hint that the signature can support which are defined by choosing parameter of the security level in Table 1. Otherwise, the signature must be rejected and a new attempt is made.

Verify(): The high-order bit of $\mathbf{w_1}$ is recovered from message, public key and signature. It is used as a part of hash function to generate challenge c which will be compared to the $\tilde{c}$ provided in the signature to confirm the authentication and integrity of the original message.

As a lattice-based cryptosystem, implementing ML-DSA faces challenges in the sample generation and computation phases, which are the two most time-consuming progresses. To achieve high-performance hardware implementation, efficient modules tailored to these phases are necessary. Based on three algorithms, we observed that the Sign algorithm requires more extensive processing compared to the others. Previous works, as documented in citations [10], [11], [12], [13], [14], typically followed a sequential approach, implementing the three algorithms in the order outlined by Algorithms 1-3. In these works, various methods were proposed, with a predominant focus on enhancing the efficiency of arithmetic module. To speed up the execution time of these algorithms, the use of multiple hash and arithmetic modules becomes essential. However, this approach increases hardware utilization, demanding careful consideration of submodule architecture efficiency and its impact on the overall architecture. Therefore, in hashing tasks, we propose the use of two independent hash modules, including a SHAKE-256 module and a double Keccak-core Hash module for SHAKE-128 instance in ML-DSA. These modules enhance the latency and critical path of the overall architecture, addressing concerns raised by a unified Keccak module proposed in previous works. Additionally, our proposed flexible arithmetic module efficiently resolves the challenge posed by the Sign algorithm, which requires more extensive processing compared to the others. By incorporating these modules into an appropriate operation scheduling, as proposed in section III-A, we achieve improvements in latency within comparable hardware resources.

B. NUMBER THEORETIC TRANSFORM

The Number Theoretic Transform (NTT) is a variant of the Fast Fourier Transform (FFT) that operates on a finite field and provides an efficient method for polynomial multiplication. It reduces the complexity of polynomial multiplication from $\mathcal{O}(n^2)$ by using school-book method to $\mathcal{O}(n\log n)$ by using point-wise multiplication method. The NTT-based multiplication can be perform as: $c = \text{INTT}(\text{NTT}(a) \circ \text{NTT}(b))$. Where a, b, c could be polynomial matrix or vector $\in R_q$, $\circ$ denotes point-wise multiplication of polynomial coefficients.

ML-DSA, as a lattice-based cryptosystems, utilizes the NTT for matrix-vector and vector-vector multiplications. The NTT is performed in the ring $R_q = Z_q[x]/(x^n + 1)$, where n is a power of 2 and q is a prime number. In ML-DSA, the modulus q is chosen such that there exists a $2n$-th root of unit ($\phi_{2n} = 1753$). There are two common algorithms for implementing NTT, Cooley-Tukey (CT) decimal-in-time (DIT) algorithm for NTT and Gentleman-Sande (GS) decimal-in-frequency (DIF) algorithm for inverse NTT (INTT). Applying the combination of these algorithms

Algorithm 4 Pipelined Radix-2 NTT Algorithm

Input: $\mathbf{a} = (A_0[0], A_0[1], \ldots, A_0[n\text{-}1]) \in R_q$, Ψ contains twiddle factor constants.

Output: $\mathbf{A} = \mathrm{NTT}(\mathbf{a})$

1: **for**($i = 1; i \leq log_2(n); i{+}{+}$) **do** // unrolling
2: **for** ($j = 0; j < n/2; j{+}{+}$) **do** // pipelining
3: $W \leftarrow \Psi_i[j]$
4: $temp \leftarrow W \times A_{i-1}[j + n/(2^i)] \pmod q$
5: $A_i[j + n/(2^i)] \leftarrow A_{i-1}[j]$ - $temp \pmod q$
6: $A_i[j] \leftarrow A_{i-1}[j] + temp \pmod q$
7: **end for**
8: **return A**

can accelerate computations and eliminate the need of bit-reversal step [20].

When implementing NTT on hardware, two popular variants of the NTT architecture are used: memory-based and pipelined architectures. The memory-based or iterative architecture has the advantage of requiring fewer hardware resources, but it is limited by memory port constraints, which restrict the number of butterfly units (BUs) that can be used to enhance NTT speed. This is evident in works of [9], [12], [15], which are limited to using a maximum of 4 BUs. Additionally, the memory-based architecture requires a complex control module to calculate the NTT layer-by-layer, which increases the critical path. The pipelined architecture, on the other hand, requires a large number of shift registers to store delay units, which depend on the chosen scheme [21]. This consumes more hardware resources, but it results in higher performance. The pipelined NTT architecture is more suitable for speeding up the execution time of ML-DSA, as it is used for the most time-consuming phase of this scheme. However, implementing this architecture in ML-DSA requires at least eight BUs, which leads to the drawback of overusing resources. This drawback will be addressed by our flexible arithmetic module, proposed in Section III. Algorithm 4 performs the pipelined radix-2 NTT computation, where the variable i in the loop represents the layer NTT being executed simultaneously (corresponding to position of BU_i), j is the order of coefficients in polynomials, which are sequentially transformed via the NTT hardware architecture.

III. ARCHITECTURE AND DESIGN DECISIONS

As the design's goal is efficient low-latency implementation for ML-DSA on the FPGA SoC platform, the proposed architecture is supported to perform all ML-DSA's main algorithms at three security levels as defined in its specification. This design supports all processes including packing and sending signature and keypair, which can cooperate with other software or hardware platforms easily. This section will introduce the overall high-level architecture and its optimized modules used to improve latency. Accompany with the efficient hardware implementation, the timing schedules in Figs. 2-4 are proposed to optimize the work

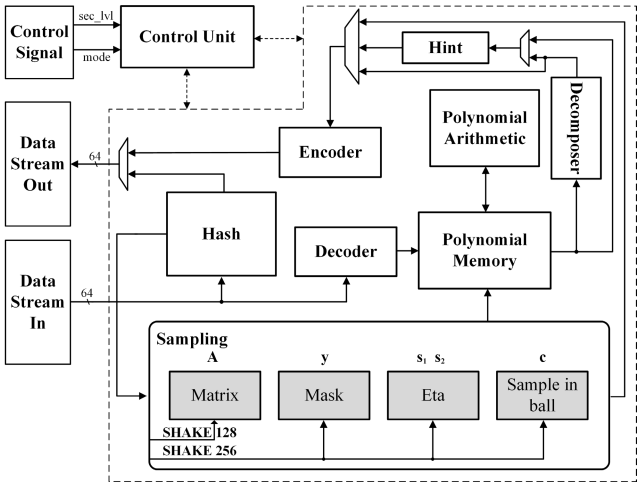

FIGURE 1. High-level configurable architecture of ML-DSA.

of entire architecture and get low-latency aspect which takes advantage of the most important and time-consuming submodules (polynomial arithmetic and hash module specify for ML-DSA).

A. HIGH-LEVEL CONFIGURABLE ARCHITECTURE

1) OVERALL HARDWARE ARCHITECTURE

The high-level architecture of our hardware design is shown in Fig. 1. As a configurable architecture that implements three main algorithms of ML-DSA, this figure depicts all components used. When executing one of those algorithms individually, some submodules can be removed. To facilitate seamless interoperability with other software/hardware platforms, our architecture supports packing/unpacking tasks with 64-bits bus width. The block inside is working in 92-bits bus width to perform four coefficients at the same time. A brief description of these modules and their works is explained below:

Control unit: Functioning as a finite-state machine (FSM), the control unit plays a vital role in overseeing various modules. It manages these operations based on required functionalities and sequences the entire hardware module to execute all the primary operations of ML-DSA in their respective orders. The control unit employs two control signals, namely 'mode' and 'sec_lvl', to configure which algorithm and security level the architecture is set to operate. Depending on these signals, the parameters are adjusted to work in accordance with the values specified in Table 1. The value *rnd* is declared as an initial value in the source code, where the default is a 256-bit string consisting entirely of zeroes. Users have the option to change it to an arbitrary value to enhance security.

Polynomial BRAM: This module contains six groups of three dual-port 36Kb BRAMs used to store intermediate values during the computation. In ML-DSA scheme, the modulus q is 8380417, allowing for coefficients to be represented within 23 bits. The Xilinx FPGA supports dual-port 36Kb BRAM memory on-chip [22]. Instead of storing

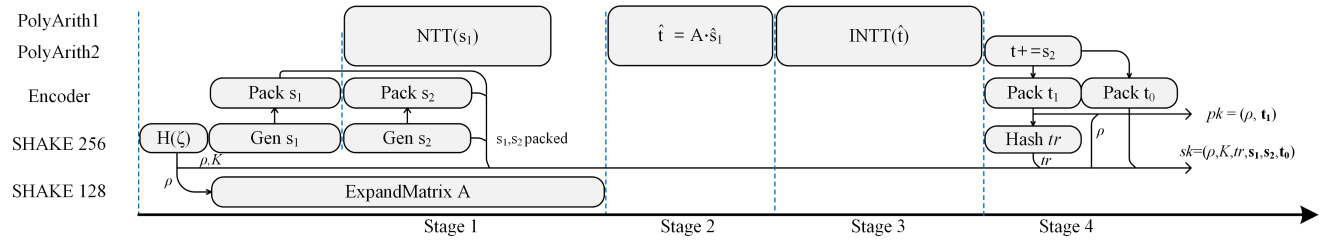

FIGURE 2. Timing diagram of key generation phase in security level 5.

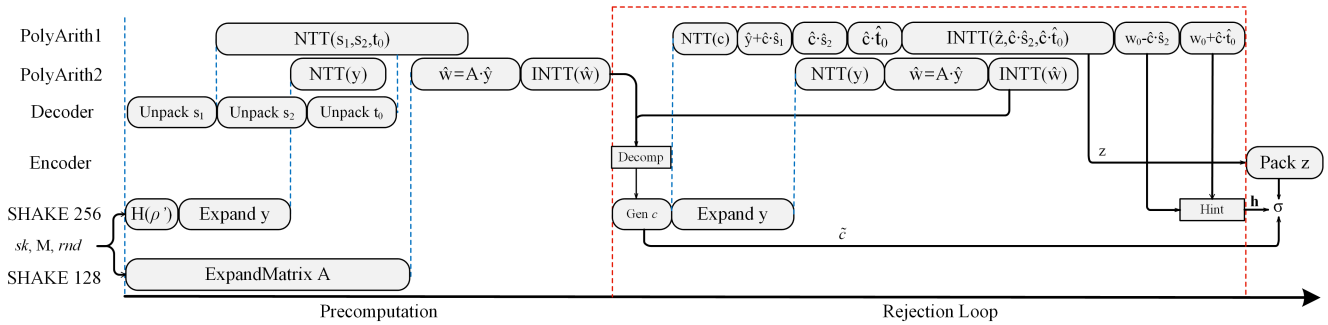

FIGURE 3. Timing diagram of sign phase in security level 5.

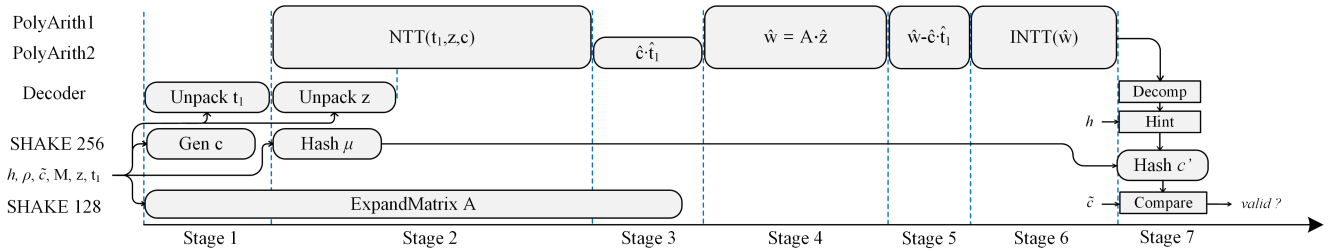

FIGURE 4. Timing diagram of verification phase in security level 5.

each coefficient in a separate BRAM address, this module is composed as a group of three BRAMs, each configured as a 1024 × 36 memory. This configuration allows for four coefficients, with a bandwidth requirement of $4 \times 23 = 92$ bits, to be stored in a single address of the group BRAMs, resulting in a 25% reduction in BRAM utilization compared to the straightforward approach. Totally, six group BRAMs are used to save intermediate values during the computation process.

Polynomial arithmetic module: used for all arithmetic operations in ML-DSA, including NTT, INTT, point-wise multiplications, additions and subtractions between polynomial matrices and vectors.

Hash module: includes SHAKE-128 and SHAKE-256 modules, which function as hash functions or Extendable-Output Functions (XOF) [23]. These modules generate hash values of messages, random seeds and pseudorandom data, which are then used for polynomial sampling in ML-DSA.

Sampling unit: This module consists of four submodules responsible for generating different types of pseudorandom samples used in ML-DSA. ExpandMatrix maps a uniform seed to matrix $\mathbf{A}$, UniformEta for generating the secret vectors $\mathbf{s_1}, \mathbf{s_2}$, ExpandMask to make the masking vector $\mathbf{y}$ and SampeInBall to create challenge c.

Decomposer: This module breaks up a finite field element r in Z_q into their "high-order" bits and "low-order" bits, such that $r = r_1 \cdot 2\gamma + r_0$ where $0 \leq r_0 < 2\gamma$.

Hint: used to describe the work of both MakeHint and UseHint functions. Given an arbitrary element $r \in Z_q$ and a small element $z \in Z_q$, "MakeHint" records a 1-bit hint $\mathbf{h}$ as the "carry" that enables the computation of the higher-order bits of $r + z$ using only r and $\mathbf{h}$. Conversely, "UseHint" utilizes the hint to recover the high-order bits of the sum.

Encoder/decoder: To interface with other hardware or software platforms, which typically have a bandwidth of 2^n, this implementation integrates encoder/decoder units. The encoder is responsible for packing polynomial vectors into byte strings. The different vectors, along with their corresponding bit widths, are packed into fixed 8-byte strings.

The decoder can then recover the original vectors from these byte strings.

Depending on the algorithm being performed, certain processing units can operate in parallel to reduce overall latency, provided there is no data flow conflict between them. From Algorithms 1-3, the polynomial generation and computation phases in the ML-DSA scheme are the most time-consuming. By prioritizing these tasks, the computation time of other tasks can be masked behind these lengthy phases. This design focuses on optimizing these tasks sequencing, enabling continuous execution to enhance overall performance.

2) OPERATION SCHEDULING

Based on algorithms and the proposed architecture, we suggest an optimized timing schedule that maximizes the utilization of submodules while maintaining constant progress on the critical path of the operations. In our architecture, the controller operates as a FSM, ensuring sequential processing of all hardware modules. To enhance clarity, we present the optimized schedule specifically for level 5 of Algorithms 1-3.

Key generation and Verification: The FSM schedule for Key generation and Verification algorithms are described straightforwardly stage by stage in Figs. 2 and 4 corresponding with Algorithms 1 and 3, respectively. The longest path among them is the computation, hashing and sampling, which occupies around 90% total time execution. The packing/unpacking phases are immediately executed in parallel with the generation matrix, vectors and computations. Pack t_0, Pack t_1 perform the function Power2Round, the straightforward bit-wise way to break up $\mathbf{t}$ into 13-bits high-order $\mathbf{t_1}$ and 10-bits low-order $\mathbf{t_0}$, in line 6 of Algorithm 1. The key generation progress is finished after packing their ingredients. In verification, the validity of the signature depends on the result of the comparison described in line 6 of Algorithm 3, which is implemented in stage 7 as depicted in Fig. 4.

Signature generation: This is the most complex main algorithm in ML-DSA. It consists of two phases: precomputation and rejection loop, as shown in Fig. 3. During the precomputation phase, the secret key and message are unpacked and the necessary calculations are performed prior to the while loop in Algorithm 2. This phase is executed only once. The rejection loop is responsible for generating the signature until it satisfies the requirements specified in the algorithm. The average number of loops can be found in [3]. Inspired by the approach proposed by Beckwith et al. [9], the polynomial arithmetic module is divided into two independent parts, functioning as described in mode 2 of this module. The first part is used to create the vector $\mathbf{w}$ and executes the functions from lines 6 to 8, while the remaining functions within the loop are performed in the second module.

B. HASHING AND SAMPLING

ML-DSA scheme utilizes SHAKE-128 and SHAKE-256 as its hashing functions, both belonging to the XOF category within the Secure Hash Algorithm 3 (SHA-3) family. These functions are based on the same Keccak permutation with a state size of 1600 bits [23]. In this context, matrix $\mathbf{A} \in R_q^{k \times l}$ is the output stream of SHAKE-128. The other sample vectors in ML-DSA are generated by SHAKE-256, described in section II, including polynomial secret vectors $\mathbf{s_1}, \mathbf{s_2}$, masking vector $\mathbf{y}$, challenge c and other hashing tasks. Due to the shared Keccak-f[1600] core with a 24-round permutation, the unified SHAKE128/256 approach has been employed in prior works. However, in our work, we propose using two independent Keccak-based Hash modules, customized to suit ML-DSA's unique requirements. This proposed method allows for the simultaneous execution of multiple tasks, such as generating matrix $\mathbf{A}$ and other vectors, resulting in reduced estimated execution times for the scheme. Moreover, implementing dedicated hash modules tailored to specific requirements can reduce the complexity of this module and the entire architecture, potentially leading to an increase in the maximum achievable frequency of the entire system.

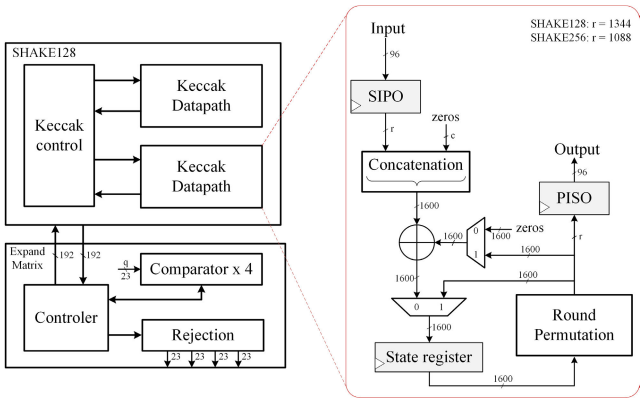

FIGURE 5. Sampling for matrix A.

ML-DSA's security primarily depends on the dimensions (k, l) of the vectors and matrices in the scheme, leading to a substantial demand for pseudorandom data, corresponding to longer execution times in this phase and entire algorithm. Especially, in the hardest security level, where matrix $\mathbf{A}$ necessitates 56 polynomial samples, totaling 344 kbits, without considering the rejection rate for each sampling. To address this challenge, we employ double 96-bit datapath Keccak cores in the SHAKE-128 module, while the SHAKE-256 module employs a single 64-bit core, as illustrated in Fig. 5. This design aligns with the characteristics of ML-DSA, optimizing the execution time of the sampling phase while minimizing resource utilization. Our modules are modified from the high-performance Keccak implementation core developed by the Keccak team [24]. The Keccak control unit manages the internal signals to control module operations and facilitate transitions between the absorb and squeeze phases. Taking the example of sampling matrix $\mathbf{A}$, depicted in Fig. 5, the SHAKE-128 module needs 4 clock cycles (CCs) to absorb 256 bits of seed and 16-bit nonce before transitioning to the squeeze phase, where sampling and hashing processes operate continuously. With the SHAKE-128 rate $r = 1344$ and 96-bits datapath of this core, the output of each squeezing phase necessitates 14 cycles, which

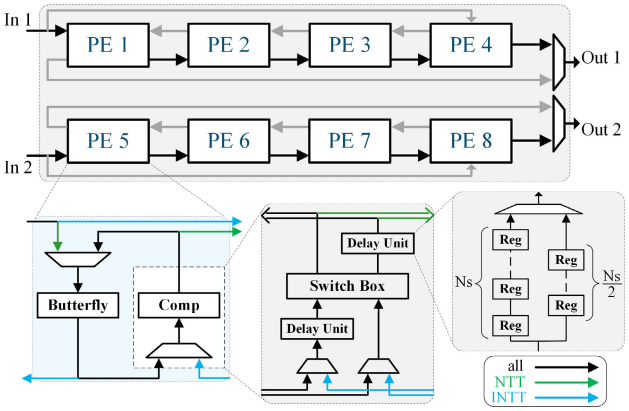

FIGURE 6. Flexible polynomial arithmetic module in mode 1.

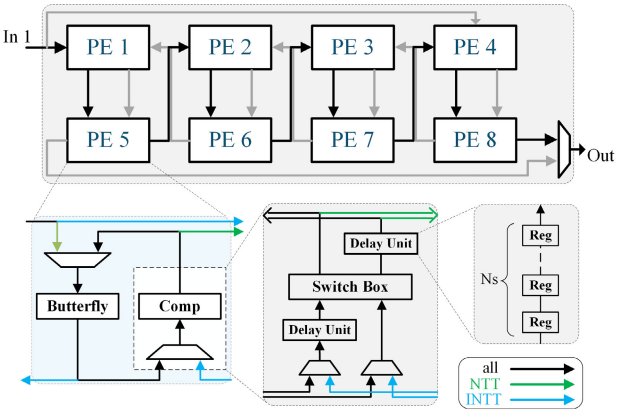

FIGURE 7. Flexible polynomial arithmetic module in mode 2.

can occur simultaneously within 24 cycles required for the Keccak permutation in the subsequent squeezing phase. In summary, considering that the coefficients in matrix **A** fall within Z_q (where q is the modulus), the acceptable rate is $q/2^{23}$, which is close to 1. Consequently, this module requires approximately 144 cycles to generate two polynomials, each with 256 coefficients.

The remaining hashing tasks and sampling vectors in ML-DSA are implemented in the SHAKE-256 module in a similar manner. This module consists of a single 64-bits data-path core controlled by the Keccak control unit, specifically designed for SHAKE-256. It interfaces with the corresponding sampling units to generate the required vectors based on their rejection conditions and received data widths, as outlined in Table 1. The selection of a 64-bit datapath is well-suited for the width of the output samples, making it compatible with integration with other software/hardware platforms.

C. FLEXIBLE POLYNOMIAL ARITHMETIC MODULE

As discussed in Section II, one of the critical considerations is the extensive computational demands in the Sign algorithm, particularly during the rejection loop phase. To improve performance in this phase, we propose an approach inspired by [9] to distribute the calculations across two polynomial arithmetic (PolyArith) modules. However, since our architecture unifies all three main algorithms of ML-DSA, implementing two PolyArith modules when Verify and Keygen algorithms only require one is an unnecessary resource allocation. Hence, our arithmetic module is designed based on the radix-2 MDC FFT pipelined architecture [21] featuring eight Processing Elements (PE), which includes butterfly unit and other sub-units, to handle all polynomial arithmetic operations in ML-DSA. This module offers two modes that can be flexibly adapted to suit the specific characteristics of ML-DSA, both of them are configurable to support NTT/INTT operations with the data flow pass through the black and faint lines, respectively, in Figs. 6, 7.

Fully pipelined mode: In this mode, we use eight PEs corresponding to the eight layers of ML-DSA for N=256, as outlined in Algorithm 4. BU_i take the input from its previous BU consequently, the twiddle factors have been precomputed and saved in corresponding ROM address. The stall delay length from input to output is 153 CCs, after that each polynomial requires 128 cycles for NTT/INTT. The BUs can be reused for all polynomial arithmetic, processing 8 coefficients in parallel. This mode is employed in the Keygen and Verify.

Folding transform mode: In this mode, we divide our module into two independent modules, each equipped with four PEs. By applying the folding transformation method [25], each PE is responsible for calculating two adjacent NTT layers in a time-sliced fashion. This enables us to employ four PEs to implement radix-2 MDC NTT in ML-DSA, similar to the approach in [10]. In this mode, during NTT/INTT, the PEs handle the odd stages in odd cycles and the even stages in even cycles. This mode has a delay of 281 CCs from input to output and requires 256 CCs to transfer a single polynomial during NTT/INTT. It also supports the processing of 4 coefficients per cycle during polynomial arithmetic. This mode is used for Sign operation, one module to compute vector **y** and **w**, the other for remaining computations as shown in Fig. 3.

To optimize memory and efficiently store twiddle factors (TFs), this module uses six registers dedicated to storing precomputed TFs for the two first layers of NTT/INTT. These registers are directly connected to PE1 and PE4. Additionally, it utilizes two 36Kb BRAM configurations in a 72×512 mode to store TFs associated with the remaining six PEs. The addresses in BRAM are carefully arranged in accordance with their order of use. Shift registers are used to implement a FIFO unit for saving intermediate values during NTT/INTT.

1) BUTTERFLY UNIT

The butterfly unit plays a crucial role in the arithmetic module for performing various operations. In the proposed design, a configurable BU architecture is presented, supporting both the CT BU and GS BU methods, as depicted in Fig. 8. The proposed butterfly unit consists of one modular multiplier, one modular adder and one modular subtractor, without requiring any additional modular multiplier, adder,

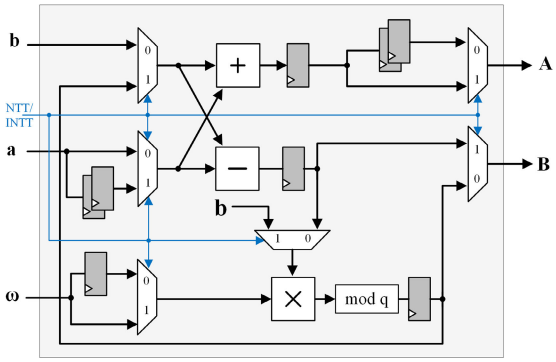

FIGURE 8. Proposed butterfly unit.

or subtractor compared to a dedicated CT or GS butterfly configuration. The proposed BU takes **a**, **b** and $\boldsymbol{\omega}$ as input, the NTT/INTT control signal in the blue line acts as a selection signal for the multiplexers, configuring the BU to operate as either a GS or CT butterfly configuration. the BU architecture enables it to handle point-wise multiplication, addition and subtraction through the corresponding internal unit. **A** and **B** denote the output ports when performing point-wise multiplication or subtraction, the result is taken from port **B**, while port **A** is utilized for addition using the CT butterfly configuration.

In [26] Zhang et al. proposed a technique to eliminate the multiplication of resulting coefficients with $n^{-1}(\text{mod } q)$ after the INTT operation. The output can be seen as (**A**, **B**):= (2^{-1}(**a** + **b**), (**a** - **b**)$\boldsymbol{\omega}$). For an odd prime q, $x/2$ (mod q) can be performed as shown in Eqn. 1. In this work, we adopt this technique by inserting divide 2 operation integrated into addition unit and pre-processed the twiddle factor for the inverse NTT to incorporate the factor 2^{-1}.

$$x/2 \ (\text{mod } q) = (x \gg 1) + x[0].(q+1)/2 \tag{1}$$

2) MODULAR MULTIPLICATION

Modular multiplication, addition and subtraction are the fundamental arithmetic components inside the butterfly unit. The reference implementation on software platforms of ML-DSA [3] uses Montgomery reduction after multiplication and Barrett reduction for addition and subtraction. Both of these can apply to hardware and have been adopted in some related works but not seem to be the best choice. While Montgomery needs to transfer from normal domain to Montgomery domain leading to additional multiplications, Barret reduction does not exploit the specification of modulus q in ML-DSA. In [13], Land proposes the reduction method by recursively exploit the relation $2^{23} \equiv 2^{13} - 1$:

$$\begin{aligned} s[45:0] &\equiv 2^{23}s[45:23] + s[22:0] \\ &\equiv 2^{13}s[45:23] - s[45:23] + s[22:0] \\ &\equiv 2^{23}s[45:33] + 2^{13}s[32:23] - s[45:23] + z \\ &\equiv 2^{13}(s[45:43] + s[42:33] + s[32:23]) \\ &\quad - (s[45:43] + s[45:33] + s[45:23]) + z \end{aligned} \tag{2}$$

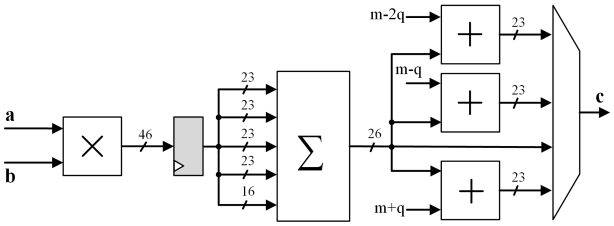

FIGURE 9. Architecture of modular multiplication unit.

Following careful consideration, we decided to incorporate this technique into our modular reduction unit. However, as subtraction is not as hardware-friendly as addition, we employ the additive inverse with $\bar{s}$ representing the negative of s, to transform the equation into:

$$\begin{aligned} s[45:0] &\equiv \bar{s}[45:23] + (2^{13}s[32:23] + \bar{s}[45:33]) \\ &\quad + (2^{13}(s[42:33] + \bar{s}[45:43]) + 2^{13}s[45:43] \\ &\quad + z + 50331648 \\ &\equiv \bar{s}[45:23] + \{s[42:33], 10'b0, \bar{s}[45:43]\} \\ &\quad + \{s[32:23], \bar{s}[45:33]\} + 2^{13}s[45:43] + z + m \end{aligned} \tag{3}$$

Since the result of the reduction at this point falls within the interval $(-q, 3q)$, we can simplify the hardware architecture for our modular multiplication based on this equation, as shown in Fig. 9. In this figure, $m = 50331648$ and $\{\}$ denotes concatenated function.

IV. IMPLEMENTATION RESULTS AND COMPARISON

A. RESOURCES UTILIZATION AND PERFORMANCE RESULTS

Table 2 provides an overview of the resource consumption for the entire design and its submodules. The complete design utilizes 49753 LUTs, 23477 FFs, 27.5 BRAMs and 16 DSPs, working at the maximum frequency of 300 MHz and total power consumption is 5.24W. To achieve the low latency target, we selected the Virtex Ultrascale+ platform and all the resource data was generated by Xilinx Vivado 2022.2. Due to our data-path employing four parallel paths to execute four coefficients per cycle, the resource usage is higher compared to other Lightweight target works. Two hash modules account for approximately 29% and 42% of the LUT usage, making them the most resource-intensive components. The design utilizes a total of 27.5 BRAMs, with 10.5 BRAMs allocated for storing matrix **A** and 15 BRAMs used for intermediate values during computation. The polynomial arithmetic module requires 2 BRAMs to store twiddle factors and 16 DSPs for eight PEs. As a combined architecture, some units are only utilized in specific algorithms and are not called when performing other tasks. For example, the Hint module is not involved in the Keygen. However, the hash modules, ExpandA, PolyArith, encoder/decoder and polynomial RAM can be fully shared among different algorithms, consuming 55% LUTs, 66% FFs and 100% DSPs.

TABLE 2. Resource utilization of submodules in the architecture.

Submodule	Resource Utilization			
	LUT	FF	DSP	RAM
Polynomial RAM	0	0	0	25.5
Hint	10476	3776	0	0
Encoder	2183	464	0	0
Decoder	2683	251	0	0
Decomposer	1565	728	0	0
PolyArith	5819	3527	16	2
SHAKE-128	9278	6041	0	0
SHAKE-256	5126	3941	0	0
Sampling	6942	2868	0	0
Other	5681	1881	0	0
Top ML-DSA Architecture	49753	23477	16	27.5

In terms of latency, we have compiled statistics on the overall execution time, including the data input, unpack and pack signature and keypair processes. From the average statistics of 1000 executions, we obtained the following specific parameters. The Keygen operation requires an average of 3320, 5330 and 8226 CCs for three security levels. The interval between each sample depends on the rejection rate of matrix $\mathbf{A}$ and secret vectors $\mathbf{s_1}$ but remains relatively small due to the high acceptance rate. The Verify operation consumes around 4137, 5846 and 8062 CCs for three NIST levels. In this operation, only valid samples are calculated, because the invalid sample is processed much faster and does not show total time execution of this operation. The interval time also depends on the rejection condition in generating matrix $\mathbf{A}$. The number of cycles required for the Sign algorithm varies widely due to the rejection loop in signature generation. The best-case scenario occurs when the signature is accepted after the first iteration, resulting in a time of 10804, 14859 and 21199 CCs for three security levels. The processing time of input message is determined by the size of the message and operates at a rate of 8 bytes per cycle. Based on the parameters in the ML-DSA specification, an average of 4.25 attempts is required for level 2, 5.1 attempts for level 3 and 3.85 for level 5. The average time to execute one attempt is 5665, 7658 and 10340 CCs, leading to average times of signature generation are 29219, 46274 and 50848 CCs for three security levels.

B. COMPARISON WITH RELATED WORKS

ML-DSA was recently introduced with slight updates to the earlier version of CRYSTALS-Dilithium, including minor parameter changes and the "hedged" versions in Algorithm 2, as explained in the ML-DSA specification [3]. Despite some minor differences, to compare the efficacy of our implementation, several state-of-the-art full hardware implementations for Round-3 Dilithium based on FPGA platforms with the best results are listed in Table 3. As mentioned above, these implementations vary in their approaches, such as Lightweight, High-performance, or others. To compare the effectiveness of these implementations, we employ the area-time tradeoff metric. The ATP values are calculated by multiplying the latency with the number of utilized LUTs, FFs, DSPs and BRAMs denoted as ATP_LUT, ATP_FF, ATP_DSP and ATP_BRAM, respectively. In this metric, our work serves as the "center point" for comparison, with a value set to 1. Lower values indicate better performance, as both area and time consumption should be minimized. Additionally, to ensure a fair comparison despite differences in hardware platforms, we use an equivalence conversion method. This method, previously used in [27], approximates the number of LUTs, FFs, DSPs, and BRAMs to equivalent slices (EqS). By this way when comparing results implemented on 7 series FPGAs platforms, such as Artix-7, they can be evaluated with half of the equivalent results on UltraScale+ platforms. Due to the lack of power consumption data in previous works [9], [10], [11], [12], [13], we do not evaluate the power efficiency between FPGA-based implementations.

The implementations in our work and in [9], [10] represent full hardware designs for all three security levels of Dilithium. On the other hand, in [11], [13], the authors propose independent designs for each security level. The Lightweight architecture in [12] is the smallest hardware accelerator for Dilithium, but it only supports level 5. Therefore, our architecture complexity is nearly equivalent to [9], [10], but higher than other implementations dedicated to specific security levels. Additionally, with the same lattice-based structure, Aikata et al. [15] propose a combined architecture for Kyber and Dilithium schemes. For a comprehensive comparison of resource utilization and performance in Keygen, Sign and Verify in our implementation and related works are listed in Table 3. As discussed earlier, the execution time of the Sign algorithm varies depending on the rejection loop conditions. Therefore, in Table 3, the parameters for the Sign column are split into the best-case scenario and average case, respectively. For an effective evaluation of the design, Figs. 10-13 present a fair comparison using the ATP and EqS metrics for corresponding parameters of level 5-the highest security level supported by Dilithium. These figures present a comparative analysis of the ATP_LUT, ATP_FF, ATP_DSP and ATP_BRAM parameters among the four most advanced designs available.

The work of Land [13] is one of the earliest designs for Round 3 Dilithium. This design has several non-optimized aspects, such as the use of separated NTT/INTT blocks and multiplication blocks, which could be combined to achieve efficiency without increasing algorithm execution time.Thus, based on the data in Table 3, even with the EqS metric, it is evident that our design achieves significantly better performance than this design, especially in security level 5.

In [12], Naina proposed a lightweight implementation for level 5 Dilithium on the Zynq Ultrascale+ hardware platform. With a focus on minimizing hardware utilization, this design achieves the best results in terms of LUTs and FFs. However, regarding execution time, this design only provides data for the best-case scenario of the Sign algorithm without specific information about the average execution time of this

TABLE 3. Comparison of FPGA-based implementations for Dilithium signature scheme.

Sec. lvl	Design	Device	Freq. (MHz)	Area LUT	Area FF	Area DSP	Area BRAM	Keygen cycles	Keygen μs	Sign cycles[d]	Sign μs[d]	Verify cycles	Verify μs
II	Aikata [15][a,e]	ZUs+	270	23277	9798	4	24	14594	54	31662/-	117/-	15423	57
	Land [13][b,e]	Artix-7	163	27433	10681	45	15	18761	115	19423/76613	119/470	19687	121
	Wang [11][b]	Zynq-7k	159	18558	7342	10	17	7757	49	-/52038	-/327	7675	48
	Zhao [10][c]	Artix-7	96.9	29998	10366	10	11	4172	43	8361/31600	86.3/326.1	4422	45.6
	Beckwith [9][c,e]	VUs+	256	53907	28435	16	29	4875	19	10945/29876	43/117	6582	26
	This work[c,e]	**VUs+**	**300**	**49753**	**23477**	**16**	**27.5**	**3320**	**11.1**	**10804/29219**	**36/97.4**	**4137**	**13.8**
III	Aikata [15][a,e]	ZUs+	270	23277	9798	4	24	23619	87	48446/-	171/-	26124	97
	Land [13][b,e]	Artix-7	145	30900	11372	45	21	33102	229	26979/123218	186/852	32050	222
	Wang [11][b]	Zynq-7k	159	19614	8466	10	21	12982	82	-/89213	-/561	11232	71
	Zhao [10][c]	Artix-7	96.9	29998	10366	10	11	5851	60.4	11399/49496	117.6/510.8	6181	63.8
	Beckwith [9][c,e]	VUs+	256	53907	28435	16	29	8291	32	16178/49437	63/193	9724	39
	This work[c,e]	**VUs+**	**300**	**49753**	**23477**	**16**	**27.5**	**5330**	**17.8**	**14859/46274**	**49.5/154.2**	**5846**	**19.5**
V	Aikata [15][a,e]	ZUs+	270	23277	9798	4	24	39737	147	70179/-	260/-	46671	173
	Naina [12][b,e]	ZUs+	391	13975	6845	4	35	63.2k	162	113.9k/-	291/-	67.9k	174
	Land [13][b,e]	Artix-7	140	44653	13814	45	31	50892	364	36609/145912	261/1042	52712	377
	Wang [11][b]	Zynq-7k	159	20973	9677	10	28	20185	127	-/93708	-/589	15875	100
	Zhao [10][c]	Artix-7	96.9	29998	10366	10	11	8765	90.5	15790/55321	163/570.9	9039	93.3
	Beckwith [9][c,e]	VUs+	256	53907	28435	16	29	14037	55	24358/55070	95/215	14642	57
	This work[c,e]	**VUs+**	**300**	**49753**	**23477**	**16**	**27.5**	**8226**	**27.4**	**21199/50848**	**70.7/169.5**	**8062**	**26.9**

[a]: multiple schemes, [b]: support dependent level, [c]: support all levels,
[d]: best-case / average-case scenarios, [e]: include packing/unpacking.

algorithm. Despite the differences in design approach, as our design supports various security levels, our implementation demonstrates better efficiency in almost all statistics when evaluated using the ATP metric, as shown in Figs. 10-13.

Wang et al. [11] used three independent cores for the corresponding Dilithium's security levels. This work is software/hardware codesign, where bit packing and unpacking are not included in the hardware part. Therefore, when comparing by ATP metric in level 5, a direct comparison may not be totally fair as our design, which supports 3 levels, is more complex and resource-consuming. However, our implementation showcases a slight improvement in terms of memory utilization and an significant performance enhancement.

Zhao et al. [10] presents a compact design for Dilithium on the low-end Artix-7 platform. Although it also supports all three levels, it lacks the flexibility of our design. Unlike our design, its results are stored only in its memory and it does not support the packing and sending of results to connect with external hardware/software platforms. Dilithium is a public key cryptosystem and the ability to pack and send public keys and digital signatures to other platforms is fundamental. Without this capability, the system loses much of its practicality. In terms of latency, their average execution time statistics do not include the data input receiving and sending processes.

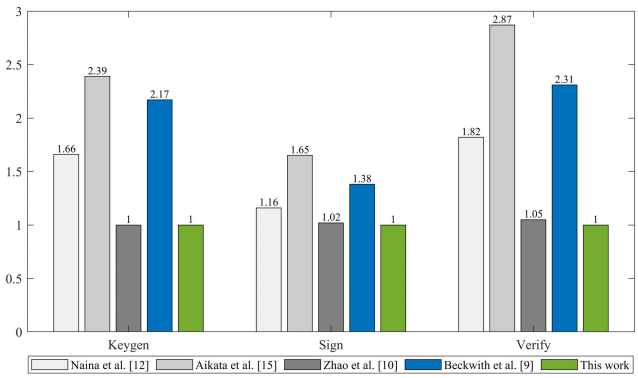

FIGURE 10. Comparison of Area × Time metric (LUTs × s).

The execution time for Verify algorithm is calculated only in cases where the public key remains unchanged. In similar cases, our work only requires 3179, 4819 and 7232 CCs for the three levels. This work also suffers from the complexity of the hash module, leading to a low operating frequency. Therefore, even with different platforms, our work offers improved latency and the potential for higher operating frequency, resulting in enhanced time execution, while

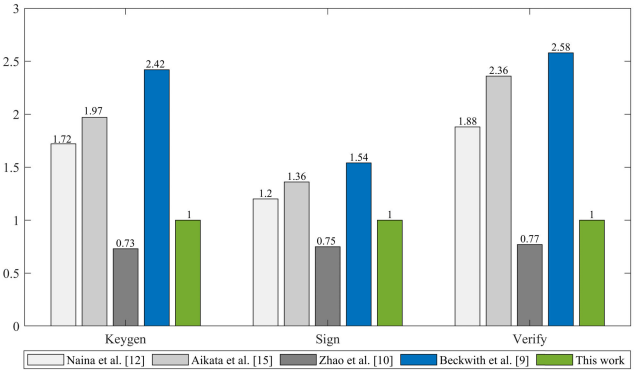

FIGURE 11. Comparison of Area × Time metric (FFs × s).

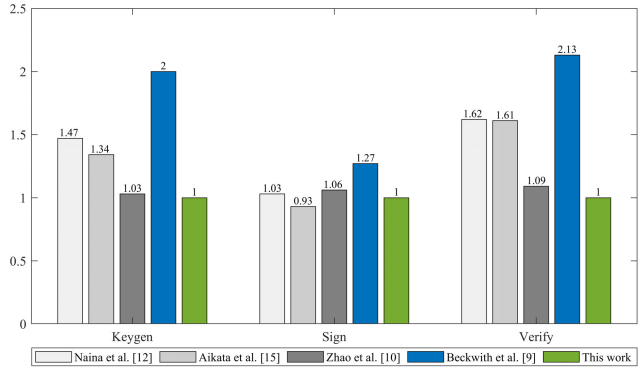

FIGURE 12. Comparison of Area × Time metric (DSPs × s).

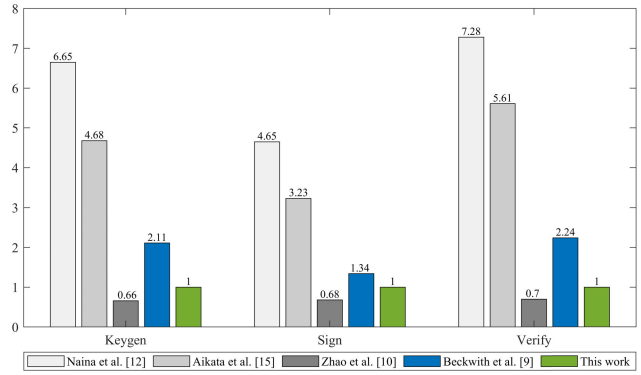

FIGURE 13. Comparison of Area × Time metric (BRAMs × s).

maintaining equivalent hardware consumption in ATP_LUT and ATP_DSP parameters, using ATP and EqS metrics.

The work of [9] presented a design similar to ours, provides support for three security levels. However, it uses the Minerva tool [28] to determine the maximum frequency, which represents a different benchmark compared to other designs. By enhancing the versatility of the arithmetic module and optimizing the hash module, specifically tailored for Dilithium, our architecture has achieved a 2× improvement in all ATP parameters during Keygen and Verify algorithms. Our results also showcase improved Sign algorithm execution times and more efficient resource utilization, detailed in Table 3.

Recently, a notable unified architecture for Kyber and Dilithium was introduced by Aikata et al. [15]. When the overall area consumption of this work is analyzed in Table 3, it utilizes 22184 LUTs, 8596 FFs, 4 DSPs, and 24 BRAMs for the hardware part without the submodules solely supporting Kyber. When evaluated using the ATP metric based on these statistics, our work exhibits substantial improvements in almost all parameters, as shown in Figs. 10-13.

In summary, our implementation exhibits the lowest latency in both clock cycle and time execution compared to the other designs. Furthermore, when compared to the combined architecture supporting all security levels of Dilithium in [9], our design demonstrates efficiency improvements ranging from 1.27∼2.58×, as evaluated by the ATP metric.

In comparison to the software implementation, our hardware implementation in this work demonstrates significant performance improvements. Seo et al. [6] reported an optimized NEON implementation on an 8-core ARM v8.2 64-bit CPU mounted on Jetson AGX Xavier. For Dilithium level 5, their implementation takes 542μs for Keygen, 625μs for Verify and 1001μs for the best-case scenario of Sign. In the same tasks, our hardware implementation outperforms their software implementation, achieving a speed-up of 19.8×, 23.2× and 14.2×, respectively. Additionally, Becker et al. [7] presented a software implementation on high-end CPUs, specifically the ARMv8 Apple M1 Firestorm core at 3.2GHz. Their reported execution times for the three main algorithms of Dilithium-III are 48μs, 114μs and 33μs. In terms of time execution, our hardware implementation achieves improvements of 2.7×, 0.74× and 1.7× for Keygen, Sign and Verify, respectively. The similarly significant results are showcased when compared with one of the most notable Dilithium implementations on the RISC-V CVA6 core on the Xilinx ZCU106 evaluation board [8], where the execution times of highest security level Dilithium are 56.21ms for Keygen, 176.3ms and 59.37ms for Sign and Verify.

V. CONCLUSION

In this paper, we present an efficient and low-latency hardware implementation of ML-DSA, which supports all main algorithms at three security levels of NIST. Our implementation enables the complete execution of all ML-DSA tasks independently or in interaction with other software/hardware platforms to accelerate specific processes. When compared with the state-of-the-art hardware architecture for all three security levels of ML-DSA, we achieve the lowest latency and efficiency in terms of area/performance tradeoffs, with a significant improvement of 1.27 to 2.58× as evaluated by the ATP metric. Additionally, this work introduces optimized arithmetic operations and hashing & sampling modules tailored to the characteristics of ML-DSA, enhancing the overall efficiency of the scheme. These modules can also work independently as accelerators to support software platforms, handling the two most time-consuming parts of these algorithms. In summary, our proposed architecture

exhibits potential adaptability to diverse applications of digital signatures, offering resilience against both quantum and classical attacks.

REFERENCES

[1] P. W. Shor, "Polynomial-time algorithms for prime factorization and discrete logarithms on a quantum computer," *SIAM Rev.*, vol. 41, no. 2, pp. 303–332, Jan. 1999.

[2] G. Alagic et al., "Status report on the third round of the NIST post-quantum cryptography standardization process," U.S. Dept. Commerce, NIST, 2022.

[3] NIST. *FIPS 204-Module-Lattice-Based Digital Signature Standard*. Accessed: Aug. 2023. [Online]. Available: https://csrc.nist.gov/pubs/fips/204/ipd

[4] S. Bai et al. *CRYSTALS-Dilithium Algorithm Specifications and Supporting Documentation (Version 3.1)*. Accessed: Feb. 2021. [Online]. Available: https://pq-crystals.org/dilithium/data/dilithium-specification-round3-20210208.pdf

[5] *Federal Information Processing Standards Publication (FIPS) NIST FIPS 186-5*, National Institute of Standards and Technology, Digital Signature Standard (DSS), Dept. Commerce, Washington, DC, USA, 2023, doi: 10.6028/NIST.FIPS.186-5.

[6] S. C. Seo and S. An, "Parallel implementation of CRYSTALS-dilithium for effective signing and verification in autonomous driving environment," *ICT Exp.*, vol. 9, no. 1, pp. 100–105, Feb. 2023.

[7] H. Becker et al., "Neon NTT: Faster dilithium, kyber, and saber on Cortex-A72 and Apple M1," *IACR Trans. Cryptograph. Hardware Embedded Syst.*, vol. 2022, no. 1, pp. 221–244, 2022.

[8] P. Nannipieri, S. Di Matteo, L. Zulberti, F. Albicocchi, S. Saponara, and L. Fanucci, "A RISC-V post quantum cryptography instruction set extension for number theoretic transform to speed-up CRYSTALS algorithms," *IEEE Access*, vol. 9, pp. 150798–150808, 2021.

[9] L. Beckwith et al., "High-performance hardware implementation of CRYSTALS-Dilithium," in *Proc. Int. Conf. Field-Program. Technol. (ICFPT)*, Auckland, New Zealand, Dec. 2021, pp. 1–10.

[10] C. Zhao, N. Zhang, H. Wang, B. Yang, W. Zhu, Z. Li, M. Zhu, S. Yin, S. Wei, and L. Liu, "A compact and high-performance hardware architecture for CRYSTALS-Dilithium," *IACR Trans. Cryptograph. Hardw. Embedded Syst.*, vol. 2021, pp. 270–295, Nov. 2021.

[11] T. Wang, C. Zhang, P. Cao, and D. Gu, "Efficient implementation of dilithium signature scheme on FPGA SoC platform," *IEEE Trans. Very Large Scale Integr. (VLSI) Syst.*, vol. 30, no. 9, pp. 1158–1171, Sep. 2022.

[12] N. Gupta, A. Jati, A. Chattopadhyay, and G. Jha, "Lightweight hardware accelerator for post-quantum digital signature CRYSTALS-Dilithium," *IEEE Trans. Circuits Syst. I, Reg. Papers*, vol. 70, no. 8, pp. 3234–3243, Aug. 2023.

[13] G. Land, P. Sasdrich, and T. Guneysu, "Hard crystal-implementing dilithium on reconfigurable hardware," in *Proc. 20th Int. Conf. Smart Card Res. Adv. Appl. (CARDIS)*, Nov. 2021, pp. 210–230.

[14] S. Ricci et al., "Implementing CRYSTALSDilithium signature scheme on FPGAs," in *Proc. 16th Int. Conf. Availability, Rel. Secur.*, Aug. 2021, pp. 1–11.

[15] A. Aikata, A. C. Mert, M. Imran, S. Pagliarini, and S. S. Roy, "KaLi: A crystal for post-quantum security using kyber and dilithium," *IEEE Trans. Circuits Syst. I, Reg. Papers*, vol. 70, no. 2, pp. 747–758, Feb. 2023.

[16] V. Lyubashevsky, "Fiat-shamir with aborts: Applications to lattice and factoring-based signatures," in *Proc. Int. Conf. Theory Appl. Cryptol. Inf. Secur.* Berlin, Germany: Springer, 2009, pp. 598–616.

[17] Y. Wang and M. Wang, "Module-LWE versus Ring-LWE, revisited," *IACR Cryptol. ePrint Arch.*, vol. 930, Aug. 2019.

[18] E. Kiltz, V. Lyubashevsky, and C. Schaffner, "A concrete treatment of Fiat Shamir signatures in the quantum random-oracle model," in *Proc. 37th Annu. Int. Conf. Theory Appl. Cryptograph. Techn.*, Tel Aviv, Israel, Apr. 2018, pp. 552–586.

[19] L. Groot Bruinderink and P. Pessl, "Differential fault attacks on deterministic lattice signatures," *IACR Trans. Cryptograph. Hardw. Embedded Syst.*, vol. 2018, pp. 21–43, Aug. 2018.

[20] P. Longa and M. Naehrig, "Speeding up the number theoretic transform for faster ideal lattice-based cryptography," in *Proc. 15th Int. Conf. Cryptol. Netw. Secur.*, Milan, Italy, Nov. 2016, pp. 124–139.

[21] S. He and M. Torkelson, "A new approach to pipeline FFT processor," in *Proc. Int. Conf. Parallel Process.*, Aug. 1996, pp. 766–770.

[22] *UltraScale Architecture Memory Resources Advance Specification User Guide*, document UG573 (v1.1) Aug. 14, 2014.

[23] *FIPS PUB 202: SHA-3 Standard: Permutation-Based Hash Extendable-Output Functions*, National Institute of Standards and Technology, Gaithersburg, MD, USA, Aug. 2015.

[24] G. Bertoni, J. Daemen, S. Hoffert, M. Peeters, and G. V. Assche. (2020). *Keccak VHDL*. [Online]. Available: https://keccak.team/hardware.html

[25] K. K. Parhi, C.-Y. Wang, and A. P. Brown, "Synthesis of control circuits in folded pipelined DSP architectures," *IEEE J. Solid-State Circuits*, vol. 27, no. 1, pp. 29–43, Jan. 1992.

[26] N. Zhang, B. Yang, C. Chen, S. Yin, S. Wei, and L. Liu, "Highly efficient architecture of NewHope-NIST on FPGA using low-complexity NTT/INTT," *IACR Trans. Cryptograph. Hardw. Embedded Syst.*, vol. 2020, pp. 49–72, Mar. 2020.

[27] P. Duong-Ngoc, S. Kwon, D. Yoo, and H. Lee, "Area-efficient number theoretic transform architecture for homomorphic encryption," *IEEE Trans. Circuits Syst. I, Reg. Papers*, vol. 70, no. 3, pp. 1270–1283, Mar. 2023.

[28] F. Farahmand, A. Ferozpuri, W. Diehl, and K. Gaj, "Minerva: Automated hardware optimization tool," in *Proc. Int. Conf. ReConFigurable Comput. FPGAs (ReConFig)*, Dec. 2017, pp. 1–8.

QUANG DANG TRUONG (Graduate Student Member, IEEE) received the B.S. degree in electrical and electronic engineering from the Le Quy Don University of Technology, Vietnam, in 2019, and the M.S. degree in electrical and computer engineering from Inha University, in 2023, where he is currently pursuing the Ph.D. degree in electrical and computer engineering. His research interests include algorithm and architecture design for post-quantum cryptography and homomorphic encryption.

PHAP NGOC DUONG (Member, IEEE) received the M.S. degree in electronic engineering from The University of Danang, Vietnam, in 2015, and the Ph.D. degree in electrical and computer engineering from Inha University, South Korea, in 2022. He is currently a Postdoctoral Fellow with the Artificial Intelligence System-on-Chip (AI-SoC) Research Center, Inha University; and a Lecturer with The University of Danang—Vietnam-Korea University of Information and Communication Technology, Vietnam. His research interests include algorithm and architecture for digital signal processing, hardware acceleration architecture, post-quantum cryptography, and homomorphic encryption.

HANHO LEE (Senior Member, IEEE) received the M.Sc. and Ph.D. degrees in electrical and computer engineering from the University of Minnesota, Minneapolis, in 1996 and 2000, respectively. He was a member of the Technical Staff with Lucent Technologies (Bell Labs Innovations), Allentown, from April 2000 to August 2002. From August 2002 to August 2004, he was an Assistant Professor with the Department of Electrical and Computer Engineering, University of Connecticut, USA. In August 2004, he joined the Department of Information and Communication Engineering, Inha University, Incheon, South Korea. He is currently a Professor with the Department of Information and Communication Engineering, Inha University, where he leads the Digital Integrated Systems Laboratory and the Director of the Artificial Intelligence System on Chip (AI-SoC) Research Center. He was a Visiting Researcher with the Electronics and Telecommunications Research Institute (ETRI), South Korea, in 2005. He was a Visiting Scholar with Bell Labs, Alcatel-Lucent, Murray Hill, USA, from 2010 to 2011, and a Visiting Professor with The University of Texas at Dallas, USA, from 2017 to 2018. His research interests include algorithm and architecture design for post-quantum cryptographic, homomorphic encryption, forward error correction coding, artificial intelligence, and digital signal processing.