`KaLi`: A Crystal for Post-Quantum Security

Aikata Aikata, Ahmet Can Mert, Malik Imran, Samuel Pagliarini, Sujoy Sinha Roy

***Abstract*—Quantum computers pose a threat to the security of communications over the internet. This imminent risk has led to the standardization of cryptographic schemes for protection in a post-quantum scenario. We present a design methodology for future implementations of such algorithms. This is manifested using the NIST selected digital signature scheme CRYSTALS-Dilithium and key encapsulation scheme CRYSTALS-Kyber. A unified architecture, `KaLi`, is proposed that can perform key generation, encapsulation, decapsulation, signature generation, and signature verification for all the security levels of CRYSTALS-Dilithium, and CRYSTALS-Kyber. A unified yet flexible polynomial arithmetic unit is designed that can processes Kyber operations twice as fast as Dilithium operations. Efficient memory management is proposed to achieve optimal latency.**

`KaLi` is explicitly tailored for ASIC platforms using multiple clock domains. On ASIC 28nm/65nm technology, it occupies 0.263/1.107 mm^2 and achieves a clock frequency of 2GHz/560MHz for the fast clock used for memory unit. On Xilinx Zynq Ultrascale+ZCU102 FPGA, the proposed architecture uses 23,277 LUTs, 9,758 DFFs, 4 DSPs, and 24 BRAMs, and achieves a 270 MHz clock frequency. `KaLi` performs better than the standalone implementations of either of the two schemes. This is the first work that provides a unified design in hardware for both schemes.

***Index Terms*—CRYSTALS-Dilithium, CRYSTALS-Kyber, Cryptoprocessor, NIST PQC Standardized**

I. Introduction

Communication over the internet forms the backbone of the digitalized world. Every communication packet passes through various insecure channels and untrusted servers before reaching the destination. Data and communication leaks in the past led to the development of public key cryptographic (PKC) schemes to ensure end-to-end security and privacy of communication. These schemes use the hard problems of the discrete logarithm, integer factorization, etc. In 1994, Peter Shor proposed that an algorithm that can help a powerful quantum computer solve them in polynomial (realistic) time, thus breaking the classical PKC schemes. Since then, the past eighteen years have witnessed a giant leap in the development of quantum computers. In 2019, Google claimed quantum supremacy by developing a 53-qubit quantum computer *Sycamore* [1]. Sycamore could solve a task in 200 seconds which would take a classical computer 10,000 years. Various labs across the world have developed even stronger quantum computers. This raises the existential question of whether our communication packets containing emails, passwords, etc., are already insecure. The answer to this is - *yes*. Even though quantum computers built until now are not strong enough to break classical public key cryptography, emails and passwords sent now can be stored and decrypted later.

This inevitable breach of security paved way for the development of post-quantum secure PKC schemes based on the hard problems that are safely sustained in a post-quantum scenario. Many standardizations were launched to select the best PKC candidates for digital signature and key-encapsulation algorithms. Key encapsulation schemes allow the communicating parties to agree on the same key securely, which can then be used for symmetric key-based encryption-decryption of messages. Thus, ensuring the security and privacy of the communications. A digital signature scheme allows the receiver to verify the authenticity of the messages. Both these schemes will replace the classical PKC schemes in various applications, like the TLS networking protocol.

These standardizations have now concluded, and the industry is now starting to gear up towards implementing the standardized candidates. After finalizing the implementations, a transition phase will start for all the devices to switch from classical to post-quantum secure PKC schemes. This transition will not only take years but also lead to a large amount of wastage in terms of chips and hardware resources which are now obsolete. However, now that we know that change is inevitable, and what we believe to be secure now might again be broken in the next 10-20 years, there is an urgent need for a design methodology for future implementations to prevent loss of time and hardware resources.

This work proposes a design methodology that covers three vital aspects for the future implementations of the PKC algorithms. The first is the need to make a *unified design*. A majority of PKC applications require both digital signature and key encapsulation schemes. Therefore, the design decisions should be adapted to help unify the two algorithms for saving area via resource sharing. Secondly, the design must be *compact*. The new PKC schemes require much larger memory and logical units to store and process the keys. If we do not attempt to make these designs compact, a lot of resource constrained CPUs that were designed for classical PKC schemes will be rendered inoperable. The final aspect is *agility/flexibility*. The architecture design should consider the ever-evolving nature of these algorithms. It will not only help prevent the huge wastage of hardware resources, but also enable a smooth transition.

To exhibit the applicative advantages of this design

Aikata, Ahmet Can Mert, and Sujoy Sinha Roy are affiliated to Institute of Applied Information Processing and Communications, Graz University of Technology, Graz, Austria. Their work was supported in part by Semiconductor Research Corporation (SRC) and the State Government of Styria, Austria – Department Zukunftsfonds Steiermark.{aikata, ahmet.mert, sujoy.sinharoy}@iaik.tugraz.at

Malik Imran and Samuel Pagliarini are with the Centre for Hardware Security, Tallinn University of Technology, Tallinn, Estonia Their work has been partially conducted in the project "ICT programme" which was supported by the European Union through the European Social Fund. It was also partially supported by European Union's Horizon 2020 research and innovation programme under grant agreement No 952252 (SAFEST). {malik.imran, samuel.pagliarini}@taltech.ee

methodology, we take the recent NIST finalized lattice-based digital signature scheme CRYSTALS-Dilithium and key-encapsulation scheme CRYSTALS-Kyber. We unify the extensive building blocks of these schemes and call the resultant architecture of the two *CRYSTALS* schemes as `KaLi`. The design choices for `KaLi` favor reducing area over improving performance. As a step toward agility, `KaLi` is modeled as an instruction-set cryptoprocessor. From here on, we will refer to CRYSTALS-Dilithium as Dilithium, and CRYSTALS-Kyber as Kyber.

Prior works in literature propose the hardware implementation of PKC schemes. Most of them focus on standalone efficient implementations [2]–[15]. The real-life applications would require both the types of schemes. Therefore, these works fail to provide complete area and timing results for the implementations that make the communication post-quantum secure. The authors in [16], [17] present hardware/software (HW/SW) co-designs for Dilithium and Kyber. Since they keep some part of the design in software, it is not sufficient to provide a good estimate for hardware-only architectures. There is a need for a unified implementation of these two types of schemes completely in hardware to get better performance. We show how `KaLi` follows the proposed design methodology and performs better than the state of the art.

Our contributions can be summarized as follow:

1) Polynomial multiplication is the most computation-intensive operation in the Dilithium signature scheme and Kyber encapsulation scheme. We propose a compact polynomial multiplier architecture that works optimally for the two cryptographic algorithms. Dilithium has a 23-bit prime modulus, whereas Kyber has a 12-bit prime modulus. A unified polynomial arithmetic unit is designed for both, Dilithium and Kyber, to save time and area. This unit has a 24-bit datapath. The core operations: addition, subtraction, multiplication, and modular reduction, are made flexible to either process two sets of 12-bit Kyber coefficients or one set of 23-bit Dilithium coefficients. This, in combination with an efficient memory management, enables performing arithmetic operations for Kyber twice as fast as Dilithium.
2) We customized the Keccak-based SHA-SHAKE and pseudo-random number generation unit to make an efficient sampling unit for both Dilithium and Kyber. The samplers for the two schemes are unified and added into the Keccak block to prevent redundant write and read of pseudo-random numbers during sampling. The remaining primitive building blocks of Dilithium and Kyber are designed discretely while ensuring low area consumption, simplicity and flexibility. The proposed arithmetic units altogether form the unified cryptoprocessor `KaLi`. It can perform key generation, encapsulation, decapsulation, signature generation, and signature verification operations for all security levels of Dilithium and Kyber. This is the **first** work that implements a unified cryptoprocessor for Kyber and Dilithium solely in hardware.
3) We propose an instruction set architecture for flexibility. The instructions are divided into two sets, and `KaLi` can run instructions from these two sets in *parallel*, thus improving the latency of the design, while keeping its area consumption low. This leads to a 35% reduction in run-time.
4) `KaLi` is engineered separately for the ASIC platform to reduce area overhead. It uses two clock domains, where the memory unit works at a higher clock frequency than the logic unit. This allowed us to use single port memory instead of dual port memory used in FPGA implementation, thus reducing the area consumption.

The paper is organized as follows. Section II provides a high-level overview of Dilithium and Kyber. The major contributions of the paper are described in Section III. It includes the design methodology for implementing the PQC schemes and implementation details. In Section IV, we give the results and compare them with the existing works in literature. Section V concludes our paper.

II. Preliminaries

Kyber and Dilithium are part of the Cryptographic Suite for Algebraic Lattices (CRYSTALS), which are recently selected for standardization by the American National Institute of Standards and Technology (NIST). Kyber's security relies on the hardness of solving learning-with-errors in module lattices (MLWE), while Dilithium's security is based on MLWE and Shortest Integer Solution (SIS) problems. The polynomials and algebraic operations are assumed to be over the polynomial ring $\mathcal{R}_q = \mathbb{Z}_q[x]/x^n + 1$. For Dilithium $n = 256$ and $q = 8380417 = 2^{23} - 2^{13} + 1$, and for Kyber $n = 256$ and $q = 3329 = (2^{12} - 3 \cdot 2^8 + 1)$. Next, we give a brief overview of these schemes and their building blocks.

A. Dilithium

This digital signature scheme has three main algorithms: key generation, signature generation, and signature verification. The sender generates a public and secret key using the key generation algorithm. Then he uses his private key to sign a message using the signature generation algorithm. The receiver can verify the signature using the sender's public key and signature verification algorithm. The signature generation algorithm continues to generate a signature until a valid signature is generated. For a signature to be valid, a set of constraints have to be satisfied to ensure that the signature does not bear any similarity with the message. Readers may refer to [18] for the original specification of Dilithium. Dilithium has three variants for NIST security levels 2, 3, and 5.

Several building blocks used by these algorithms are explained below.

- **Polynomial generation**: SHAKE-128 is used to generate the polynomials of the public matrix $\boldsymbol{A} \in R_q^{k \times \ell}$ by expanding the seed $\rho \in \{0,1\}^{256}$ along with 16-bit nonce values. The secret polynomial vectors $\boldsymbol{s}_1$ and $\boldsymbol{s}_2$ $\in S_\eta^\ell \times S_\eta^k$ are generated using SHAKE-256. For each polynomial, the seed ς and a 16-bit nonce are fed to SHAKE-256 and passed through rejection sampling in the range $\{-\eta, \eta\}$. The two types of generations are named as $\texttt{ExpandA}()$ and $\texttt{ExpandS}()$. Another function

ExpandMask(), is used to generate a polynomial vector in the range $[0, 2\gamma_1 - 1]$. For this too, the SHAKE output is processed using a rejection sampler. SampleInBall() is used during signature generation and verification, to generate a polynomial with only τ coefficients set to $+1$ or -1 and the remaining coefficients as 0.

- **Polynomial Arithmetic**: Polynomial multiplications are performed using the Number Theoretic Transform (NTT) method. The addition and subtraction operations are coefficient-wise linear operations.
- **Hash functions**: SHAKE-256 is used to make a collision resistant hash function- CRH().
- **Power2Round** : The function, $\texttt{Power2Round}_q()$, takes an element $r = r_1 \cdot 2^d + r_0$ and returns r_0 and r_1, where $r_0 = r \mod {}^{\pm}2^d$ and $r_1 = (r - r_0)/2^d$.
- **Decompose and other related functions**: Let α be a divisor of $q - 1$. The function $\texttt{Decompose}_q()$ is defined in the same way as Power2Round() with α replacing 2^d. The $\texttt{HighBits}_q()$/$\texttt{LoweBits}_q()$ return r_1/r_0 from the output of $\texttt{Decompose}_q()$. MakeHint uses $\texttt{HighBits}_q()$ to produce a hint $\boldsymbol{h}$. UseHint uses the hint $\boldsymbol{h}$ produced by $\texttt{MakeHint}_q()$ to recover the high-bits.

B. Kyber

Kyber is an IND-CCA2-secure key encapsulation scheme. It has three principal algorithms: key generation, encapsulation, and decapsulation. The receiver generates a public and secret key using the key generation algorithm and broadcasts the public key. When the sender wishes the send a message, he/she can encapsulate it using the receiver's public key through the encapsulation algorithm. The receiver can then decapsulate it using her/his secret key through the decapsulation scheme. Three variants of Kyber, Kyber-512, Kyber-768, and Kyber-1024 are provided for NIST Security levels 1, 3, and 5, respectively. The variants differ in module dimensions and coefficient distributions. Readers may refer to [19] for the detailed specifications of Kyber.

Kyber has the following main internal routines:

- **Pseudorandom functions**: Kyber uses PRF (SHAKE-256) and XOF (SHAKE-128) to generate the pseudo-random numbers for polynomial coefficients.
- **Hash functions**: Kyber provides functions H and G for SHA3-256 and SHA3-512, respectively, for hashing.
- **Key-derivation function** (KDF): It is instantiated using SHAKE-256 in Kyber.
- **Polynomial Arithmetic**: Kyber uses *a new method* NTT-based polynomial multiplication unit. Polynomial addition and subtractions are also supported.
- **Samplers**: Uniform sampling (Parse) is used to generate the public polynomials, and Binomial sampling (CBD) is used to generate secret and error polynomials.
- **Encode/Decode**: These modules are used to serialize/deserialize the polynomials to/from byte arrays.
- **Compress/Decompress**: They are used to reduce the size of ciphertext by discarding low-order bits. They are defined on an element $x \in \mathbb{Z}_q$ as $\lceil (2^d/q) \cdot x \rfloor \pmod{2^d}$ and $\lceil (q/2^d) \cdot x \rfloor$ respectively, where $d < \lceil \log_2(q) \rceil$. The value x' such that $x' = \texttt{Decompress}(\texttt{Compress}(a, d), d)$ is an element close to x.

C. NTT-based Polynomial Multiplication

Polynomial multiplication of $(n - 1)$-degree polynomials has been the focus of works for PQC implementations. Most implementations use the traditional NTT-based multiplication technique, while others show how methods like schoolbook $O(n^2)$, Karatsuba $O(n^{1.59})$, etc., can be used. NTT-based multiplication has a time complexity of $O(n(log\ n))$. The designers of Dilithium and Kyber select polynomials in Ring $\mathcal{R}_q = \mathbb{Z}_q[x]/x^n + 1$, where modulus q is an NTT-friendly prime. Thus, making it easier to use the fast NTT-based multiplication method.

Forward NTT transform converts an $(n - 1)$-degree polynomial (coefficient representation) to n 0-degree polynomials (value representation). Then two polynomials in their value-representation form (NTT-domain) can be multiplied pointwise to get the multiplied values in NTT-domain. Now, if we need to get the polynomial in coefficient representation again, a backward NTT transform (INTT) is used. The conversion to-and-from NTT domain has a time-complexity of $O(n(log\ n))$. Pointwise multiplication has a time-complexity of $O(n)$. Thus, a total time complexity of $O(n(log\ n))$. Various algorithms exist in the literature to facilitate these transforms. The most used ones are the Cooley-Tukey (Alg. 1) transform for NTT and Gentleman-Sande for INTT. For more information on NTT/INTT, refer to [20].

Next, we discuss the major optimizations made to realize the design methodology in the context of Dilithium and Kyber.

Algorithm 1 The Cooley-Tukey NTT Algorithm [21]

```
In:  An $n$-element vector $x = [x_0, \cdots, x_{n-1}]$ where $x_i \in [0, q-1]$
In:  $n$ (power of 2), modulus $q$ ($q \equiv 1 \pmod{2n}$)
In:  $\mathbf{g}$ (precomputed table of $2n$-th roots of unity, bit-reversed order)
Out: $x \leftarrow NTT(x)$
 1: $t \leftarrow n/2$; $m \leftarrow 1$
 2: while ($m < n$) do
 3:     $k \leftarrow 0$
 4:     for ($i \leftarrow 0; i < m; i \leftarrow i + 1$) do
 5:         $S \leftarrow \mathbf{g}[m + i]$
 6:         for ($j \leftarrow k; j < k + 1; j \leftarrow j + 1$) do
 7:             $U \leftarrow x[j]$                          ▷ Butterfly starts
 8:             $V \leftarrow x[j + t] \cdot S \pmod{q}$
 9:             $x[j] \leftarrow U + V \pmod{q}$
10:             $x[j + t] \leftarrow U - V \pmod{q}$          ▷ Butterfly ends
11:         end for
12:         $k \leftarrow k + 2t$
13:     end for
14:     $t \leftarrow t/2$; $m \leftarrow 2m$
15: end while
16: return $x$
```

III. Proposed Unified Hardware Architecture

The first and foremost goal is to unify the digital signature scheme and the key-encapsulation scheme. While doing this, it is important to ensure that the design is compact and flexible. Unification has a very straightforward three-step approach. First, we must identify the most area and time consuming building blocks, the Giants. This is because unifying the low area and time consuming building blocks (the Dwarves) will not reduce the area consumption significantly, and instead

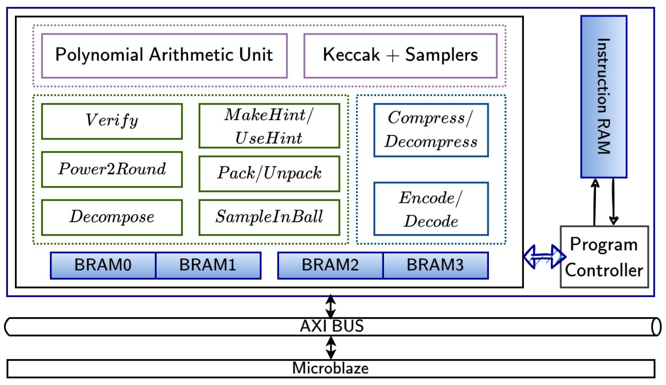


Fig. 1. High level architecture of `KaLi`

limit the flexibility of the design. The next step is to find the algorithmic synergies between the Giants of the two schemes. The final step is to discern if some of the Dwarves which are dependent on the Giants can be unified with Giants to reduce both area and time consumption.

A high-level view of the proposed architecture `KaLi` is given in Fig. 1. The Keccak-based SHA-SHAKE unit and polynomial arithmetic unit are the two Giants in both the schemes. The remaining building blocks are deemed as Dwarves since they comprise only 20% of the total area consumption. Unifying the Keccak-based SHA-SHAKE unit is relatively easy since we can use a common Keccak core for both schemes. Therefore in this section, we will discuss how we efficiently unified the polynomial arithmetic unit. We will also discuss how we efficiently manage the memory for the two schemes. Another facet of the work, the optimization for ASIC platforms, is also presented. We utilize multiple clock domains and boosted memory bandwidth budget on ASIC platforms to reduces the area consumption.

A. The colossal Giant: Polynomial Arithmetic Unit

The Polynomial Arithmetic Unit performs polynomial addition, subtraction, and multiplication. Polynomial addition and subtraction are simple coefficient-wise operations, hence cheap. Polynomial multiplication is rather complex, and it is what makes the polynomial arithmetic unit a Giant. Both schemes perform this using NTT, as discussed in Section II-C. Although the two schemes use NTT-based polynomial multiplication units, there are many differences between the two schemes that make their NTT units quite distinct.

1) A clash of the Giants: The differences between the presumed similar NTTs

The first distinction between the NTTs used by two schemes lies in the algorithm itself. The NTT-based polynomial multiplication method used in Dilithium requires the existence of $2n$-th root of unity that mandates $q \equiv 1 \pmod{2n}$. Accordingly, Dilithium uses a *complete-NTT*. After a *complete-NTT* transform of an n-degree polynomial, we get n polynomials of degree 0. In [22], Lyubashevsky *et al.* propose a new method for NTT-based polynomial multiplication that requires only $q \equiv 1 \pmod{n}$, without pre-processing and post-processing operations. This technique is adopted by Kyber, and their 12-bit prime modulus does not have a $2n$-th root of unity. Therefore, Kyber has to use an *incomplete-NTT*. An *incomplete-NTT* gives us $n/2$ polynomials of degree 1. These polynomials cannot be multiplied coefficient-wise.

For the *incomplete-INTT*, multiplication operation of two degree-1 polynomials is performed in the ring $\mathbb{Z}_q[x]/(x^2 - \omega^i)$ where ω is the n-th root of unity and i depends on the index of coefficients. For the details readers may follow original Kyber specifications [19] or related prior works in the literature [23]. Along with this, they also have a difference in datapath design. Dilithium has a 23-bit prime modulus ($2^{23} - 2^{13} + 1$), while Kyber has a 12-bit prime modulus. Therefore, while Kyber requires 12-bit adder/subtracter/multiplier units, Dilithium requires them in 23-bits. Designing a datapath for one of them and using it for the other one would lead to over-or-under saturation.

Next, we will discuss in detail how we achieved a unified polynomial multiplication unit with full utilization. The unit of interest here is the butterfly unit (BFU). Each BFU performs dyadic addition, subtraction, and multiplication, on the two input coefficients. The results are reduced by $mod\ q$. This is shown by steps 7-10 in Algo. 1. Since modulus multiplication is the most expensive operation, we will discuss how we unify this unit. Then, we will discuss how with a few more changes, the entire BFU can be consolidated.

Algorithm 2 Integer Multiplication Algorithm

```
In:  $a, b \in \mathbb{Z}_{8380417}$ or
     $a[23:12], b[23:12], a[11:0], b[11:0] \in \mathbb{Z}_{3329}$
In:  $sel \in \{0,1\}$ (0 for Dilithium and 1 for Kyber)
Out: $d = a \cdot b$ or $d = \{a[23:12] \cdot b[23:12], a[11:0] \cdot b[11:0]\}$
  1: $d_0 = (sel)\ ?\ b[23:12]\ :\ b$
  2: $m_0 = d_0 \cdot a[23:12]$
  3: $m_1 = (sel)\ ?\ (m_0 \ll 24)\ :\ (m_0 \ll 12)$
  4: $d_1 = (sel)\ ?\ b[11:0]\ :\ b$
  5: $d = d_1 \cdot a[11:0] + m1$
  6: return $d$
```

2) Flexible fusion of Modular Multiplier Unit

As discussed above, if we naively use the 23-bit Dilithium polynomial multiplier unit for Kyber, then it will always be undersaturated as half of it will be unused. Instead, if we aim to use a 12-bit Kyber unit for Dilithium it will require extra control logic but also slow down Dilithium's NTT. Therefore, we need to find a solution using a 23-bit Dilithium unit that does not lead to undersaturation. The modular multiplier unit has two parts: (i) integer multiplier and (ii) modular reduction unit. We propose an algorithm (Alg. 2) to make the integer multiplication unit designed for Dilithium flexible for Kyber. It performs two 12-bit$\times$12-bit integer multiplications. The result is added for Dilithium and concatenated for Kyber. This algorithm gives us one multiplied coefficient in the case of Dilithium and two multiplied coefficients in the case of Kyber.

The modular multiplier unit, designed to support modular multiplication using both the primes, uses two DSP units of Xilinx FPGAs. The hardware architecture of the reconfigurable integer multiplier is shown in Fig. 2. The datapath depends on the scheme type and is heavily pipelined. We used internal registers of DSP units to synchronize two DSP unit outputs and achieve a high clock frequency. Now we need to design a modular reduction unit accordingly.

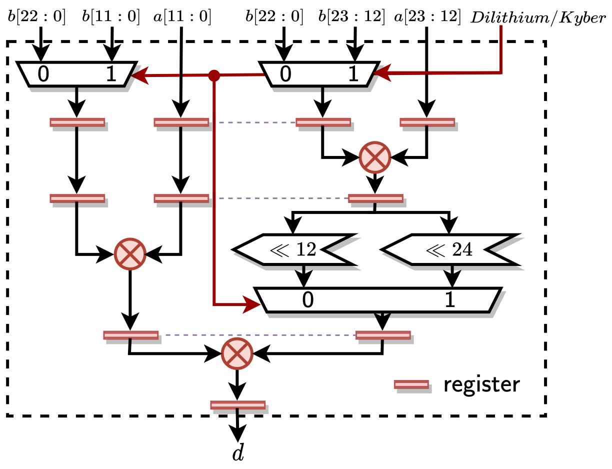


Fig. 2. Flexible yet compact integer multiplier. The red lines show control signals and black lines show data movement.

3) Versatile Modular Reduction Unit

The naive solution is to design separate modular reduction units for the two primes. It would require one modular reduction unit for Dilithium prime and two reduction units for Kyber prime, which will result in extra hardware costs. To avoid this, we propose a unified modular reduction unit. Both Dilithium ($2^{23}-2^{13}+1$) and Kyber ($2^{12}-2^{9}-2^{8}+1$) primes have pseudo-Mersenne structure. For Dilithium prime, we followed the method described in [24] which uses $2^{23} \equiv 2^{13}-1$ equation recursively. Using this equation we can reduce a 46-bit integer $d \pmod{2^{23}-2^{13}+1}$ to the integer $2^{13}d[45:24]+d[22:0]-d[45:23]$ which consists of addition/subtraction of 36-bit and 23-bit partial results. If we apply this operation recursively until the result contains only 23-bit or lower partial results, then this will unit only require adders and subtractors. For Kyber prime, we followed the add-shift-based method proposed in [23] which generates partial results using equations $2^{12} \equiv 2^{9}+2^{8}-1$ and $2^{11} \equiv -2^{10}-2^{8}-1$ recursively.

Summing all partial results using carry propagate adders (CPAs) will result in either very long carry chain or multiple pipeline stages. In order to avoid long carry chain and pipeline delay, we used carry save adders (CSAs) along with CPA. The proposed unified modular reduction unit is shown in Fig. 3. Since both the methods first generate partial results, then use CSAs for summing them, we used multiplexers to select proper partial results based on the prime used and perform addition using the CSA tree along with CPA. In Fig. 3, we also show the partial result generation for the Dilithium and Kyber primes. The boxes with 'D' and 'K' letters (in Fig. 3) represent the partial result generation circuits for Dilithium and Kyber primes, respectively. First we generate the initial partial results using the equations $2^{23} \equiv 2^{13}-1$, $2^{12} \equiv 2^{9}+2^{8}-1$ and $2^{11} \equiv -2^{10}-2^{8}-1$ iteratively. Then, the subtraction operations are converted into the additions by taking the 2's complements of negative partial results. Then, the bits of the generated partial results are moved up to reduce adder depth before being sent to the CSAs. In Fig. 3, each number inside a box represents a bit index of the input integer (0

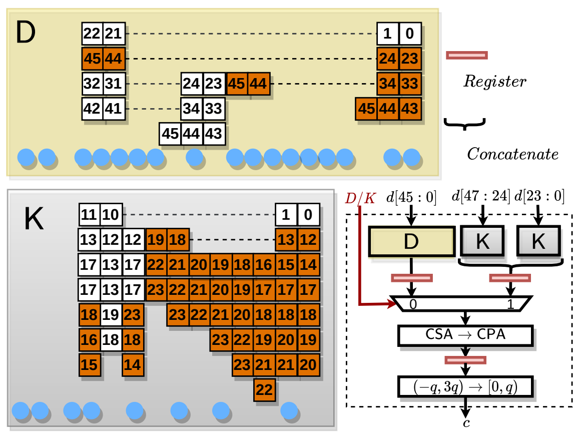


Fig. 3. Unified modular reduction unit for Dilithium and Kyber primes.

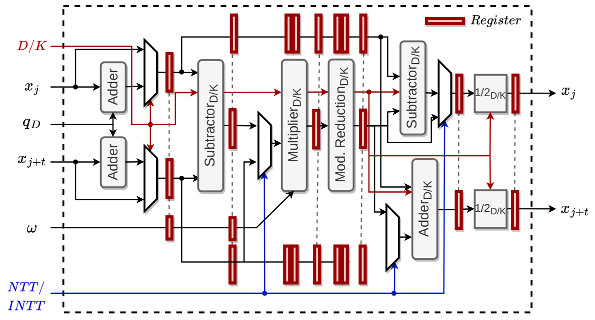


Fig. 4. Compact butterfly unit (BFU) with flexibility for both Dilithium and Kyber. The red and blue lines show control signals and black lines show data movement.

to 45 for Dilithium and 0 to 22 for Kyber). The white and brown(terracotta) boxes represent the normal and negated bits. Blue circles represent constant 1s which are generated during the 2's complement operation.

After the final addition, we also perform a final correction which brings the resulting integer from the range $(-q, 3q)$ to the range $[0, q)$. The proposed modular reduction unit can either perform one reduction for Dilithium prime or two reductions for Kyber prime. The latency of the modular reduction unit is two cycles and it is fully pipelined.

4) Coalesced datapath for the Butterfly Unit

Now that we have unified the modular multiplication unit, we propose a unified BFU (Fig. 4). It can perform one butterfly operation for Dilithium and two butterfly operations for Kyber using the same datapath. All the arithmetic units are made re-configurable to work for both schemes. New Re-configurable adder and subtractor units are shown in Fig. 5. The idea is to divide each 24-bit adder/subtractor into two small 12-bit parts and select proper input signals based on the scheme. The complete unified butterfly unit, designed using the re-configurable arithmetic units, is shown in Fig. 4.

The schoolbook multiplication for Kyber requires five multiplication operations for multiplying two 1-degree polynomials. We have two flexible butterfly units which act as four butterfly units for Kyber. This only allows four multiplications. One way to perform these five multiplications is to add another DSP multiplier just for the extra multiplication. This unit will not be useful for any other operation. To avoid this extra

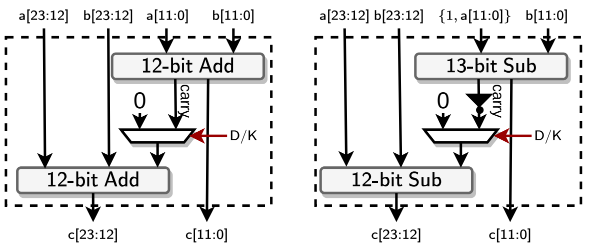


Fig. 5. Unified adder and subtractor for the butterfly unit. The red lines show control signals and black lines show data movement.

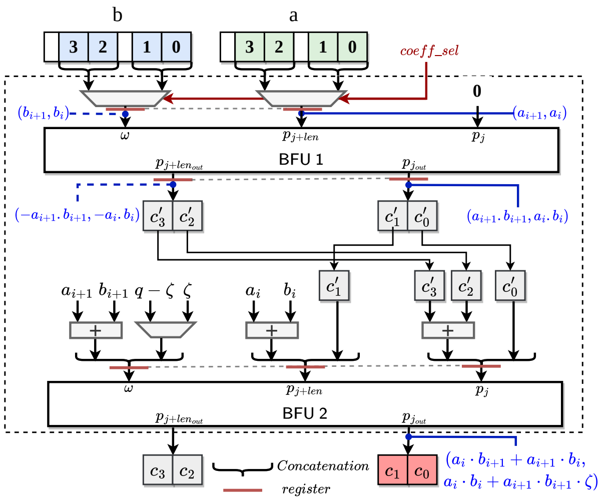


Fig. 6. Butterfly feedback unit for Kyber's NTT-domain polynomial multiplication.

multiplier, we condense five multiplications into four using Karatsuba-like reduction. Then we use these four independent butterfly units as a set of two. The output of the first set is the input for the second set. Thus, working in a feedback manner as shown in Fig. 6. The inputs and outputs for the BFUs are highlighted in blue. The control flow here is separated from the Dilithium polynomial multiplication control flow, for simplicity. The entire flow is pipelined to achieve a high clock frequency.

The coefficient consumption during NTT/INTT is shown in Fig. 7. Owing to the flexible datapath and efficient memory arrangement(discussed in the next subsection), the Kyber NTT coefficients can be consumed faster. This enables full utilization of the datapth. The complete polynomial arithmetic unit consumes 3,487 LUT, 1,918 FF, and 1 BRAM. This BRAM is used to store the powers of roots-of-unity (twiddle factors) required during NTT/INTT operation.

In the next section, we will discuss the efficient memory arrangement designed to optimally feed the polynomial arithmetic unit.

B. Memory Arrangement

The polynomial arithmetic unit is designed to consume the Kyber coefficients twice as fast as Dilithium coefficients. It requires that the memory unit feeds it at the same rate, otherwise, making these unifications will not help improve the performance. Dilithium coefficients are 23-bit, and we have designed the NTT/INTT unit using two butterfly cores. Each of the cores requires exclusive access to read/write port of a memory. Therefore, we split the memory into two blocks, each storing two Dilithium coefficients per address. For Kyber, each memory block stores four 12-bit Kyber coefficients. Fig. 8 shows the storage of Kyber polynomials in one 64-bit word of memory. One Dilithium polynomial coefficient will occupy two of these coefficients, thus requiring twice the amount of storage. It also ensures that the two required coefficients during NTT/INTT are always stored across different BRAMs. Fig. 8 shows an example of the coefficients storage during Kyber's incomplete-NTT iterations for a 15-degree polynomial.

Next, we will discuss how we used multiple clock domains to reduce the area consumption in ASIC platforms.

C. Multi-clock domains: Customization for ASIC platforms

The memory organization discussed above has two sets of BRAMs to feed the two BUF. These BRAMs are used by all the remaining building blocks as well. It is generated using dual-port BRAMs in FPGA. In ASIC, dual-port BRAMs consume more area than single-port BRAMs. Therefore, to reduce the area consumption, we decide to replace dual-port RAMs with single port RAMs, which work at a clock frequency twice as fast as the rest of the design. Using two different sources for the two clocks leads to an asynchronous setting. This creates meta-stability problems due to clock-domain crossing. To avoid these problems we decided to keep the clocking synchronous and generate the slow clock(clock_logic) using the fast clock(clock_mem).

Fig. 9 gives the description of the handshake between memory and logic. A wrapper is provided to process the simultaneous reads and writes to the memories. The read operation is given preference over the write operation to ensure data is valid when the building blocks fetch it and avoid any issues due to clock glitches. The read latency is three clock cycles, and all the building blocks are tailored accordingly. This design helps reduce the area for ASIC designs and gives. Note that a similar change will not change FPGA area consumption and instead face timing problems running the memory at a high clock frequency. Therefore, this adaptation specifically targets ASIC platforms.

Until now we discussed the major contributions of the work. Next we will briefly discuss how we efficiently implement the remaining building block. We will start with the rejection samplers used in both the schemes. These are the Giant dependent Dwarves which might help reduce the area and time consumption without compromising the flexibility of the design.

D. The Giant and the Dwarves: Keccak based SHA-SHAKE unit and the rejection samplers

Dilithium requires SHAKE-128 and SHAKE-256 for pseudo-random number generation and hashing. Kyber requires SHA3-256 and SHA3-512 for hashing and SHAKE-128 and SHAKE-256 for KDF and pseudo-random number generation. These different Keccak-based functions are implemented as modes of the same Keccak output. Therefore, we

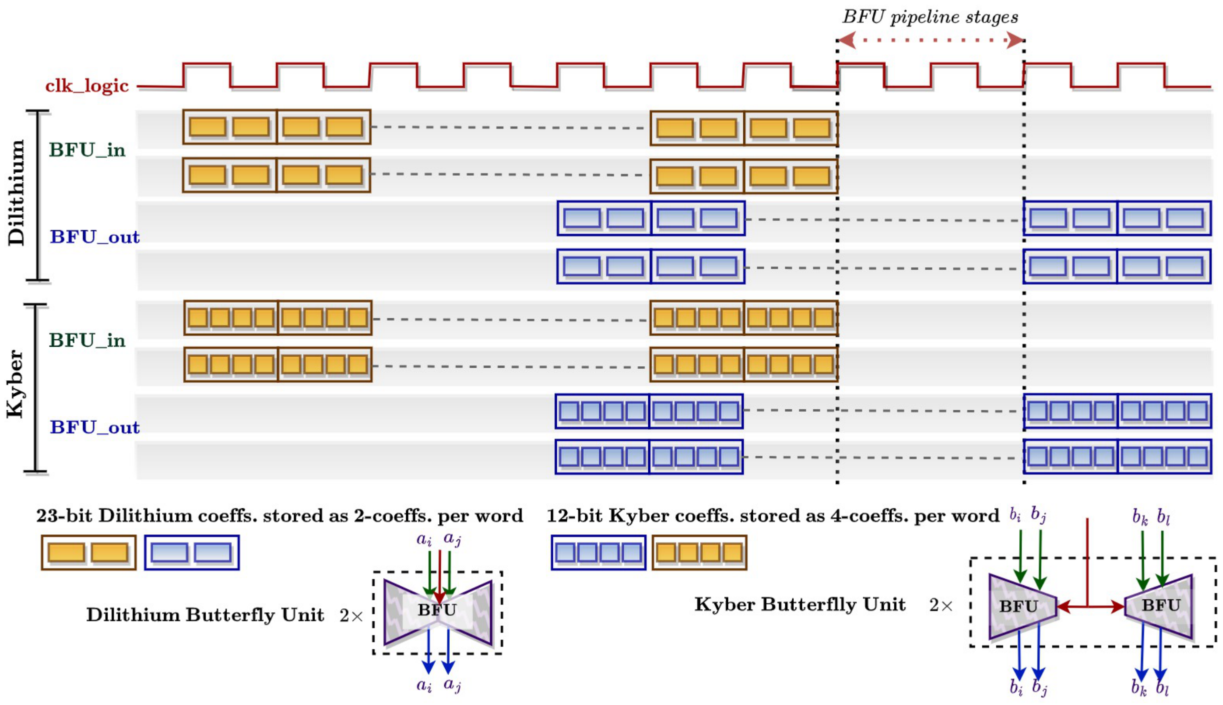

BFU pipeline stages
clk_logic
Dilithium
BFU_in
BFU_out
Kyber
BFU_in
BFU_out
23-bit Dilithium coeffs. stored as 2-coeffs. per word
12-bit Kyber coeffs. stored as 4-coeffs. per word
Dilithium Butterfly Unit 2×
BFU
Kyber Butterfly Unit 2×
BFU
BFU

Fig. 7. Timeline showing the unified butterfly unit processing Dilithium and Kyber coefficients.

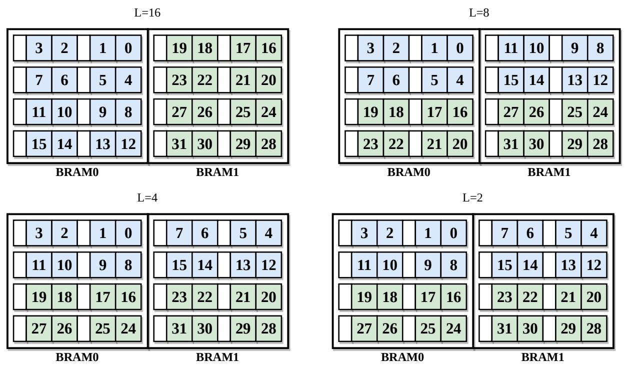


Fig. 8. Storage of coefficients during Kyber's NTT for 16-coefficient polynomial

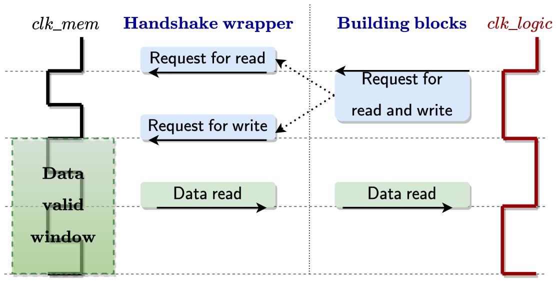


Fig. 9. The data read-and-write handshake between memory and logic unit

can use the same Keccak instance for all these modes. Both the schemes employ different sampling for the generation of secret and error polynomials. While some of these fully consume the Keccak output, the remaining have to keep track of the leftover bits.

We combine the rejection sampler with the Keccak unit using a book-keeping approach similar to [24]. It improves the performance of the sampling operation, as we do not need to store and then read the Keccak output in between. The base implementation of Keccak follows a high-speed and parallel directive. The control and data path are modified to work for rejection samplers as it depends on coefficients passing the rejection constraints. The complete Keccak unit consumes 12,326 Look Up Tables (LUT) and 3,560 Flip-Flops(FF).

We have unified all the Giants, so now we will discuss the optimized implementation techniques for the Dwarves.

E. *Optimizations for the* Dwarves

Making a design compact while keeping it agile increases the life and usability of `KaLi` on the FPGA and ASIC platforms. However, this comes with a series of challenges. We must ensure that for keeping the design agile/flexible, we do not pay a huge price in terms of area. Similarly, while making the design compact, the performance should not get worse. We now discuss how to make certain building blocks of the two schemes compact, while maintaining the flexibility.

1) *Compress/Decompress Unit*

The decompress unit performs division by power-of-two and rounding operation which is trivial to implement in hardware. On the other hand, the compress operation requires division by q and rounding. Some works in the literature use Barrett reduction and division algorithms to perform compress operation. We decide to use sufficient precision and convert division by q operation into multiplication and shift operations. The proposed multiplication-based compress algorithm is shown in Algorithm 3. The input is the Kyber coefficient x and the type of compression required d. The compressed coefficient y is returned as the output.

Algorithm 3 The Proposed Compression Algorithm

```
In: $x \in \mathbb{Z}_{3329}$, $d \in \{1, 4, 5, 10, 11\}$
Out: $y = \lceil (2^d/q) \cdot x \rfloor$
1: switch $d$ do
2:     case 1: $t = (10079 \cdot x); y = (t \gg 24) + (t[23] \gg 23)$
3:     case 4: $t = (315 \cdot x); y = (t \gg 16) + (t[15] \gg 15)$
4:     case 5: $t = (630 \cdot x); y = (t \gg 16) + (t[15] \gg 15)$
5:     case 10: $t = (5160669 \cdot x); y = (t \gg 24) + (t[23] \gg 23)$
6:     case 11: $t = (10321339 \cdot x); y = (t \gg 24) + (t[23] \gg 23)$
7: end switch
8: return $y \pmod{2^d}$
```

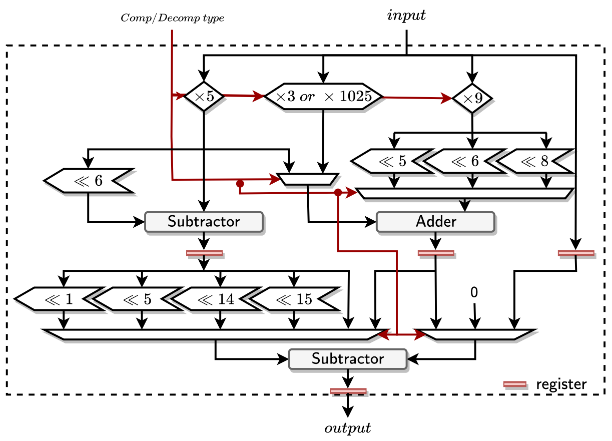


Fig. 10. Architecture of the Compress/Decompress unit. The red lines show control signals and black lines show data movement.

Since the multiplications are by constant values, we implement these operations using add and shift technique utilizing the LUTs. Fig. 10 shows the hardware architecture of this multiplication unit, used for retrieving the t values in Alg. 3. This is unified and works for both compress and decompress operations. The control flow is dependent on the type of compression or decompression required.

2) Encode/Decode Unit

Encode and decode units perform coefficient-to-byte and byte-to-coefficient conversions, for all security levels of the Kyber scheme. We used a similar idea as proposed in [25] which uses a 32-bit interface. Our architecture uses 64-bit interface and thus the proposed encode unit uses 104-bit buffer. It can encode 1-bit, 4-bit, 5-bit, 10-bit and 11-bit long coefficients. The decode unit can decode 64-bit inputs into 1-bit, 4-bit, 5-bit, 10-bit and 11-bit long coefficients using a 72-bit buffer.

3) Pack/Unpack unit

Similar to Kyber, Dilithium requires coefficient-to-byte and byte-to-coefficient conversions for various coefficient sizes. Pack and unpack units perform these conversions for all security levels of Dilithium for coefficient sizes 3, 4, 6, 10, 13, 18 and 20 bits. We again followed the idea proposed in [25] for the pack and unpack units.

The remaining building blocks of both schemes are different and unifying them would not save any area, but instead complicate the control logic and reduce the flexibility of the design. These building blocks do not require any DSP units and comprise simple bit-wise packing, unpacking, or addition/subtraction operations. Their implementation is done with sufficient pipeline stages to avoid any timing violation. They occupy only 18% of the total area consumed by the crypto processor.

F. Instruction set cryptoprocessor

We made the building blocks compact while ensuring flexibility, but this is insufficient. What happens if, in two years, the Keccak pseudo-random number generation and hashing unit is obsolete? Do we then need to redesign the entire cryptoprocessor? To counter this and increase the agility/flexibility of the architecture, we design this as an instruction set architecture (ISA). It allows the users to send instructions for operations they need.

With this flexibility comes the hardware-software communication cost. If the user has to send every instruction one by one to execute, then this communication time would take a toll on the total run-time. It also requires constant user interaction. To avoid this, we design a program controller with a small memory for storing instructions. The user just needs to send all the data and instructions in the beginning, and then hand over the control to the program controller. The program controller then ensures that all the instructions get executed correctly and returns a done signal in the end. This comprises 14% of the area consumption.

G. Running the Giants and the Dwarves in parallel

Our goal was to make the unified design compact and agile. However, does this mean we have to pay an equal price in terms of performance? To some extent, this is correct. However, we should continue to ponder on some methods that could boost the performance without increasing the area consumption. One such way is to run the Giant instructions in parallel to each other or to multiple Dwarf instructions, as shown in [24]. We make sure that two Giants, the Keccak unit and the polynomial arithmetic unit, can always run in parallel to cancel each other's run-time. It leads to a reduction of 35% in the total run-time. Following the design methodology, we design the unified cryptoprocessor- `KaLi` as shown in Fig. 1.

With this, we conclude the design methodology and implementation section and move towards the results and comparisons section.

IV. Results

In this section, we present the performance and area results of `KaLi`. The proposed architecture is described in Verilog, and it is synthesized and implemented for Zynq Ultrascale+ ZCU102 with an performance-optimized implementation strategy using Vivado 2019.1 tool. The proposed architecture achieves 270 MHz clock frequency on FPGA. The proposed architecture is also implemented with 65nm and 28nm ASIC technologies using Cadence Genus tool. On 65nm/28nm ASIC technology, it achieves 280 MHz/1 GHz for the slow clock (for logic units), and 560 MHz/2 GHz for the fast clock (for memory units).

TABLE I
PERFORMANCE RESULTS FOR DILITHIUM AND KYBER-KEM IN FPGA

Operation	Dilithium-2 Kyber-512		Dilithium-3 Kyber-768		Dilithium-5 Kyber-1024	
	Cycle	μs	Cycle	μs	Cycle	μs
Dil.Gen	14,594	54.05	23,619	87.48	39,737	147.17
Dil.Sign$_{pre}$	7,883	29.2	9,640	35.7	12,943	46.27
Dil.Sign	21,812	80.79	36,643	135.72	53,965	199.87
Dil.Sign$_{post}$	1,967	7.23	2,463	9.12	3,271	12.12
Dil.Verify	15,423	57.12	26,124	96.76	46,671	172.86
Kyb.Keygen	3,395	12.6	6,291	23.2	9,089	33.7
Kyb.Encaps	4,956	18.4	7,862	29.11	11,351	42.04
Kyb.Decaps	6,807	25.21	11,291	41.82	13,905	51.5

TABLE II
AREA OF KaLi ON THE ZYNQ ULTRASCALE+ ZCU102 FPGA PLATFORM. ALL SECURITY LEVELS OF DILITHIUM AND KYBER ARE SUPPORTED.

	Unit	LUTs	DFFs	DSPs	BRAMs
	ComputeCore	**21,228**	**9,273**	**4**	**21**
Dilithium (D)	⌊Decompose	474	338	0	0
	⌊Power2Round	55	84	0	0
	⌊MakeHint	61	124	0	0
	⌊UseHint	565	433	0	0
	⌊Encode_$\mathcal{H}$	202	233	0	0
	⌊Pack	582	181	0	0
	⌊Unpack	315	182	0	0
	⌊SampleInBall	505	285	0	0
	⌊Refresh	8	7	0	0
	⌊Verify_equality	13	76	0	0
Kyber (K)	⌊Encode	517	190	0	0
	⌊Decode	237	180	0	0
	⌊Compress/Decompress	272	376	0	0
	⌊Verify	102	216	0	0
	⌊CMOV	20	120	0	0
	⌊COPY	15	120	0	0
D+K	⌊Memory	268	12	0	20
	⌊Keccak	12,306	3,467	0	0
	⌊Multiplier	3,487	1,918	4	1
	ProgramController	**2,136**	**296**	**0**	**3**
	Total	**23,347**	**9,798**	**4**	**24**

A. *Performance and Area Results*

Table I presents the cycle count and latency (in μs) for the operations of Dilithium (key generation, signature generation, and signature verification) and Kyber (key generation, encapsulation, and decapsulation). With 270 MHz clock frequency in the FPGA, the CCA-secure key generation, encapsulation and decapsulation operations for Kyber-768 takes 23.2, 29.11 and 41.82 μs, respectively. For the best-case scenario where a valid signature is generated after the first loop iteration [24], the key generation, signature generation and signature verification operations for Dilithium-3 take 87.48, 179.91 and 96.76 μs, respectively. The ASIC implementation with 65nm/28nm technology (with 560 MHz/2 GHz clock frequency for the memory unit) can perform the operations for Kyber-768 and Dilithium-3 in 22.07/6.18, 27.59/7.73, and 39.62/11.09 μs, and 82.87/23.2, 171.03/47.89, and 91.66/25.67 μs, respectively.

In the Table II, we present the detailed utilization of each building blocks in KaLi for UltraScale+ ZCU102 platform. The proposed cryptoprocessor uses only 23,347 LUTs (8.4%), 10,121 DFFs (1.7%), 4 DSPs (0.1%) and 24 BRAMs (2.6%). On ASIC, KaLi consumes 1.107 mm^2 (769.04 KGE) in 65nm technology, and 0.263 mm^2 (747.81 KGE) in 28nm technology. Next, we compare these results with the existing works in literature.

B. *Comparison with unified designs in literature*

In [16], the authors present a unified HW/SW co-design for Dilithium and Kyber. They implement Kyber in hardware (they also provide results for Kyber with HW/SW co-design) while keeping some part of Dilithium in software. Their NTT unit occupies 25,674 LUTs, 3,137 DFFs, 64 DSPs, and 6 BRAMs on a Xilinx Artix-7 FPGA. The NTT unit alone occupies more LUT and DSP units than our entire design. On ASIC, it occupies 697 KGE on 28nm technology [16]. Our complete unified design occupies 747.8 KGE on 28nm technology. Their implementation show better performance for Kyber as they implement all building blocks of the Kyber in hardware and use 32 butterfly units for NTT. Thus, their NTT unit is $8\times$ faster and consumes more area. However, for Dilithium, the dominant scheme, KaLi shows much better results. For Dilithium-3, the key generation, signature generation, and verification have a latency of 126.7, 415.9, and 204.3 μs on 28nm technology. KaLi gives $2\times$ better performance on 65nm, an older technology.

To the best of our knowledge, no work exists in the literature that *unifies* Dilithium and Kyber solely in hardware. Therefore, next we compare our work with standalone implementations of Dilithium or Kyber in hardware.

C. *Comparison with Dilithium-only designs in literature*

Comparison with FPGA-based implementations: Table III gives the comparison of implementation results for Dilihtium-3 on FPGA platforms. Zhou et al. [3] present a HW/SW co-design in which they only implement the polynomial arithmetic unit in hardware. Thus, they consume less area but report an inferior performance. Ricci et al. [4] provide separate designs for each of the Dilithium operations. These designs in total occupy $9\times$ more area compared to our design and still perform as good as our design for signature verification. For a signature generation, their implementation shows only $3\times$ improvement. Thus, our design gives a much better area-time product.

The authors in [5], [6], [26] present Dilithium implementations which consume much more area compared to our design and still report a much lower clock frequency. Thus, lowering the performance of their designs. In [24], the authors present a unified cryptoprocessor for Dilithium and Saber. Their area is almost comparable, considering the difference between the two schemes, Kyber and Saber. We achieve a much higher clock frequency and report 1.4$\times$ better performance.

Comparison with ASIC-based implementations: Table IV gives the comparison of implementation results for Dilihtium-3 on ASIC platforms. Banerjee et al. [17] present ASIC results for HW/SW co-design of Dilithium with Round 2 parameters. KaLi outperforms them significantly in terms of performance. Our hardware only design gives $45\times$ better performance at the cost of only $7.5\times$ more area. KaLi consumes almost the same area as reported in [16] but gives a $10\times$ and $2.8\times$ better performance with 28nm and 65nm technology, respectively. [24] reports a higher number of logic gates than our design.

Thus, our FPGA and ASIC models are the most compact compared to all the existing Dilithium implementations.

TABLE III
Comparison Table for Dilithium-3 FPGA Implementations

Ref.	Plat.	Performance (in μs)	Freq. (*MHz*)	Resources (LUT/ FF/DSP/BRAM)
[3]†	Zynq	-/8.8K/9.9K	100	2.6K/-/-/-
[4]^a	US+	51.9/-/-	350	54.1K/25.2K/182/15
[4]^{b,d}		-/63.1/-	333	68.4K/86.2K/965/145
[4]^c		-/-/95.1	158	61.7K/34.9K/316/18
[5]^d	Ar.-7	229/0.3K/0.2K	145	30.9K/11.3K/45/21
[5]^e		229/0.85K/0.2K		
[26]^d	Ar.-7	60/0.12K/63.8	96.9	30K/10.34K/10/11
[26]^e		60/0.46K/63.8		
[6]^d	US+	32/63/39	145	55.9K/28.4K/16/29
[6]^e		32/193/39		
[24]^{d,f}	US+	114.7/237/127.6	200	18.5K/9.3K/4/24
KaLi^{d,f}	**US+**	**82.8/171.3/96.7**	**270**	**23K/9K/4/24**

^{a}: Implements K. Gen. ^{b}: Implements Sign. Gen. ^{c}: Implements Verify. ^{d}: Reports best-case scenario. ^{e}: Reports average-case scenario. ^{f}: Supports multiple schemes. †: HW/SW co-design.

TABLE IV
Comparison Table for Dilithium-3 ASIC Implementations

Ref.	Tech. (*nm*)	Perf.* (in μs)	Freq. (*MHz*)	Logic Gates (*KGE*)	SRAM (*KB*)
[17]†	40	18,266	72	106	40.25
[16]$^{\dagger a}$	28	747	540	697	24.75
[24]^{a,b}	65	182.3	400	854	34.82
KaLi^{a,b}	**65**	**262.69**	**280&560**	**769**	**34.82**
KaLi^{a,b}	**28**	**73.55**	**1000&2000**	**747**	**34.82**

*:Performance is measured as total time for signature generation and verification (key generation can be done offline). †: HW/SW co-design. ^{a}:Supports multiple schemes. ^{b}: Reports best-case scenario.

D. Comparison with Kyber-only designs in literature

Comparison with FPGA-based implementations: Table V gives the comparison of implementation results for Kyber-1024 on FPGA platform. Banerjee et al. [17] present an HW/SW co-design for Kyber. KaLi surpasses their performance results on both the platforms, at the cost of some area. Observe that KaLi gives better results compared to software only [15] as well as HW/SW co-designs [12], [14], [17] . Amongst all the hardware-only designs [7]–[11] for Kyber we report the highest clock frequency. Note that the area of our design is determined by Dilithium and not by Kyber. Therefore, even though the results show that we consume a very high area, we only consume the bare minimum and give almost the best performance results on the FPGA platform.
Comparison with ASIC-based implementations: Table VI gives the comparison of implementation results for Kyber-1024 on ASIC platform. On ASIC platform, KaLi consumes the same area as reported in [16] but gives a 2.3/8.1× better performance under 65nm/28nm technology. In fact, we surpass all existing designs [11]–[13], [17] in terms of performance. However, compared to some of the designs, we use more area, and for this, we must remind again that Kyber is the recessive scheme among the two, and therefore this area is higher when compared to Kyber-only implementations.

In the section, we established that our unified cryptoprocessor KaLi transcends all the state-of-the-art works that exist in literature. Thus, showing that the proposed design methodology yields better results. In the next section, we will discuss the high-level advantages of following the design methodology and conclude the paper.

TABLE V
Comparison Table for Kyber-1024 FPGA Implementations

Ref.	Platform	Performance* (in μs)	Freq. (*MHz*)	Resources (LUT/ FF/DSP/BRAM)
[15]‡	Cortex-M4	33,850	100	-/-/-/-
[17]†	Artix-7	18,560	25	15K/3K/11/14
[12]†	Zynq	-	-	24K/11K/21/32
[14]†	Artix-7	85,559	59	2K/2K/5/34
[7]	Virtex-7	1,260	192	133K/-/548/202
[8]	Artix-7	154	161	7K/5K/2/3
[9]	Artix-7	63	210	12K/12K/8/15
[10]	Artix-7	56	185	13K/12K/16/16
[11]	Artix-7	286	112	16K/6K/12/17
[11]	Virtex-7	205	156	16K/6K/12/17
KaLi^a	**US+**	**93**	**270**	**23K/9K/4/24**

*:Performance is measured as total time for encapsulation and decapsulation (key generation can be done offline). ^{a}:Supports multiple schemes. †: HW/SW co-design. ‡: SW design.

TABLE VI
Comparison Table for Kyber-1024 ASIC Implementations

Ref.	Tech. (*nm*)	Perf.* (in μs)	Freq. (*MHz*)	Logic Gates (*KGE*)	SRAM (*KB*)
[17]†	40	6,444	72	106	40.25
[12]†	65	18,444	45	170	465
[13]†	28	727	300	979	12
[11]	65	160	200	104	190
[16]^{†,a}	28	206	540	697	24.75
[16]^a	28	22/17.7^b	540	623	36.75
KaLi^a	**65**	**90.2**	**280&560**	**769**	**34.82**
KaLi^a	**28**	**25.26**	**1000&2000**	**747**	**34.82**

*:Performance is measured as total time for encapsulation and decapsulation (key generation can be done offline). †: HW/SW co-design. ^{a}:Supports multiple schemes. ^{b}:Depending on the type of scheduling.

V. Discussion and Conclusion

The post-quantum key encapsulation and digital signature schemes are required for secure communication. A unified architecture for these two types of schemes will help make the design compact. This architecture should also be agile for future transitions. In this paper, we present a design methodology for the implementation of one such unified architecture. To this end, we designed and implemented the first unified architecture KaLi that can perform all the operations for all the security levels of Dilithium and Kyber.

To give a good comparison, KaLi is realized on both FPGA and ASIC platforms. Special optimizations are done to make the design compact for ASIC platforms using multiple clock domains. KaLi outperforms all the existing implementations and is a strong step towards compact and agile designs. The proposed design methodology can be easily customized depending on the constraints and requirements.

References

[1] F. Arute, K. Arya, R. Babbush, D. Bacon, J. C. Bardin, R. Barends, R. Biswas, S. Boixo, and many more, "Quantum supremacy using a programmable superconducting processor," Nature, 2019, https://doi.org/10.1038/s41586-019-1666-5.

[2] S. S. Roy and A. Basso, "High-speed instruction-set coprocessor for lattice-based key encapsulation mechanism: Saber in hardware," *IACR Trans. Cryptogr. Hardw. Embed. Syst.*, vol. 2020, no. 4, pp. 443–466, 2020. [Online]. Available: https://doi.org/10.13154/tches.v2020.i4.443-466

[3] Z. Zhou, D. He, Z. Liu, M. Luo, and K.-K. R. Choo, "A software/hardware co-design of crystals-dilithium signature scheme," *ACM Trans. Reconfigurable Technol. Syst.*, vol. 14, no. 2, Jun. 2021. [Online]. Available: https://doi.org/10.1145/3447812

[4] S. Ricci, L. Malina, P. Jedlicka, D. Smékal, J. Hajny, P. Cibik, P. Dzurenda, and P. Dobias, "Implementing crystals-dilithium signature scheme on fpgas," in *The 16th International Conference on Availability, Reliability and Security*, ser. ARES 2021. New York, NY, USA: Association for Computing Machinery, 2021. [Online]. Available: https://doi.org/10.1145/3465481.3465756

[5] G. Land, P. Sasdrich, and T. Güneysu, "A hard crystal - implementing dilithium on reconfigurable hardware," *IACR Cryptol. ePrint Arch.*, vol. 2021, p. 355, 2021. [Online]. Available: https://eprint.iacr.org/2021/355

[6] L. Beckwith, D. T. Nguyen, and K. Gaj, "High-performance hardware implementation of crystals-dilithium," Crypto. ePrint Arch., Report 2021/1451, 2021, https://ia.cr/2021/1451.

[7] Y. Huang, M. Huang, Z. Lei, and J. Wu, "A pure hardware implementation of CRYSTALS-KYBER PQC algorithm through resource reuse," *IEICE Electron. Express*, vol. 17, no. 17, p. 20200234, 2020. [Online]. Available: https://doi.org/10.1587/elex.17.20200234

[8] Y. Xing and S. Li, "A compact hardware implementation of cca-secure key exchange mechanism CRYSTALS-KYBER on FPGA," *IACR Trans. Cryptogr. Hardw. Embed. Syst.*, vol. 2021, no. 2, pp. 328–356, 2021. [Online]. Available: https://doi.org/10.46586/tches.v2021.i2.328-356

[9] V. B. Dang, F. Farahmand, M. Andrzejczak, K. Mohajerani, D. T. Nguyen, and K. Gaj, "Implementation and benchmarking of round 2 candidates in the NIST post-quantum cryptography standardization process using hardware and software/hardware co-design approaches," *IACR Cryptol. ePrint Arch.*, p. 795, 2020. [Online]. Available: https://eprint.iacr.org/2020/795

[10] M. Bisheh-Niasar, R. Azarderakhsh, and M. M. Kermani, "High-speed ntt-based polynomial multiplication accelerator for crystals-kyber post-quantum cryptography," *IACR Cryptol. ePrint Arch.*, p. 563, 2021. [Online]. Available: https://eprint.iacr.org/2021/563

[11] M. Bisheh-Niasar, R. Azarderakhsh, and M. M. Kermani, "Instruction-set accelerated implementation of crystals-kyber," *IEEE Trans. Circuits Syst. I Regul. Pap.*, vol. 68, no. 11, pp. 4648–4659, 2021. [Online]. Available: https://doi.org/10.1109/TCSI.2021.3106639

[12] T. Fritzmann, G. Sigl, and J. Sepúlveda, "RISQ-V: tightly coupled RISC-V accelerators for post-quantum cryptography," *IACR Trans. Cryptogr. Hardw. Embed. Syst.*, vol. 2020, no. 4, pp. 239–280, 2020. [Online]. Available: https://doi.org/10.13154/tches.v2020.i4.239-280

[13] G. Xin, J. Han, T. Yin, Y. Zhou, J. Yang, X. Cheng, and X. Zeng, "VPQC: A domain-specific vector processor for post-quantum cryptography based on RISC-V architecture," *IEEE Trans. Circuits Syst. I Regul. Pap.*, vol. 67-I, no. 8, pp. 2672–2684, 2020. [Online]. Available: https://doi.org/10.1109/TCSI.2020.2983185

[14] E. Alkim, H. Evkan, N. Lahr, R. Niederhagen, and R. Petri, "ISA extensions for finite field arithmetic accelerating kyber and newhope on RISC-V," *IACR Trans. Cryptogr. Hardw. Embed. Syst.*, vol. 2020, no. 3, pp. 219–242, 2020. [Online]. Available: https://doi.org/10.13154/tches.v2020.i3.219-242

[15] L. Botros, M. J. Kannwischer, and P. Schwabe, "Memory-efficient high-speed implementation of kyber on cortex-m4," in *Progress in Cryptology - AFRICACRYPT 2019 - 11th International Conference on Cryptology in Africa, Rabat, Morocco, July 9-11, 2019, Proceedings*, ser. Lecture Notes in Computer Science, J. Buchmann, A. Nitaj, and T. Rachidi, Eds., vol. 11627. Springer, 2019, pp. 209–228. [Online]. Available: https://doi.org/10.1007/978-3-030-23696-0_11

[16] Y. Zhao, R. Xie, G. Xin, and J. Han, "A high-performance domain-specific processor with matrix extension of RISC-V for module-lwe applications," *IEEE Trans. Circuits Syst. I Regul. Pap.*, vol. 69, no. 7, pp. 2871–2884, 2022. [Online]. Available: https://doi.org/10.1109/TCSI.2022.3162593

[17] U. Banerjee, T. S. Ukyab, and A. P. Chandrakasan, "Sapphire: A configurable crypto-processor for post-quantum lattice-based protocols (extended version)," *IACR Cryptol. ePrint Arch.*, p. 1140, 2019. [Online]. Available: https://eprint.iacr.org/2019/1140

[18] S. Bai, L. Ducas, E. Kiltz, T. Lepoint, V. Lyubashevsky, P. Schwabe, G. Seiler, and D. Stehlé, "CRYSTALS-Dilithium," Proposal to NIST PQC Standardization, Round3, 2021, https://csrc.nist.gov/Projects/post-quantum-cryptography/round-3-submissions.

[19] P. Schwabe, R. Avanzi, J. Bos, L. Ducas, E. Kiltz, T. Lepoint, V. Lyubashevsky, J. M. Schanck, G. Seiler, and D. Stehle, "CRYSTALS-KYBER," Proposal to NIST PQC Standardization, Round3, 2021, https://csrc.nist.gov/Projects/post-quantum-cryptography/round-3-submissions.

[20] D. Sprenkels, "The Kyber/Dilithium NTT," https://dsprenkels.com/ntt.html.

[21] M. Scott, "A note on the implementation of the number theoretic transform," in *Cryptography and Coding - 16th IMA International Conference, IMACC 2017, Oxford, UK, December 12-14, 2017, Proceedings*, ser. Lecture Notes in Computer Science, M. O'Neill, Ed., vol. 10655. Springer, 2017, pp. 247–258. [Online]. Available: https://doi.org/10.1007/978-3-319-71045-7_13

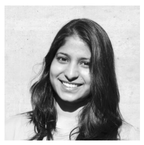
[22] V. Lyubashevsky and G. Seiler, "NTTRU: Truly Fast NTRU Using NTT," *IACR Trans. on CHES*, vol. 2019, no. 3, pp. 180–201, May 2019.

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

[24] Aikata, A. C. Mert, D. Jacquemin, A. Das, D. Matthews, S. Ghosh, and S. S. Roy, "A unified cryptoprocessor for lattice-based signature and key-exchange," Cryptology ePrint Archive, Report 2021/1461, 2021, https://ia.cr/2021/1461.

[25] Y. Xing and S. Li, "A compact hardware implementation of cca-secure key exchange mechanism crystals-kyber on fpga," *IACR Transactions on Cryptographic Hardware and Embedded Systems*, pp. 328–356, 2021.

[26] 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. Cryptogr. Hardw. Embed. Syst.*, vol. 2022, no. 1, pp. 270–295, 2022.



Aikata obtained her Bachelors in Technology degree from IIT Bhilai, India, in 2020 and Masters degree from Graz University of Technology, Austria, in 2022. She is currently a PhD student at Institute of Applied Information Processing and Communications, Graz University of Technology. Her research interests include lattice-based cryptography and hardware design.

Ahmet Can Mert received his PhD degree in electronics engineering from Sabanci University, Turkey in 2021. Currently, he is working as a postdoctoral researcher at the Institute of Applied Information Processing and Communications, Graz University of Technology, Austria. His research interest include homomorphic encryption, lattice-based cryptography and hardware design.

Malik Imran received his bachelor's and master's degrees from Pakistan in 2011 and 2015, respectively. Now, he is in with the Center for Hardware Security, Tallinn University of Technology (TalTech), Tallinn, Estonia, as a doctoral student. Before joining TalTech, Malik contributed to different research labs for efficient hardware accelerators for intrusion detection systems and asymmetric cryptography.

Samuel Pagliarini (M'14) received the PhD degree from Telecom ParisTech, Paris, France, in 2013. He has held research positions with the University of Bristol, Bristol, UK, and with Carnegie Mellon University, Pittsburgh, PA, USA. He is currently a Professor with Tallinn University of Technology (TalTech) in Tallinn, Estonia where he leads the Centre for Hardware Security.

Sujoy Sinha Roy is an Assistant Professor of cryptographic engineering at IAIK, Graz University of Technology. He is a Co-Designer of "Saber," which is a finalist key encapsulation mechanism (KEM) candidate in NIST's Post-Quantum Cryptography Standardization Project. He is interested in the implementation aspects of cryptography.