A Unified Cryptoprocessor for Lattice-based Signature and Key-exchange

Aikata*, Ahmet Can Mert*, David Jacquemin*, Amitabh Das†, Donald Matthews†, Santosh Ghosh‡, Sujoy Sinha Roy*
*Graz University of Technology, Graz, Austria †AMD, Austin, Texas, US ‡Intel Labs, Intel Corporation, OR, US
*{aikata, ahmet.mert, david.jacquemin, sujoy.sinharoy}@iaik.tugraz.at
†{Donald.Matthews, Amitabh.Das}@amd.com ‡santosh.ghosh@intel.com

ABSTRACT

We propose a compact, unified and instruction-set cryptoprocessor for performing both lattice-based digital signature (Crystals-Dilithium) and key exchange (Saber). The implementation leverages from algorithmic and structural synergies in the two schemes to realize a unified high-speed post-quantum key-exchange and digital signature engine within a compact area. On a Xilinx Ultrascale+ FPGA, the cryptoprocessor consumes 19,140 LUTs, 9,351 FFs, 4 DSPs, and 24 BRAMs. It meets 200 MHz clock frequency and finishes CCA-secure key-generation, encapsulation, and decapsulation operations for Saber in 54.9, 69.7, and 94.9μs, respectively. For Dilithium-3, key-generation, signing, and verification take 114.7, 237 and 127.6μs, respectively, for the best-case scenario.

KEYWORDS

Dilithium, Saber, Hardware Implementation, Lattice-based Cryptography, Post-quantum cryptography

1 INTRODUCTION

Shor's quantum algorithm solves the integer factorization and discrete logarithm problems using quantum computers in polynomial time. These number theoretic problems are the foundations of the two most widely used public-key cryptosystems, namely the RSA and Elliptic Curve cryptosystems. Hence, if a sufficiently powerful quantum computer is ever constructed, then the present-day public-key cryptographic schemes can be broken using Shor's algorithm. Post-quantum cryptography aims at developing new cryptographic protocols that will remain secure even after the quantum computers are built. National Institute of Standards and Technology (NIST) initiated a project 'Post-Quantum Cryptography (PQC) Standardization' in 2016 to develop and standardize post-quantum public-key cryptography algorithms. After the first two rounds, NIST initiated the final round in July 2020 and announced the finalists. There are four finalists (including lattice-based Saber [8]) in the key encapsulation mechanism (KEM) category and three finalists (including lattice-based Crystals-Dilithium [2]) in the digital signature category. NIST encouraged more research on improving the implementation and physical security aspects of all the finalist and alternate candidates. Making post-quantum cryptography ready for deployment on a wide range of platforms is a challenging task. Hence, significant research on the implementation aspects of post-quantum cryptography is needed to make it practical, efficient, and secure on a wide range of platforms.

Our Contribution: In this paper, we propose a compact and fast cryptoprocessor architecture for performing both lattice-based signature and key-exchange operations. We realized this unified cryptoprocessor architecture by exploring synergies in the lattice-based finalist PKE/KEM candidate Saber [8] and the signature candidate Dilithium [2]. In detail, we make the following contributions:

- As a first step, we identify several algorithmic and structural synergies in Saber and Dilithium. Polynomial multiplication is a central, time- and area-consuming arithmetic operation in both schemes. While Dilithium [2] uses a prime modulus and Number Theoretic Transform (NTT)-based polynomial multiplication, Saber [8] uses power-of-two moduli and keeps the choice of a polynomial multiplication algorithm open to implementers. To design a unified cryptoprocessor for the two schemes, we use the NTT-based polynomial multiplication method for Saber too and design a common NTT multiplier in minimum area overhead.
- Both Saber and Dilithium make use of Keccak-based pseudorandom number generations and hash calculations. However, as the two schemes use different parameter sets, pre- and post-processing of the data at the input and output of the Keccak function happen in different ways. To make our cryptoprocessor compact and at the same time fast, we implement an optimized wrapper around the Keccak block for performing scheme-specific processing of data on-the-fly. This reduces both area and cycle counts significantly.
- We realize a programmable (thus flexible) instruction-set cryptoprocessor to perform Saber KEM and Dilithium signature. Furthermore, the cryptoprocessor is capable of executing several data-independent instructions in parallel, thus overcoming a major shortcoming of ISA-based sequential post-quantum cryptoprocessors.

2 PRELIMINARIES

This section gives the specifications of Saber and Dilithium. Saber [8] is an IND-CCA secure KEM and its security relies on the hardness of the Module Learning With Rounding (MLWR) problem. It has three variants: LightSaber, Saber, and FireSaber targeting different security levels. All of these variants use the same polynomial rings $R_q = \mathbb{Z}_q[x]/\langle x^{256}+1\rangle$ and $R_p = \mathbb{Z}_p[x]/\langle x^{256}+1\rangle$ with the power-of-two moduli $q = 2^{13}$ and $p = 2^{10}$. The three variants use different module-dimensions and secret-distributions. Dilithium [2] is a digital signature scheme and its security is based on the computational hardness of the Module Learning With Errors (MLWE) and Module Short Integer Solution (MSIS) problems. Depending on the size of the module $R_q^{k\times\ell}$ with $k, \ell > 1$, Dilithium also comes with three variants, namely Dilithium-2, 3, and 5 for the NIST-specified security levels 2, 3, and 5 respectively [2]. All the three variants of Dilithium use the polynomial ring $R_q = \mathbb{Z}_q[x]/\langle x^{256}+1\rangle$ with $q = 2^{23} - 2^{13} - 1$, a prime modulus.

2.1 Saber modules

- `gen()`: It expands a uniform seed $\rho \in \{0,1\}^{256}$ using the Keccak-based expandable output function (XOF) SHAKE-128 and generates the public matrix $\boldsymbol{A} \in R_q^{l\times l}$.
- $\beta_\mu()$: It samples a secret polynomial vector ($\boldsymbol{s}$) from a binomial distribution with the parameter μ.
- Hash functions: Saber uses three hash functions: $\mathcal{F}()$, $\mathcal{H}()$ and $\mathcal{G}()$. The $\mathcal{F}()$ and $\mathcal{H}()$ are implemented using SHA3-256 while $\mathcal{G}()$ is implemented using SHA3-512. All hash functions are Keccak-based.
- Polynomial arithmetic: They include polynomial multiplication, polynomial addition/subtraction, coefficient-wise rounding using bit-shifting, equality checking of two polynomials, etc.

2.2 Dilithium modules

- `ExpandA()`: This function uses SHAKE-128 to generate the polynomials of the public matrix $\boldsymbol{A} \in R_q^{k\times\ell}$ in parallel by expanding the common seed $\rho \in \{0,1\}^{256}$ along with different 16-bit nonce values.
- `ExpandS()`: It is used to generate the secret polynomial vectors $\boldsymbol{s}_1$ and $\boldsymbol{s}_2 \in S_\eta^\ell \times S_\eta^k$. For each polynomial the seed ς and a 16-bit nonce are fed to SHAKE-256 and the squeezed output is given to the rejection sampler for sampling the signed values in the range $\{-\eta, \eta\}$.
- `Power2Round`$_q()$: This function 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 $r_1 = (r - r_0)/2^d$.
- `HighBits`$_q()$ and `LowBits`$_q()$: Let α be a divisor of $q-1$. The function `Decompose`$_q()$ is defined in the same way as `Power2Round()` with α replacing 2^d in `Power2Round()`.
- `MakeHint`$_q()$: It uses `Decompose`$_q()$ to produce a hint $\boldsymbol{h}$.
- `UseHint`$_q()$: It use the hint $\boldsymbol{h}$ produced by `MakeHint`$_q()$ to recover the high-bits.
- `CRH()`: This is a collision resistant hash function which utilizes 384 bits of the output of SHAKE-256.
- `SampleInBall()`: It fills a polynomial with only τ coefficients set to +1 or −1 and the remaining coefficients as 0.
- `ExpandMask()`: This function expands ($\acute{\rho} \parallel \kappa$) string to generate a polynomial vector. The SHAKE output is broken into a sequence of positive integers in the range $[0, 2\gamma_1 - 1]$ and these are processed using a rejection sampling.
- Polynomial Arithmetic and `NTT()`: Polynomial multiplications are performed using the NTT method.

The signing operation generates a potential signature and checks a set of constraints on the generated signature. If satisfied, a valid signature is produced as the output; otherwise, the loop continues with generating another potential signature.

3 SYNERGIES AND DESIGN DECISIONS

Both Saber and Dilithium are based on module lattices and therefore they share structural similarities to some extent. For example, both schemes operate on matrices and vectors of polynomials where the polynomials are always of 256 coefficients. Hence, the underlying elementary polynomial arithmetic operators are common to Saber and Dilithium. Furthermore, both schemes use Keccak-based hash functions and pseudorandom number generators. Note that in module lattice-based public-key schemes, polynomial multiplications and pseudorandom number generations are the most expensive operations as shown in [14].

3.1 Polynomial multiplication & Hash functions

When implementing a unified cryptoprocessor for both Dilithium and Saber, we have two options. The first option is to instantiate an NTT-based multiplier for Dilithium (with prime modulo) and a schoolbook or Toom-Cook multiplier for Saber (with power-of-two modulo) so that both schemes can be executed at their optimal speeds. This approach requires a large area in hardware and could potentially slowdown the clock frequency of the implementation. The other option will be to instantiate a common polynomial multiplier for both schemes. In this case, the common multiplier must be NTT-based as the Dilithium protocol makes the use of NTT an integral part of the protocol, and an implementation of Saber [8] could use any type of polynomial multiplication algorithm.

Keccak-based SHA3 and SHAKE are used in both Dilithium and Saber. Hence, a common Keccak core along with a wrapper around it is used to implement all different SHA3 and SHAKE functionalities needed by the two schemes.

3.2 Remaining scheme-specific building blocks

The remaining building blocks in the two schemes do not share many similarities. They mostly perform simple operations (i.e., additions, packing) of linear time-complexity. To reduce the area consumption further an option could be to resource-share a common set of arithmetic circuits (e.g., addition and subtraction) with algorithm-specific finite state machines for generating the control signals. This design-decision might decrease the area and at the same time might make the overall design complex and serial instead of parallel. Therefore, to make the design simple and easily configurable, we decide to keep the scheme-specific blocks separate in the implementation.

4 OPTIMIZED IMPLEMENTATION

The proposed unified cryptoprocessor for Saber and Dilithium has an NTT-based polynomial multiplier, a Keccak-core (with a wrapper around it), and several scheme-specific building blocks. The first two are the most expensive in terms of both computation time and area requirements, and thus they must be well optimized in our unified cryptoprocessor.

4.1 NTT-based unified polynomial multiplier

This section describes the implementation decisions we make for designing the NTT-based polynomial multiplier for both Saber and

Dilithium. To correctly perform NTT-based polynomial multiplications in Saber, we need to use a sufficiently large prime p' so that no true modular reductions happen [5].

Prime selection for NTT in Saber: Saber's secret polynomial coefficients are signed values in the range [-3,3], [-4,4], and [-5,5] depending on the security level. When all the coefficients are positive, a prime modulus close to $2^3 \times 2^{13} \times 256 = 2^{24}$ is sufficient to prevent any true modular reduction by $q = 2^{13}$. If negative coefficients are turned into unsigned coefficients by adding q to each of them, then the required modulus size increases to $2^{13} \times 2^{13} \times 2^8 = 2^{34}$. In [5, 9], the authors discuss a similar problem and mention that a 24-bit modulus can be used along with special provision for signed number representation.

In Saber, matrix-vector and vector-vector multiplications perform accumulation of polynomial multiplication results. If a 25-bit prime, such as $2^{25} - 2^{14} + 1$, is used as the NTT-modulus in Saber, then there no true modular reduction will ever happen and thus the result will always be correct. Experimentally we observe that if slightly smaller primes are used, then true modular reductions happen with very low probabilities. For example, with the 24-bit prime $2^{24} - 2^{14} + 1$ and the 23-bit prime of Dilithium, a true modular reduction happens with the probabilities $\approx 2^{-100}$ and $\approx 2^{-350}$ respectively.

Using Dilithium's prime for the NTTs of Saber gives us the special advantage that we need just one 23-bit NTT unit for both schemes, thus making the implementation really compact. For the other two primes, data-path extensions become essential.

Efficient modular reduction unit and Post-processing elimination: If the two primes have similar structures, then their modular reduction circuits can be unified very well to reduce the area overhead. Therefore, after carefully choosing sparse and reduction-friendly primes for Saber, we followed the add-shift-based method [22] and used a similar fast modular reduction technique. We use $2^{24} \equiv 2^{14}-1 \pmod q / 2^{25} \equiv 2^{14} - 1 \pmod q$ or $2^{23} \equiv 2^{13} - 1 \pmod q$ recursively, generate six partial results and add them to perform modular reduction. Finally, a correction is performed to bring the result to the range $\{0, \ldots, q-1\}$. Fig. 1 shows the modular reduction unit which uses a carry-save adder tree to reduce the critical path. In the Inverse NTT (INTT) operation the resulting coefficients are scaled by $1/n$, which requires extra n multiplications. In our design, these extra scaling is removed by processing the coefficients using the equation $x/2 \mod q = (x \gg 1) + (x \,\&\, \texttt{0x1}) \times ((q+1)/2)$ during the INTT [22], where x is the output of the butterfly operation during INTT. This way both the NTT and INTT are of the same cost and require no post-processing.

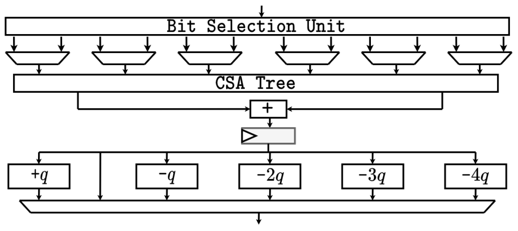

Figure 1: Unified modular reduction unit

Algorithm 1: The Cooley-Tukey NTT algorithm [20]

```
Input  : A vector $x = [x_0, \cdots, x_n - 1]$ where $x_i \in \mathbb{Z}_p, n, q \in \mathbb{Z}_q$
Input  : Table of $2n^{th}$ roots of unity $g$, in bit reversed order
Output : $\hat{x} \leftarrow NTT(x), x_i \in \mathbb{Z}_q$, in bit-reversed order
1  $t, m \leftarrow (n/2), 1$;
2  while $(m < n)$ do
3      $k \leftarrow 0$;
4      for $(i = 0; i < m; i = i + 1)$ do
5          for $(j = k; j < (k + l); j = j + 1)$ do
6              $V \leftarrow x[j + t] \times g[m + i] \pmod q$;
7              $x[j + t] \leftarrow x[j] - V \pmod q$;
8              $x[j] \leftarrow x[j] + V \pmod q$;
9          $k \leftarrow k + 2t$;
10     $t, m \leftarrow t/2, 2m$;
11 return $x$
```

Internal architecture of NTT: Following the official reference code of Dilithium, we use the Cooley-Tukey (Alg. 1) and Gentleman-Sande butterfly configurations for the NTT and INTT respectively. Both butterfly configurations are implemented in a unified butterfly core. The circuits are all pipelined to achieve high clock frequency.

As one butterfly core consumes two coefficients and simultaneously produces two coefficients every cycle, we always keep two coefficients in a single memory-word following [19]. This enables accessing two coefficients by just one memory-read and storing two coefficients by just one memory-write. Our NTT unit has two such butterfly cores in parallel to reduce the cycle count of NTT. To feed the two butterfly cores, we spread the coefficients into two BRAM sets. In this way, a polynomial of 256 coefficients occupies a total of 128 memory words of which 64 are in the first BRAM set and the remaining 64 are in the other BRAM set. When the two butterfly cores load the j^{th} and $(j + l/2)^{th}$ coefficients, they also get the $(j+1)^{th}$ and $(j + l/2 + 1)^{th}$ coefficients automatically. One NTT or INTT operations take 512 clock cycles only. During the NTT loops, the newly generated coefficients are written back in the BRAMs in such as way that during the next iteration of the NTT loop, the required coefficients for each butterfly can be read as a pair from the memory.

4.2 SHA3-256/512 and SHAKE-128/256

For implementing the Keccak-based hash and expandable output functions, we instantiate a single high-speed Keccak core in the proposed cryptoprocessor architecture. Implementation of the Keccak core is similar to the high-speed Keccak core available on the website of Keccak-team [21]. We use a wrapper module around the Keccak core to perform parsing of input and output data bits. Additionally, the state buffer has been changed so that the pseudo-random polynomial coefficients can be generated in scheme-specific optimal representations and then stored immediately in the memory of the cryptoprocessor. This strategy helps reduce the overall cycle counts for both Dilithium and Saber.

Saber's public polynomials, generated using SHAKE-128, have a 13-bit coefficient size. Before these polynomials are multiplied, they are converted into the NTT representation in our unified cryptoprocessor. As described in Sec. 4.1, the NTT unit requires its operand data to be present in 'two coefficients per BRAM word' format, for reading and writing the coefficients efficiently. One option for processing the public polynomials will be to generate a continuous bitstream in 64-bit words (which is the default output format of Keccak), then write them in BRAMs, and later parse them into 13-bit coefficients using separate parser hardware. This approach is sequential by nature and results in a bloated cycle count and area consumption due to required buffers. To avoid such a redundant memory read/write step, we modify the output buffer of Keccak to directly produce a pair of 13-bit coefficients during the generation of the public matrix $\boldsymbol{A}$. However, this strategy requires a book-keeping mechanism as the output length of a SHAKE-128 squeeze operation is 1,344 bits which is not a multiple of 13. Therefore, after each squeeze of SHAKE-128, there will be leftover bits that must be prepended to the output string generated by the next SHAKE-128 squeeze operation. We observe that during the generation of $\boldsymbol{A}$ in Saber, the number of leftover bits is always an even number in [0, 24]. We use this observation to simplify the implementation of the Keccak-output buffer.

The prepending of the leftover bits to a newly generated SHAKE-128 squeeze output requires shifting and filling of the buffer bits. As the size of the Keccak output buffer (when operated as SHAKE-128) is 1,344 bits which is quite large, we investigated efficient implementation techniques that reduce the area-overhead without affecting the cycle count. The first and very naive method that comes to our mind is to implement a simple multiplexer that assigns the output buffer with 1344 bits of the Keccak state and the leftover bits. But since there can be 13 (even numbers in [0, 24]) such possibilities we will require a 12-to-1 multiplexer for assigning to a buffer of size 1,368 (=1344+24) bits. With this implementation option, there are 13 shift possibilities and as a consequence, the multiplexing overhead is ≈8000 LUTs, which is large. We aim to make a very efficient and lightweight design on hardware, therefore we need a much better solution.

The leftover bits are handled using a small 'left-over-bits buffer'. The content of this left-over-bits buffer is then concatenated at the beginning of the output buffer. We decide to just make three intermediate buffers for zero, two, and four shifts, for both the output buffer and left-over-bits buffer, as shown in Fig. 2. After the Keccak squeeze is done we write the remaining bits to the left-over-bits buffer. To avoid using a multiplexer to decide on the number of remaining bits we need to pick, we just write the 24 bits as the remaining bits. Then based on the count of remaining bits we shift the left-over-bits buffer by four or two bits towards the left. Once the left-over-bits buffer is aligned, we start shifting both the output buffer and left-over-bits buffer towards the left by four or two. The values pushed out by the left-over-bits buffer are put in front of the output buffer. Since we run Keccak in parallel with NTT, the extra cycle count for this bookkeeping does not account for an increase in the total cycle count.

4.3 Samplers

Saber uses a binomial sampler as described in Sec. 2.1 and it needs to read 13-bit from the Keccak output buffer. Dilithium requires three different kinds of rejection sampling units for a coefficient generation: uniform, η, and γ sampling. For the uniform and η sampling, we need to extract 24 and 4 bits from the Keccak output

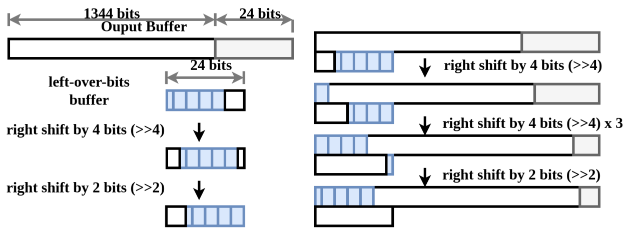
1344 bits
24 bits
Ouput Buffer
24 bits
left-over-bits buffer
right shift by 4 bits (>>4)
right shift by 2 bits (>>2)
right shift by 4 bits (>>4)
right shift by 4 bits (>>4) x 3
right shift by 2 bits (>>2)

Figure 2: Example of book-keeping with 18 remaining bits

buffer, respectively, and we can utilize Keccak output fully after every squeeze. On the other hand, the γ sampling needs 18 or 20 bits from the Keccak output and it does not utilize the Keccak output fully after every squeeze with some leftover bits. The same approach of shifting the output buffer and leftover bit buffer as described in the previous section can be used here as well. However, this leads to a Keccak output buffer outputting six different types of outputs 4, 13, 18, 20, 24, and 64 bits. This can be controlled using a multiplexer, which in hardware is very expensive. In order to reduce the cost, we take an intermediate smaller buffer of size 192 bits (=lcm(4, 24, 64)) and use it for squeezing the results for 4, 24, and 64 bits. The Keccak output buffer then outputs only four different types of outputs 13, 18, 20, and 192 bits. Thus saving around ≈1200 LUTs.

4.4 Memory

For the Dilithium variant with the NIST security level 5, the public matrix $\boldsymbol{A}$ has dimensions (8×7). During sign operation, we need to precompute and store the secret vectors $\boldsymbol{s}_1^{7\times1}$, $\boldsymbol{s}_2^{8\times1}$, and $\boldsymbol{t}_0^{8\times1}$ in the NTT domain, thus requiring storage for at least 23 polynomials. Storing the entire public matrix $\boldsymbol{A}$ in the memory makes the signing operation faster as $\boldsymbol{A}$ is used in the loop several times. If we pre-compute and store all polynomials, we require to store 79 polynomials before the signing loop starts along with seeds and hash values. Also, we need to store intermediate results during the signing operation, which further increases the memory requirement. In the proposed work, we use the ability of our cryptoprocessor to process data-independent instructions in parallel. Instead of generating and storing the public matrix at once, we generate it on-the-fly with parallel to the polynomial multiplication operation, thus reducing more than half the memory requirement without compromising the performance. Since Dilithium-5 has a much larger public matrix and secret vector than Saber with security level 5 (4×4), the overall memory requirement of the cryptoprocessor is determined by Dilithium. With all the constraints in consideration and flexibility requirements in place, the implementation of Saber requires only four BRAM36K units while Dilithium requires 20 BRAM36K.

The proposed cryptoprocessor uses parallel memory organization to ensure efficient load and storage of polynomials. This is especially important for the parallel execution of NTT and Keccak operations. To that end, the memory was split across four major blocks, with each of them having five BRAM36K elements, which enables the parallel execution of NTT and Keccak. The constants for NTT and inverse NTT computations are kept in a ROM which is also interpreted using one BRAMs in our implementation. Along with this, the program controller which is used to load all instructions at once and then handle all the data-independent executions in parallel requires three BRAM36K.

4.5 Parallel processing

In [7, 11], overlapping of data-independent computations at the block level is used to reduce the clock cycle counts of several lattice-based post-quantum schemes. Overlapping of computations in an instruction-set cryptoprocessor is relatively more challenging than overlapping computations in a block-unrolled architecture. In [18], all the instructions, including data-independent instructions, are executed in a series to compute the Saber protocol. In our work, we apply overlapping of data-independent computations in the context of an *instruction-set architecture* and execute data-independent Keccak-based and polynomial arithmetic-based operations in parallel. This strategy effectively reduces the overall cycle count at the cost of a negligible area overhead. To support the parallel execution of Keccak and polynomial arithmetic, we split the memory unit into four BRAM sets. While the NTT unit occupies read and write ports of any two BRAM sets, the Keccak unit works with the remaining two sets. We also add a program controller unit that loads all the instructions in an instruction RAM and then sends them one by one to the compute core for processing in parallel or sequence as specified in the instruction.

5 RESULTS

The proposed unified cryptoprocessor architecture is described entirely in Verilog and it is implemented for FPGA and ASIC platforms. For FPGA, the proposed architectures with 23-bit, 24-bit, and 25-bit primes for Saber's NTT are synthesized and implemented using Vivado 2019.1 tool suite for the target platform Zynq Ultrascale+ ZCU102 with an area-optimized implementation strategy. The FPGA implementations achieve a 200 MHz clock frequency. The implementation with only 23-bit prime uses 19,146 LUTs (6.9%), 9,338 DFFs (1.7%), 4 DSPs (0.1%) and 24 BRAMs (2.6%) only. The implementation with 24-bit Saber prime uses 19,140 LUTs (6.9%), 9,351 DFFs (1.7%), 4 DSPs (0.1%) and 24 BRAMs (2.6%) only. The implementation with 25-bit Saber prime uses 19,042 LUTs (6.9%), 9,479 DFFs (1.7%), 4 DSPs (0.1%) and 24 BRAMs (2.6%) only. The number of BRAMs in our cryptoprocessor is determined by the memory requirement of Dilithium since it is significantly more memory-consuming than Saber. The Keccak and multiplier units together consume more than half of the overall area. For the sake of simplicity, we will use the implementation with 24-bit Saber prime for the rest of this section. For ASIC, the proposed architecture is synthesized with UMC 65nm library and it achieves 400 MHz clock frequency with 0.317mm^2 area ($\approx$220 kGE) excluding on-chip memory.

In Table 1, we present the cycle count and latency (in μs) for the operations of Saber and Dilithium for different security levels in FPGA. With 200 MHz clock frequency in FPGA, the CCA-secure key generation, encapsulation, and decapsulation operations for Saber take 54.9, 69.7, and 94.9 μs, respectively. The Dilithium signature generation operation has a loop and it iterates until a valid signature is generated. In Table 1, we report the performance for the best-case scenario where the valid signature is generated after the first loop iteration. We also divide signature generation operation into three parts (pre-sign, sign, post-sign) and report their performances separately. For a signature generation, the pre-sign and post-sign parts are performed only once while the sign part is repeated until a valid signature is generated. For the best-case scenario, the key generation, signing ,and verification operations for Dilithium-3 take 114.7, 237, and 127.6 μs, respectively, in the FPGA.

Table 1: Performance results for Saber-KEM and Dilithium

Operation	LS/D-2		S/D-3		FS/D-5	
	Cycle	Lat.	Cycle	Lat.	Cycle	Lat.
Sab.Keygen	5,935	29.6	10,980	54.9	17,523	87.6
Sab.Encaps	8,081	40.4	13,941	69.7	21,603	108.0
Sab.Decaps	11,678	58.3	18,991	94.9	27,890	139.4
Dil.Gen	14,183	70.9	22,957	114.7	38,841	194.2
Dil.Sign$_{pre}$	7,554	37.7	9,273	46.3	12,448	62.2
Dil.Sign	21,115	105.5	35,865	179.3	52,955	264.7
Dil.Sign$_{post}$	1,689	8.4	2,280	11.4	3,057	15.2
Dil.Verify	15,044	75.2	25,535	127.6	45,789	228.9

5.1 Comparison with the existing results

The proposed cryptoprocessor is compared with related works in the literature in terms of area, performance, and flexibility for Saber and Dilithium-3 as shown in Table 2 and Table 3, respectively. In the literature, only a few works are targeting a unified architecture that supports multiple PQC schemes [3, 9, 10]. In [3], the authors present *Sapphire*, a cryptoprocessor coupled with RISC-V processor implemented in ASIC for various lattice-based Round 2 schemes in NIST's PQC standardization. It does not support or provide performance results for Saber while the results provided for Dilithium are using Round-2 specifications. In [9], the authors present a RISC-V architecture coupled with optimized hardware accelerators. It provides support for Crystals-Kyber, NewHope, and Saber schemes and targets the ASIC platform. Compared to the Saber implementation in [9], our FPGA and ASIC implementations show up to 304× and 564× better performance, respectively. The work in [10] presents a HW/SW co-design of Crystals-Kyber and Saber schemes. Our implementation shows superior performance in terms of both performance and area consumption as we target an implementation entirely in hardware.

There are several works in the literature implementing Saber in hardware for FPGA [1, 6, 12, 16, 18] and ASIC [13, 24] platforms. Our unified cryptoprocessor outperforms the works in [1, 16] and shows similar performance compared to the architectures in [6, 12]. The high-performance implementations in [13, 18, 24] shows better performance than our design. However, their implementations are optimized for the Saber scheme as our work targets a compact design supporting multiple schemes.

There are few FPGA-based implementations of Dilithium [4, 15, 17, 23] in the literature. Zhou *et al.* [23] propose a HW/SW co-design solution by offloading computationally intensive operations such as SHA3/SHAKE and polynomial multiplication to the hardware while keeping the rest of the operation in the software. Although their implementation has small area, our pure-hardware solution shows almost up to two orders of magnitude better performance compared to their HW/SW co-design solution. In [17], the authors present three high-performance architectures for key generation, sign, and verification operations of Dilithium scheme targeting FPGA. Their implementations can perform key generation, sign, and verification operations in 51.9, 63.1, and 95.1 μs, respectively. Although they

Table 2: Comparison Table for Saber-KEM

Ref.	Plat.	Performance (in μs)	Freq. (*MHz*)	Area (mm^2 or LUT/FF/DSP/BRAM)
[9]^b	65nm	16K/21K/26K	45.47	0.914 mm^2
[13]	65nm	7.1/7.1/9.3	1000	0.314 mm^2
[24]	40nm	2.7/3.6/4.3	400	0.380 mm^2
[10]^{†,b}	Ar.-7	3.6K/4.9K/5.5K	62.5	20K/11K/13/36.5
[16]†	Ar.-7	3.2K/4.1K/3.8K	125	7.4K/7.3K/28/2
[1]	Ar.-7	–/467.1/527.6	100	6.7K/7.3K/32/0
[12]	US+	48.9/63.2/78.5	250	10.1K/7.7K/0/3
[6]†	US+	-/60/65	322	12.5K/11.6K/256/4
[18]	US+	21.8/26.5/32.1	250	23.6K/9.8K/0/2
Our^{a,b}	US+	54.9/69.7/94.9	200	19.1K/9.3K/4/24
	65nm	≈27.5/34.9/47.5	400	≈0.317+1.230 mm^2

^{a}:On-chip memory area is estimated as ≈1.230 mm^2.
^{b}:Supports multiple schemes. †: HW/SW co-design.

Table 3: Comparison Table for Dilithium-3

Ref.	Plat.	Performance (in μs)	Freq. (*MHz*)	Area (mm^2 or LUT/FF/DSP/BRAM)
[23]†	Zynq	-/8.8K/9.9K	100	2.6K/-/-/-
[17]^a	US+	51.9/-/-	350	54.1K/25.2K/182/15
[17]^{b,d}		-/63.1/-	333	68.4K/86.2K/965/145
[17]^c		-/-/95.1	158	61.7K/34.9K/316/18
[15]^d	Ar.-7	229/311.1/221.5	145	30.9K/11.3K/45/21
[15]^e		229/852.3/221.5		
[4]^d	US+	32/63/39	145	55.9K/28.4K/16/29
[4]^e		32/193/39		
Our^{d,f}	US+	114.7/237/127.6	200	19.1K/9.3K/4/24
	65nm	≈57.4/118.5/63.8	400	≈0.317+1.230 mm^2

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

show better performance than our implementation, their implementation for sign operation- consumes 3.5×, 9.2×, 241.2× and 6× more LUTs, DFFs, DSPs and BRAMs compared to our implementation. Moreover, our work can perform all three operations in a single implementation and it provides support for the Saber scheme as well. The work in [15] presents a Dilithium implementation for FPGA. They target reducing LUT utilization by employing extra DSP units for computations. Our implementation shows better performance and uses fewer hardware resources. In [4], a high performance Dilithium implementation is presented. It shows better performance than our implementation at the expense of 3× and 4× more LUT/DFF and DSP, respectively.

6 CONCLUSION

In this work, we designed and implemented a unified hardware architecture for the two finalists Crystals-Dilithium and Saber of the NIST PQC Standardization. The optimized cryptoprocessor architecture greatly benefits from the algorithmic and structural similarities in the two implemented cryptographic schemes. To that end, we showed that it is possible to realize a compact yet fast cryptoprocessor for performing both post-quantum KEM and digital signature on ASIC and FPGA platforms.

7 ACKNOWLEDGEMENTS

This work was supported in part by the Semiconductor Research Corporation through SRC task 3043.001.

REFERENCES

[1] Abubakr Abdulgadir, Kamyar Mohajerani, Viet Ba Dang, Jens-Peter Kaps, and Kris Gaj. 2021. A Lightweight Implementation of Saber Resistant Against Side-Channel Attacks. Cryptology ePrint Archive, Report 2021/1452.

[2] Shi Bai, Léo Ducas, Eike Kiltz, Tancrède Lepoint, Vadim Lyubashevsky, Peter Schwabe, Gregor Seiler, and Damien Stehlé. 2021. CRYSTALS-Dilithium. Proposal to NIST PQC Standardization, Round3.

[3] Utsav Banerjee, Tenzin S. Ukyab, and Anantha P. Chandrakasan. 2019. Sapphire: A Configurable Crypto-Processor for Post-Quantum Lattice-based Protocols. *IACR Trans. on CHES* 2019, 4 (2019), 17–61.

[4] Luke Beckwith, Duc Tri Nguyen, and Kris Gaj. 2021. High-Performance Hardware Implementation of CRYSTALS-Dilithium. Cryptology ePrint Archive, Report 2021/1451. https://ia.cr/2021/1451.

[5] Chi-Ming Marvin Chung, Vincent Hwang, Matthias J. Kannwischer, Gregor Seiler, Cheng-Jhih Shih, and Bo-Yin Yang. 2021. NTT Multiplication for NTT-unfriendly Rings New Speed Records for Saber and NTRU on Cortex-M4 and AVX2. *IACR Trans. on CHES* 2021, 2 (2021), 159–188.

[6] Viet B. Dang, Farnoud Farahmand, Michal Andrzejczak, and Kris Gaj. 2019. Implementing and Benchmarking Three Lattice-Based Post-Quantum Cryptography Algorithms Using Software/Hardware Codesign. In *2019 Int. Conf. on Field-Programmable Technology.* 206–214.

[7] Viet B. Dang, Farnoud Farahmand, Michal Andrzejczak, Kamyar Mohajerani, Duc Tri Nguyen, and Kris Gaj. 2020. 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.* 2020 (2020), 795.

[8] Jan-Pieter D'Anvers, Angshuman Karmakar, Sujoy Sinha Roy, Frederik Vercauteren, Jose Maria Bermudo Mera, Michiel Van Beirendonck, and Andrea Basso. 2021. SABER. Proposal to NIST PQC Standardization, Round3.

[9] Tim Fritzmann, Georg Sigl, and Johanna Sepúlveda. 2020. RISQ-V: Tightly Coupled RISC-V Accelerators for Post-Quantum Cryptography. *IACR Trans. on CHES* 2020, 4 (Aug. 2020), 239–280.

[10] Tim Fritzmann, Michiel Van Beirendonck, Debapriya Basu Roy, Patrick Karl, Thomas Schamberger, Ingrid Verbauwhede, and Georg Sigl. 2021. Masked Accelerators and Instruction Set Extensions for Post-Quantum Cryptography. *IACR Cryptol. ePrint Arch.* 2021 (2021), 479.

[11] Kris Gaj. 2020. Implementation and Benchmarking of Round 2 Candidates in the NIST Post-Quantum Cryptography Standardization Process Using FPGAs. *NIST PQC Round 3 Seminars* (October 2020).

[12] Pengzhou He, Chiou-Yng Lee, and Jiafeng Xie. [n. d.]. Compact Coprocessor for KEM Saber: Novel Scalable Matrix Originated Processing. ([n. d.]).

[13] Malik Imran, Felipe Almeida, Jaan Raik, Andrea Basso, Sujoy Sinha Roy, and Samuel Pagliarini. 2021. Design Space Exploration of SABER in 65nm ASIC. arXiv:2109.07824 [cs.CR]

[14] Matthias J. Kannwischer, Joost Rijneveld, Peter Schwabe, and Ko Stoffelen. 2019. pqm4: Testing and Benchmarking NIST PQC on ARM Cortex-M4. Cryptology ePrint Archive, Report 2019/844. https://ia.cr/2019/844.

[15] Georg Land, Pascal Sasdrich, and Tim Güneysu. 2021. A Hard Crystal - Implementing Dilithium on Reconfigurable Hardware. *IACR Cryptol. ePrint Arch.* 2021 (2021), 355. https://eprint.iacr.org/2021/355

[16] Jose Maria Bermudo Mera, Furkan Turan, Angshuman Karmakar, Sujoy Sinha Roy, and Ingrid Verbauwhede. 2020. Compact domain-specific co-processor for accelerating module lattice-based KEM. In *2020 57th ACM/IEEE DAC.* 1–6.

[17] Sara Ricci, Lukas Malina, Petr Jedlicka, David Smékal, Jan Hajny, Peter Cibik, Petr Dzurenda, and Patrik Dobias. 2021. Implementing CRYSTALS-Dilithium Signature Scheme on FPGAs. In *The 16th Int. Conf. on ARES* (Vienna, Austria). Association for Computing Machinery, New York, NY, USA, Article 1, 11 pages.

[18] Sujoy Sinha Roy and Andrea Basso. 2020. High-speed Instruction-set Coprocessor for Lattice-based Key Encapsulation Mechanism: Saber in Hardware. *IACR Trans. on CHES* 2020, 4 (2020), 443–466.

[19] Sujoy Sinha Roy, Frederik Vercauteren, Nele Mentens, Donald Donglong Chen, and Ingrid Verbauwhede. 2014. Compact Ring-LWE Cryptoprocessor. In *CHES 2014.* Springer Berlin Heidelberg, 371–391.

[20] Michael Scott. 2017. 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.* Springer, 247–258.

[21] Keccak Team. Accessed on November 2019. Keccak in VHDL: High-speed Core. https://keccak.team/hardware.html.

[22] Ferhat Yaman, Ahmet Can Mert, Erdinç Öztürk, and Erkay Savas. 2021. A Hardware Accelerator for Polynomial Multiplication Operation of CRYSTALS-KYBER PQC Scheme. In *DATE 2021, Grenoble, France, Feb. 1-5, 2021*. IEEE, 1020–1025.

[23] Zhen Zhou, Debiao He, Zhe Liu, Min Luo, and Kim-Kwang Raymond Choo. 2021. A Software/Hardware Co-Design of Crystals-Dilithium Signature Scheme. *ACM Trans. Reconfigurable Technol. Syst.* 14, 2, Article 11 (June 2021), 21 pages.

[24] Yihong Zhu, Min Zhu, Bohan Yang, Wenping Zhu, Chenchen Deng, Chen Chen, Shaojun Wei, and Leibo Liu. 2021. LWRpro: An Energy-Efficient Configurable Crypto-Processor for Module-LWR. *IEEE Trans. on CAS I: Regular Papers* 68, 3 (2021), 1146–1159. https://doi.org/10.1109/TCSI.2020.3048395