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

Abstract

In this paper, we propose a compact, unified and instruction-set cryptoprocessor architecture for performing both lattice-based digital signature and key exchange operations. As a case study, the cryptoprocessor architecture has been optimized targeting the signature scheme 'Crystals-Dilithium' and the key encapsulation mechanism 'Saber', both finalists in the NIST's post-quantum cryptography standardization project. The implementation is entirely in hardware and leverages from algorithmic as well as structural synergies in the two schemes to realize a high-speed unified post-quantum key-exchange and digital signature engine within a compact area.

The area consumption of the entire cryptoprocessor architecture is 18,040 LUTs, 9,101 flip-flops, 4 DSP units, and 14.5 BRAMs on the Xilinx Zynq Ultrascale+ ZCU102 FPGA. The FPGA implementation of the cryptoprocessor achieving 200 MHz clock frequency finishes the CCA-secure key generation, encapsulation, and decapsulation operations for Saber in 54.9, 72.5 and 94.7 μs, respectively. For Dilithium-II, the key generation, signature generation, and signature verification operations take 78.0, 164.8 and 88.5 μs, respectively, for the best-case scenario where a valid signature is generated after the first loop iteration.

The cryptoprocessor is also synthesized for ASIC with the UMC 65nm library. It achieves 370 MHz clock frequency and consumes 0.301 mm^2 area ($\approx$200.6 kGE) excluding on-chip memory. The ASIC implementation can perform the key generation, encapsulation, and decapsulation operations for Saber in 29.6, 39.2, and 51.2 μs, respectively, while it can perform the key generation, signature generation, and signature verification operations for Dilithium-II in 42.2, 89.1, and 47.8 μs, respectively.

Index Terms

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

I. Introduction

Shor's quantum algorithm solves the integer factorization and discrete logarithm problems using quantum computers in polynomial time [1]. 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.

As fruits of significant research in quantum computing engineering, small-scale quantum computers have been constructed independently by Google, IBM, and Intel, over the last five years. In October 2019, Google's 54-qubit quantum processor 'Sycamore' computed a specific task in around 200 seconds, the equivalent of which can be computed in 10,000 years using a state-of-the-art supercomputer [2]. Although the present-day quantum computers are not powerful enough to break the RSA and Elliptic curve-based public-key cryptography, giant leaps are forecast and quantum computing experts anticipate that powerful enough quantum computers can be built within the next 10 to 15 years.

Post-quantum cryptography aims at developing new cryptographic protocols that will remain secure even after the quantum computers are built. In 2016, NSA recommended a gradual transition towards post-quantum cryptography. Furthermore, 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, and invited researchers from all over the world to contribute to their standardization project. After the first two rounds, NIST initiated the final round in July 2020 and announced the finalists. The finalists in the public-key encryption (PKE) or key encapsulation mechanism (KEM) category are Classic-McEliece [3], CRYSTALS-Kyber [4], NTRU [5], and Saber [6]; and in the digital signature category are CRYSTALS-Dilithium [7], Falcon [8], and Rainbow [9]. Of these finalists, only Classic-McEliece and Rainbow are code-based and oil-vinegar-based constructions, whereas all the remaining candidates are lattice-based constructions. After the second round of the standardization project, the project report from NIST [10] mentions that it is likely that at least one of these lattice-based candidates will be chosen as a standard. Additionally, 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. In this paper, we address this challenge by realizing a compact yet high-speed cryptoprocessor for lattice-based KEM and signature.

Aikata, Ahmet Can Mert, David Jacquemin, and Sujoy Sinhar Roy are with Graz University of Technology, Graz, Austria Contact email: {aikata, ahmet.mert, david.jacquemin, sujoy.sinharoy} (at) iaik.tugraz.at

Amitabh Das, and Donald Matthews are with AMD. Austin, Texas

Santosh Ghosh is with Intel Labs, Intel Corporation, Hillsboro, OR, USA

Following the initiation of the first round of NIST's PQC standardization in November 2017, hardware implementations of some of the PQC candidates started appearing in 2018. To the best of our knowledge, there are only a few published unified cryptoprocessors that could execute more than one lattice-based PQC scheme. In [11], a configurable cryptoprocessor architecture 'Sapphire' was designed to execute multiple lattice-based PQC schemes, namely Frodo, NewHope, qTESLA, CRYSTALS-Kyber, and CRYSTALS-Dilithium. Although Saphire was able to reduce the number of core computation cycles with respect to software implementations of the above-mentioned PQC schemes on resource-constrained microcontrollers, it required a large silicon area. A major drawback of Saphire is that it does not support the latest specifications of the two finalists CRYSTALS-Kyber [4] and CRYSTALS-Dilithium [7].

In [12], a tightly-coupled RISC-V extension, known as 'RISQ-V', was proposed for providing hardware acceleration support to NewHope and CRYSTALS-Kyber (and Saber to a minor extent). Their architecture mostly focuses on accelerating the Number Theoretic Transform (NTT)-based polynomial multiplication of NewHope. As the round 2 and 3 specifications of CRYSTALS-Kyber use a different variant of NTT that requires arithmetic in a quadratic extension field, the speedup for Kyber is not as good as that for NewHope. Additionally, for Saber, due to limited hardware support, the cycle count is arguably not superior to optimized software implementations on an ARM Cortex M4 microcontroller [13]. No hardware support for the lattice-based signature schemes Dilithium or Falcon is available in RISQ-V.

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 [6] and the signature candidate Dilithium [7].

Contributions

We make the following contributions:

- As a first step, we take two NIST-PQC finalists, namely Dilithium and Saber, and identify several algorithmic and structural synergies in them. These synergies are the key to implementing an optimized and compact unified cryptoprocessor architecture for post-quantum digital signature and key encapsulation mechanism.
- Polynomial multiplication is a central arithmetic operation in both schemes. It is a time- as well area-consuming operation. While Dilithium [7] uses a prime modulus and makes the Number Theoretic Transform (NTT)-based polynomial multiplication an integral part of the scheme, Saber [6] uses power-of-two moduli and gives implementors the freedom to choose an appropriate polynomial multiplication. As we aim at designing a unified cryptoprocessor for both Dilithium and Saber, we use the NTT-based polynomial multiplication for Saber too. We compute optimized parameters for the NTT of Saber and then design a common NTT unit for Dilithium and Saber in minimum area overhead.
- Both Dilithium and Saber spend significant proportions of their overall computation times in Keccak-based pseudo-random 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 fast at the same time, we have implemented an optimized wrapper around the Keccak block to perform scheme-specific processing of data on-the-fly. Our design approach reduces both area and cycle counts significantly.
- Besides polynomial multiplication and Keccak-based operations, the two schemes use a set of additional building blocks. Although these blocks are of low-complexity, typically of $O(n)$, their implementations in hardware are scheme-specific and require bit-level manipulations. We optimize these building blocks to reduce their memory access and implement them area-optimally.
- As the two schemes consist of multiple data-independent sub-operations, we reduce the overall cycle counts by executing several of these sub-operations in parallel, when possible. With this strategy, we are able to obtain significant reductions in the number of computation cycles.

II. Preliminaries

In this section we discuss the design specifications of the two lattice based schemes: Saber and Dilithium.

A. Saber

Saber [6] is an IND-CCA secure Key Encapsulation Mechanism (KEM) which has been selected as a finalist in the NIST PQC standardization project. Its security relies on the hardness of the Module Learning With Rounding (MLWR) problem. Saber uses the MLWR problem with two moduli p and q, both powers-of-two, to construct a Chosen Ciphertext Attack (CCA) secure key encapsulation mechanism (KEM). It has three variants: LightSaber, Saber, and FireSaber targeting low, medium, and high security levels respectively. All of these variants use the same polynomial rings $R_q = Z_q[x]/\langle x^{256}+1\rangle$ and $R_p = Z_p[x]/\langle x^{256}+1\rangle$ with the power-of-two moduli $q = 2^{13}$ and $p = 2^{10}$. The three variants use different module-dimensions and secret-distributions. The module dimensions for LightSaber, Saber, and FireSaber are 2, 3, and 4 respectively; the secret coefficients for them are sampled from binomial distributions with parameters $\mu = 10$, 8, and 6 respectively. The construction of Saber follows two steps. In the first step, a Chosen Plaintext Attack resistant (i.e., IND-CPA) public-key encapsulation scheme

is built. Next, a post-quantum variant of the Fujisaki-Okamoto transform is applied on the top of the IND-CPA encryption scheme to realize an IND-CCA KEM. The IND-CPA algorithms used in Saber-PKE [6] are shown in Alg. 1, 2, and 3; and the IND-CCA algorithms used in Saber-KEM are shown in Alg. 4, 6, and 5.

Algorithm 1: `Saber.PKE.KeyGen()` [14]

$seed_{\boldsymbol{A}} \leftarrow \mathcal{U}(\{0,1\}^{256})$
$\boldsymbol{A} = \mathtt{gen}(\mathrm{seed}_{\boldsymbol{A}}) \in \mathcal{R}_q^{l \times l}$
$r = \mathcal{U}(\{0,1\}^{256})$
$\boldsymbol{s} = \beta_\mu(\mathcal{R}_q^{l \times 1}; r)$
$\boldsymbol{b} = ((\boldsymbol{A}^T \boldsymbol{s} + \boldsymbol{h}) \bmod q) \gg (\varepsilon_q - \varepsilon_p) \in \mathcal{R}_p^{l \times 1}$
return $(pk := (seed_{\boldsymbol{A}}, \boldsymbol{b}), sk := (\boldsymbol{s}))$

Algorithm 2: `Saber.PKE.Enc`$(pk = (seed_{\boldsymbol{A}}, \boldsymbol{b}), m \in R_2; r)$ [14]

$\boldsymbol{A} = \mathtt{gen}(\mathrm{seed}_{\boldsymbol{A}}) \in \mathcal{R}_q^{l \times l}$
if *r is not specified* **then**
    $r = \mathcal{U}(\{0,1\}^{256})$
$\boldsymbol{s'} = \beta_\mu(\mathcal{R}_q^{l \times 1}; r)$
$\boldsymbol{b'} = ((\boldsymbol{A}\boldsymbol{s'} + \boldsymbol{h}) \bmod q) \gg (\varepsilon_q - \varepsilon_p) \in R_p^{l \times 1}$
$v' = \boldsymbol{b}^T (\boldsymbol{s'} \bmod p) \in R_p$
$c_m = (v' + h_1 - 2^{\varepsilon_p - 1} m \bmod p) \gg (\varepsilon_p - \varepsilon_T) \in \mathcal{R}_T$
return $c := (c_m, \boldsymbol{b'})$

Algorithm 3: `Saber.PKE.Dec`$(sk = \boldsymbol{s}, c = (c_m, \boldsymbol{b'}))$ [14]

$v = \boldsymbol{b'}^T (\boldsymbol{s} \bmod p) \in R_p$
$m' = ((v - 2^{\varepsilon_p - \varepsilon_T} c_m + h_2) \bmod p) \gg (\varepsilon_p - 1) \in R_2$
return m'

Algorithm 4: `Saber.KEM.KeyGen()` [14]

$(seed_{\boldsymbol{A}}, \boldsymbol{b}, \boldsymbol{s}) = \mathtt{Saber.PKE.KeyGen}()$
$pk = (seed_{\boldsymbol{A}}, \boldsymbol{b})$
$pkh = \mathcal{F}(pk)$
$z = \mathcal{U}(\{0,1\}^{256})$
return $(pk := (seed_{\boldsymbol{A}}, \boldsymbol{b}), sk := (\boldsymbol{s}, z, pkh))$

The function `gen()` expands a uniform seed $\rho \in \{0,1\}^{256}$ using the Keccak-based expandable output function (XOF) SHAKE-128 and generates the public matrix $\boldsymbol{A} \in R_q^{k \times l}$. The CCA transforms in Alg. 4, 5, and 6 also use the Keccak-based hash functions SHA3-256 and SHA3-512.

Secret polynomials are sampled from a binomial distribution with parameter μ using a binomial sampler. To compute these binomial-distributed samples, first a μ-bit pseudorandom string is generated using SHAKE-128, and then it is split into two substrings of length $\mu/2$. Next, the Hamming weights of the two substrings are subtracted to produce a binomial-distributed sample. As a subtraction is performed in this step, the output sample can have a positive or a negative sign with equal probability.

As shown in the three IND-CPA algorithms 1, 2, and 3, polynomial multiplications are performed several times. That makes polynomial multiplication a performance-critical building block.

The algorithms also use other less-complicated operations, such as polynomial addition/subtraction, coefficient-wise rounding using bit-shifting, equality check of two polynomials, pack/unpacking of polynomial-coefficients into/from byte strings, etc. These operations are of linear time complexity.

B. Dilithium

Dilithium [7] is a finalist digital signature scheme in the NIST PQC standardization project. It is built upon the well-known Fiat-Shamir with aborts framework [15] and its security is based on the computational hardness of the Module Learning With Errors (MLWE) and Module Short Integer Solution (MSIS) problems.

Algorithm 5: Saber.KEM.Encaps$(pk = (seed_{\boldsymbol{A}}, \boldsymbol{b}))$ [14]

1 $m \leftarrow \mathcal{U}(\{0,1\}^{256})$
2 $(\hat{K}, r) = \mathcal{G}(\mathcal{F}(pk), m)$
3 $c = \texttt{Saber.PKE.Enc}(pk, m; r)$
4 $K = \mathcal{F}(\hat{K}, c)$
5 **return** (c, K)

Algorithm 6: Saber.KEM.Decaps$(sk = (\boldsymbol{s}, z, pkh), pk = (seed_{\boldsymbol{A}}, \boldsymbol{b}), c)$ [14]

1 $m' = \texttt{Saber.PKE.Dec}(\boldsymbol{s}, c)$
2 $(\hat{K}', r') = \mathcal{G}(pkh, m')$
3 $c' = \texttt{Saber.PKE.Enc}(pk, m'; r')$
4 **if** $c = c'$ **then return** $K = \mathcal{H}(\hat{K}', c)$;
5 **else return** $K = \mathcal{H}(z, c)$;

Depending on the size of the module $R_q^{k \times \ell}$ with $k, \ell > 1$, Dilithium comes with three variants, namely Dilithium-2, 3 and 5 for the NIST-specified security levels 2, 3 and 5 respectively [7]. Dilithium-2 uses $(k, \ell) = (4, 4)$, Dilithium-3 uses $(k, \ell) = (6, 5)$ and Dilithium-5 uses $(k, \ell) = (8, 7)$. All the three variants of Dilithium use the polynomial ring $R_q = Z_q[x]/\langle x^{256} + 1\rangle$ with $q = 2^{23} - 2^{13} - 1$ a prime modulus. The key-generation, signing, and verification algorithms are described in Alg. 7, 8, and 9 respectively. Readers may follow the official specification of Crystals-Dilithium [7] for full details of the steps performed in the algorithms. These algorithms use the following low-level functions:

- ExpandA(): This function generates the polynomials in matrix $\boldsymbol{A} \in R_q^{k \times \ell}$ separately by expanding the common seed $\rho \in \{0,1\}^{256}$ along with different 16-bit nonce values. To generate a polynomial, SHAKE-128 is used to expand a seed-nonce pair and then the expanded bit string is post-processed using rejection sampling to ensure that all the coefficients are uniform in the set $\{0, \cdots, q-1\}$. The polynomials are generated in the NTT representation directly.
- ExpandS(): This function is used to generate the secret polynomial vectors $\boldsymbol{s}_1$ and $\boldsymbol{s}_2 \in S_\eta^\ell \times S_\eta^k$. Forr 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 $\{-\eta, \eta\}$.
- $\texttt{Power2Round}_q()$: It is used to perform bit-wise break up of an element in Z_q into higher-order and lower-order bits. An element $r = r_1 \cdot 2^d + r_0$ will be broken into r_0 and r_1, where $r_0 = r \mod^{\pm} 2^d$ and $r_1 = (r - r_0)/2^d$.
- $\texttt{HighBits}_q()$ and $\texttt{LowBits}_q()$: 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 in Power2Round(). Thus $\texttt{Decompose}_q()$ breaks an input $r \in Z_q$ into $r = r_1 \cdot \alpha + r_0$. Now r_1 will be the output of $\texttt{HighBits}_q()$ and r_0 will be the output of $\texttt{LowBits}_q()$.
- $\texttt{MakeHint}_q()$: It uses $\texttt{Decompose}_q()$ to produce a hint $\boldsymbol{h}$.
- $\texttt{UseHint}_q()$: It use the hint $\boldsymbol{h}$ produced by $\texttt{MakeHint}_q()$ 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 produces a polynomial with only τ coefficients set to $+1$ or -1 and the remaining coefficients set to 0.
- ExpandMask(): This function expands $\acute{\rho} \parallel \kappa$ using SHAKE to generate the polynomial vector $\boldsymbol{y}$. During this expansion, 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.
- NTT: Polynomial multiplications are performed using the Number Theoretic Transform (NTT) method.

The key generation of Dilithium (Alg. 7) samples random secret-key vectors $\boldsymbol{s}_1$ and $\boldsymbol{s}_2$ in line 3. The polynomials in these vectors have coefficients of magnitude at most η.

Algorithm 7: Dilithium.Gen() [7]

1 $\zeta \leftarrow \{0,1\}^{256}$
2 $(\rho, \varsigma, K) \in \{0,1\}^{256 \times 3} := \mathcal{H}(\zeta)$ {$\mathcal{H}$ is instantiated as SHAKE-256.}
3 $(\boldsymbol{s}_1, \boldsymbol{s}_2) \in S_\eta^\ell \times S_\eta^k := \texttt{ExpandS}(\varsigma)$
4 $\boldsymbol{A} \in R_q^{k \times \ell} := \texttt{ExpandA}(\rho)$ {$\boldsymbol{A}$ is generated and stored in NTT form as $\hat{\boldsymbol{A}}$}
5 $\boldsymbol{t} := \boldsymbol{A}\boldsymbol{s}_1 + \boldsymbol{s}_2$ { $\boldsymbol{A}\boldsymbol{s}_1$ is computed as $\texttt{NTT}^{-1}(\hat{\boldsymbol{A}} \cdot \texttt{NTT}(\boldsymbol{s}_1))$}.
6 $(\boldsymbol{t}_1, \boldsymbol{t}_0) := \texttt{Power2Round}_q(\boldsymbol{t}, d)$
7 $tr \in \{0,1\}^{384} := \texttt{CRH}(\rho || \boldsymbol{t}_1)$
8 **return** $(pk = (\rho, \boldsymbol{t}_1), sk = (\rho, K, tr, \boldsymbol{s}_1, \boldsymbol{s}_2, \boldsymbol{t}_0))$

The signing operation (Alg. 8) contains a loop that generates a potential signature and checks a set of constraints on the generated signature. When all the constraints are satisfied, a valid signature is produced as the output; otherwise the generated signature is rejected and the loop continues with generating another potential signature. These rejections are essential to avoid the dependency of the generated signature on the secret key. Inside the signing-loop, a masking vector $\boldsymbol{y}$ with coefficients less than magnitude γ_1 is generated. The polynomial c in line 11 is a sparse polynomial with exactly τ coefficients set to the values 1 or -1 and the rest 256 - τ coefficients set to zeros. A potential signature $\boldsymbol{z}$ is computed in line 12 and then constraints are checked starting from line 14 to 19.

Algorithm 8: Dilithium.Sign(sk, M) [7]

1 $\boldsymbol{A} \in R_q^{k \times l} :=$ ExpandA(ρ) {$\boldsymbol{A}$ is generated and stored in NTT form as $\hat{\boldsymbol{A}}$.}
2 $\mu \in \{0,1\}^{384} :=$ CRH($tr||M$)
3 $\kappa := 0, (\boldsymbol{z}, \boldsymbol{h}) :=\perp$
4 $\acute{\rho} \in \{0,1\}^{384} :=$ CRH($K||\mu$) (or $\acute{\rho} \leftarrow \{0,1\}^{384}$ for randomized signing)
5 Before the loop starts, precompute $\hat{\boldsymbol{s_0}} =$ NTT($\boldsymbol{s_0}$), $\hat{\boldsymbol{s_1}} =$ NTT($\boldsymbol{s_1}$), and $\hat{\boldsymbol{t_0}} =$ NTT($\boldsymbol{t_0}$)
6 **while** $(\boldsymbol{z}, \boldsymbol{h}) =\perp$ **do**
7 $\boldsymbol{y} \in \tilde{S}_{\gamma_1}^{\ell} :=$ ExpandMask($\acute{\rho}, \kappa$)
8 $\boldsymbol{w} := \boldsymbol{Ay}$ {This is computed as $\boldsymbol{w} :=$ NTT$^{-1}(\hat{\boldsymbol{A}} \cdot$ NTT$(y))$.}
9 $\boldsymbol{w}_1 :=$ HighBits$_q(\boldsymbol{w}, 2\gamma_2)$
10 $\tilde{c} \in \{0,1\}^{256} := \mathcal{H}(\mu \parallel \boldsymbol{w}_1)$
11 $c \in B_\tau :=$ SampleInBall($\tilde{c}$) {c is stored as $\hat{c} =$ NTT(c)}
12 $\boldsymbol{z} := \boldsymbol{y} + c\boldsymbol{s}_1$ {$c\boldsymbol{s}_1$ is computed as NTT$^{-1}(c \cdot \hat{\boldsymbol{s}}_1)$}
13 $\boldsymbol{r}_0 :=$ LowBits$_q(\boldsymbol{w} - c\boldsymbol{s}_2, 2.\gamma_2)$ {$c\boldsymbol{s}_2$ is computed as NTT$^{-1}(c \cdot \hat{\boldsymbol{s}}_2)$}
14 **if** $\parallel \boldsymbol{z} \parallel_\infty \geq \gamma_1 - \beta$ *or* $\parallel \boldsymbol{r}_0 \parallel_\infty \geq \gamma_2 - \beta$ **then**
15 $(\boldsymbol{z}, \boldsymbol{h}) :=\perp$
16 **else**
17 $\boldsymbol{h} :=$ MakeHint$_q(-c\boldsymbol{t}_0, \boldsymbol{w} - c\boldsymbol{s}_2 + c\boldsymbol{t}_0, 2.\gamma_2)$ {$c\boldsymbol{t}_0$ is computed as NTT$^{-1}(c \cdot \hat{\boldsymbol{t}}_0)$}
18 **if** $\parallel c\boldsymbol{t}_0 \parallel_\infty \geq \gamma_2$ *or* Hamming.Weight($\boldsymbol{h}$) $> \omega$ **then**
19 $(\boldsymbol{z}, \boldsymbol{h}) :=\perp$
20 $\kappa := \kappa + \ell$
21 **return** $\sigma = (\boldsymbol{z}, \boldsymbol{h}, \tilde{c})$

The verification operation (Alg. 9) is cheaper than key-generation and signing. It accepts a signature if all the three conditions specified in line 5 are satisfied.

III. Synergies and design decisions

As described in the previous section, 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 Dilithium and Saber. Furthermore, both schemes use Keccak-based [16] hash functions and pseudorandom number generators. Note that in ring or module lattice-based post-quantum public-key schemes, polynomial multiplications, hash calculations and pseudorandom number generations are the most expensive operations. The two schemes also have their own scheme-specific (hence distinct) building blocks. Fortunately, these exclusive building blocks have $O(n)$ time complexity and hence are computationally cheap. That gives us the freedom to design them using small amount of computational resources. These above-mentioned synergies in Dilithium and Saber motivated us to investigate efficient implementation techniques such that we could design a unified cryptoprocessor for accelerating the two schemes. Having a compact as well as a unified implementation of the two schemes could make lattice-based KEM and digital signature a reality on resource-constrained platforms. In the following part of this

Algorithm 9: Dilithium.Verify($pk, M, \sigma = (\boldsymbol{z}, \boldsymbol{h}, \tilde{c})$) [7]

1 $\boldsymbol{A} \in R_q^{k \times \ell} :=$ ExpandA(ρ) {$\boldsymbol{A}$ is generated and stored in NTT form as $\hat{\boldsymbol{A}}$.}
2 $\mu \in \{0,1\}^{384} :=$ CRH(CRH($\rho||\boldsymbol{t}_1$)$||M$)
3 $c :=$ SampleInBall($\tilde{c}$)
4 $\acute{\boldsymbol{w}}_1 :=$ UseHint$_q(\boldsymbol{h}, \boldsymbol{Az} - c\boldsymbol{t}_1.2^d, 2\gamma_2)$ {NTT$^{-1}(\hat{\boldsymbol{A}} \cdot$ NTT$(\boldsymbol{z}) -$ NTT$(c) \cdot$ NTT$(\boldsymbol{t}_1 \cdot 2^d))$.}
5 **return** [$\parallel \boldsymbol{z} \parallel_\infty < \gamma_1 - \beta$] **and** [$\tilde{c} = \mathcal{H}(\mu, \acute{\boldsymbol{w}}_1)$] **and** [# of 1's in $\boldsymbol{h}$ is $\leq \omega$]

section, we discuss these synergies in detail and also highlight the challenges in implementing a unified cryptoprocessor for Dilithium and Saber.

We would like to remark that synergies also exist in other lattice-based schemes. For example, CRYSTALS-Kyber [17] shows great similarities with Saber [6] as both are based on module lattices. Hence, our study could be extended to integrate Kyber along with Saber and Dilithium in the unified cryptoprocessor architecture. As a hardware implementation typically has a long design cycle, in this paper we stay focused on unifying Saber with Dilithium, and by doing so we show that a compact and unified cryptoprocessor for post-quantum digital signature and key exchange is feasible.

A. Polynomial multiplication

For multiplying two polynomials, the most commonly used algorithms are the schoolbook method, the Karatsuba method [18], the Toom-Cook method [19], the Fast Fourier Transform or Number Theoretic Transform method [20], with the time complexities $\mathcal{O}(n^2)$, $\mathcal{O}(n^{\log_2(3)})$, $\mathcal{O}(c(k)\cdot n^e)$ where $e = \log(2k-1)/\log(k)$, and $\mathcal{O}(n \log n)$ respectively. Different hybrid polynomial multiplication techniques are available by combining the above-mentioned algorithms.

Dilithium [7] makes the Number Theoretic Transform (NTT) method an integral part of the protocol to compute polynomial multiplications in least time. However, this brings a restriction on the choice of the coefficient-modulus in Dilithium as the modulus must be a prime of a special form to enable the NTT method.

On the contrary, an implementation of Saber [6] could use any type of polynomial multiplication algorithm or a hybrid of multiple algorithms. In Saber, the coefficient-moduli are powers-of-two and with that modular reductions become free of cost if schoolbook or Karatsuba or Toom-Cook or any combination of them is used. In [21] the hardware implementation of Saber uses a highly parallel schoolbook multiplier that computes one polynomial multiplication in just 256 cycles. An NTT-based polynomial multiplication [22] in Saber requires computations with respect to a larger prime modulus and cannot take advantage of free modular reductions.

When implementing a unified cryptoprocessor for both Dilithium and Saber, we have two options for computing polynomial multiplications. The first option is to instantiate an NTT-based multiplier for Dilithium and a schoolbook multiplier (following [21]) for Saber so that both schemes can be executed at their optimal speeds. This approach requires a large area in hardware and could potentially have a negative impact on the clock frequency of the implementation due to the increased routing complexity. 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. With the NTT-based multiplication, the temporary coefficient-modulus (which should be a prime) in Saber needs to be sufficiently large so that correct results are computed.

B. Hash and Expandable output functions

Besides polynomial multiplications, Keccak-based [16] hash computations and pseudorandom number generations are used in both Dilithium and Saber. For example, Saber uses SHAKE-128 for pseudorandom string generations and SHA3-256/512 for hash calculations. Similarly, Dilithium uses both SHAKE-128 and SHAKE-256 for pseudorandom string generations and SHA3-256 for hash calculations. Since all of the SHAKE and SHA3 operations essentially use the Keccak sponge function internally, a common Keccak core along with a wrapper will be enough to support the needs of both Dilithium and Saber. The wrapper around the Keccak core will implement the functionalities of different SHAKE and SHA3 modes. Note, several previous works [11], [21] showed that if a high-speed Keccak module is used in a lattice-based cryptoprocessor, then a lion share of the resources is spent on implementing the high-speed Keccak core. Hence, by using a common Keccak core in the hardware, we could greatly reduce the area of a unified cryptoprocessor for Saber and Dilithium.

We would like to remark that the Keccak module will not be specific to the designed signature and KEM schemes. Following the standardization of Keccak as the SHA3 algorithm, several new generation commercial processors have integrated hardware supports for Keccak. If these processors are extended to includ post-quantum signature and KEM, then their Keccak cores could be reused.

C. Remaining scheme-specific building blocks

The remaining building blocks are of $O(n)$ complexity and do not share many similarities. They mostly perform simple additions, subtractions, packing, etc. To further reduce the area consumption one option would be to use the addition and subtraction units present in the butterflies for `NTT`. This will decrease the area but make the design complex and dependable on the availability of the butterfly units thus hindering the parallel computations. Therefore, in order to make the design simple and easily configurable we decide to keep the remaining building blocks with their own simple addition subtraction units required.

In the next section, we discuss the various design decisions we took and various trade-offs that we considered during the implementation of the common cryptoprocessor.

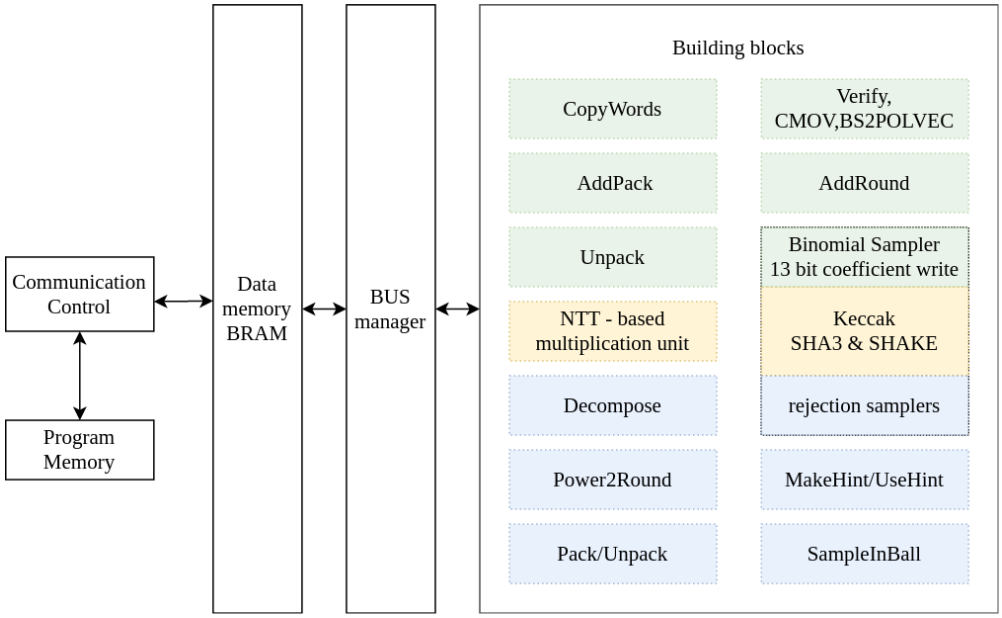

Fig. 1. The high-level design of the cryptoprocessor with the Saber modules in green, Diltihium's modules in blue and common modules in yellow. For the scheme-specific uncommon blocks, computational and architectural synergies exist.[Simplified - Other smaller building blocks are not included]

IV. Optimized implementation

As described in the previous section, the proposed unified cryptoprocessor for Dilithium and Saber will have an NTT-based polynomial multiplier, a Keccak-core and a SHA/SHAKE 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 require very optimized implementations for realizing a unified cryptoprocessor. Although the remaining scheme-specific blocks are computationally cheap (thanks to their $\mathcal{O}(n)$ time complexities), their implementations as hardware-blocks could be significantly time consuming and laborious as they require low-level bit, or byte or word manipulations and signed arithmetic. In this section we describe how we implement the building blocks of the unified cryptoprocessor. Fig. 1 shows a high-level block diagram of the unified cryptoprocessor.

A. NTT-based unified polynomial multiplier

In this section, we describe 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 as the temporary modulus p' so that no actual modular reductions take place [22].

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. Hence, for positive coefficients a modulus of order $2^3 \times 2^{13} \times 256 = 2^{24}$ is sufficient but for negative coefficients, if we convert them to unsigned values by performing modular reduction by $q = 2^{13}$, then the required modulus size increases to $2^{13} \times 2^{13} \times 2^8 = 2^{34}$. Implementing a common NTT-based multiplier that supports 2^{34} order modulus will become expensive in comparison to the one supporting 2^{23} order modulus required by Dilithium. In [23], [22], the designers also discuss a similar problem and mention that a 24-bit modulus can be used along with special provision for signed number representation. However, we observe that if we still take a prime p' of the order of 2^{24} and convert the negative coefficients to unsigned values modulo p', that is in $[0, p'-1]$, the modular reductions caused are ineffective and we get the correct result. This is explained in Appendix A of this paper.

In this way, we need a common NTT core that supports *a 24-bit prime* for Saber and *the* 23-bit prime modulus of Dilithium. Now we describe how we choose an appropriate prime p' for Saber. Of all the modular arithmetic operations that are performed during an NTT, modular multiplication is the most expensive in terms of both area and time. The original software source code of Dilithium [7] uses the Montgomery modular reduction. We observe that a dedicated bit-parallel modular reduction unit will be small and more efficient on hardware platforms as the prime $q = 2^{23} - 2^{13} + 1$ in Dilithium has a sparse structure and therefore a fast modular reduction could be implemented using simple additions and subtractions. For Saber, we choose the 24-bit prime p' in such a way that it resembles the prime of Dilithium closely.

Efficient modular reduction unit: If the two primes have similar structures, then their modular reduction circuits can be unified very well to reduce the area overhead. Hence, we take $p' = 2^{24} - 2^{14} + 1$ as the NTT-modulus of Saber. A circuit diagram for the optimized modular reduction unit is shown in Fig. 2.

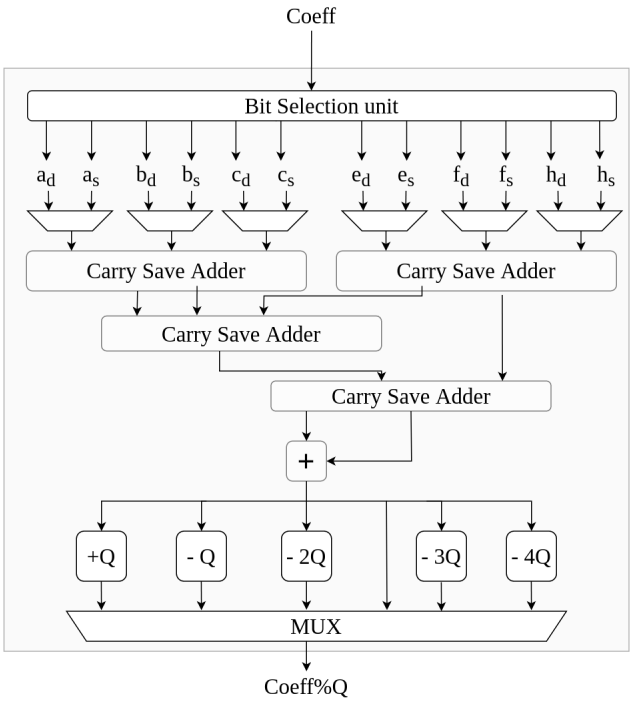

Fig. 2. Unified modular reduction unit for both Saber and Dilithium. The bit selection unit selects the different amount of bits required by the two schemes for processing

Post-processing elimination: Dilithium's official software implementation [7] uses an extra loop at the end of the inverse NTT for scaling the output coefficients, as shown in [24]. In our implementation, these extra scaling-related multiplications are removed by processing the coefficients using the following equation [25] during inverse NTT.

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

Here, q is the modulo and x is the result of the butterfly operation during inverse NTT. This way both the NTT and inverse NTT are of the same cost and require no post-processing.

Algorithm 10: The Cooley-Tukey NTT algorithm [24]

Input : A vector $x = [x_0, \cdots, x_n - 1]$ where $x_i \in [0, p-1]$ of degree n (a power of 2) and modulus q = 1 mod 2n
Input : Precomputed table of 2n-th roots of unity **g**, in bit reversed order
Output : $x \leftarrow NTT(x)$
function $NTT(x)$
  $t \leftarrow n/2$
  $m \leftarrow 1$
  while $m < n$ **do**
    $k \leftarrow 0$
    for $i \leftarrow 0; i < m; i \leftarrow i + 1$ **do**
      $S \leftarrow \mathbf{g}[m + i]$
      for $j \leftarrow k; j < k + 1; j \leftarrow j + 1$ **do**
        $U \leftarrow x[j]$
        $V \leftarrow x[j + t].S \bmod q$
        $x[j] \leftarrow U + V \bmod q$
        $x[j + t] \leftarrow U - V \bmod q$
      $k \leftarrow k + 2t$
    $t \leftarrow t/2$
    $m \leftarrow 2m$
  return

Internal architecture of NTT: Following the official reference code of Dilithium, we use the Cooley-Tukey (Alg. 10) and Gentleman-Sande (Alg. 11) butterfly configurations for the NTT and inverse NTT respectively. Both butterfly configurations are implemented in a unified butterfly core. Fig. 3 shows the internal blocks of the unified butterfly core. The multiplexers are used to select the appropriate data-paths during the Cooley-Tukey and Gentleman-Sande butterfly operations. The arithmetic circuits, namely modular multiplier, adder and subtracter, are all pipelined to achieve high clock frequency.

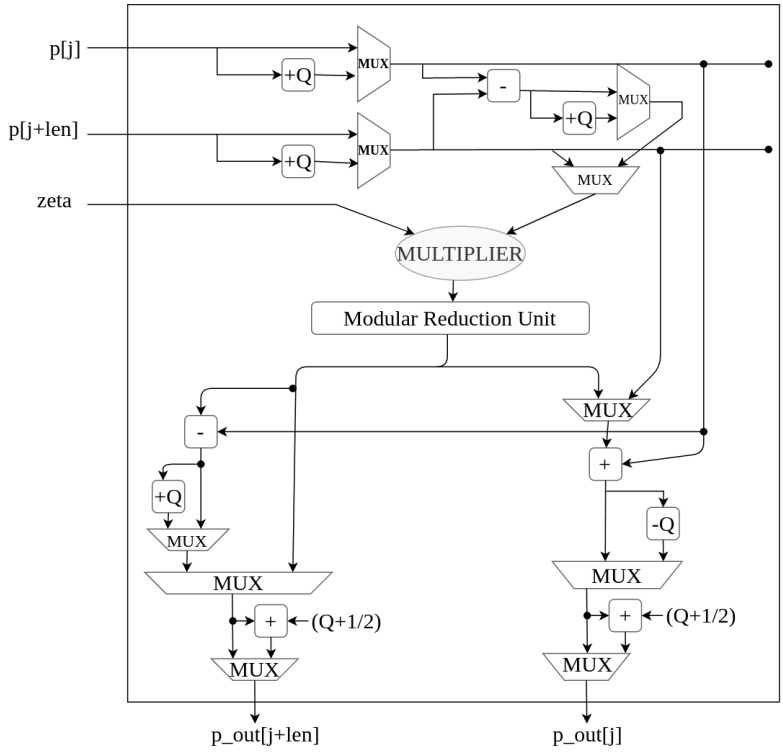

Fig. 3. Internal architecture of the butterfly unit for unified Cooley-Tukey NTT and Gentleman-Sande inverse NTT

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 [26]. That 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. This spreading is necessary as one BRAM-set could feed only one butterfly core due the limitations in the number of read/write ports. 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. At any time during an NTT or inverse NTT, the two coefficients in a single memory-word have an index difference of $l/2$ where $l \in \{N, N/2, N/4, \ldots, 4, 2\}$ during the outermost NTT-loops (Alg. 10), and $l \in \{2, 4, \ldots, N/4, N/2, N\}$ during the outermost inverse NTT-loops (Alg. 11). In this way 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 inverse NTT operations take 512 clock cycles only.

Fig. 4 shows the arrangement of coefficients in memory words during NTT loop-iterations using a toy example. For the

Algorithm 11: The Gentleman-Sande inverse NTT algorithm [24], [25]

Input : A vector $x = [x_0, \cdot, x_n - 1]$ where $x_i \in [0, p - 1]$ of degree n (a power of 2) and modulus q = 1 mod 2n
Input : Precomputed table of 2n-th roots of unity $\mathbf{g}^{-1}$, in bit reversed order
Input : n^{-1} mod q
Output : $x \leftarrow INTT(x)$
function $INTT(x)$
  $t \leftarrow 1$
  $m \leftarrow N/2$
  while $m > 0$ **do**
    $k \leftarrow 0$
    for $i \leftarrow 0; i < m; i \leftarrow i + 1$ **do**
      $S \leftarrow \mathbf{g}^{-1}[m + i]$
      for $j \leftarrow k; j < k + 1; j \leftarrow j + 1$ **do**
        $U \leftarrow x[j]$
        $V \leftarrow x[j + t]$
        $x[j] \leftarrow (U + V)/2 \bmod q$
        $W \leftarrow U - V \bmod q$
        $x[j + t] \leftarrow (W.S)/2 \bmod q$
      $k \leftarrow k + 2t$
    $t \leftarrow 2t$
    $m \leftarrow m/2$
  return

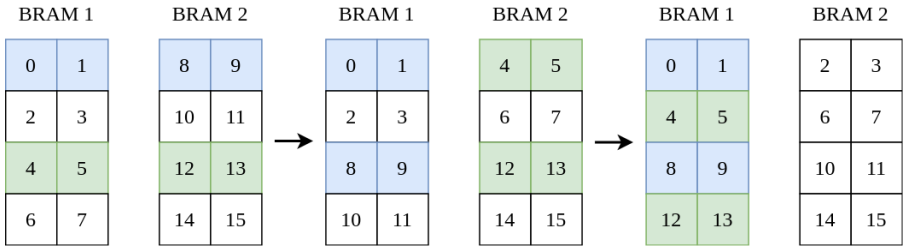

Fig. 4. Coefficients storage for 3 iterations of NTT on a polynomial of coefficient size 16

simplicity of the explanation when we say 'coefficient i' we actually mean the coefficient with the index i. 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. In the first iteration, the coefficients (shown in blue color in Fig. 4) zero-and-eight are input to the first butterfly unit; and the coefficients one-and-nine are input to the second butterfly unit. After the first iteration, we want the processed coefficients eight-and-nine to be stored in BRAM 1, at the address where currently coefficients four-and-five are stored. This will simplify the coefficient read pattern during the next iteration of the loop. Because coefficients four-and-five have not been processed yet, we can not write new values to their memory location. Hence, we read the coefficients four-and-five, and similarly twelve-and-thirteen immediately after reading the coefficients zero-and-one, and eight-and-nine respectively. We do similar for every iteration and this approach avoids the read-write conflict and simplifies the read and writes for both NTT and Inverse NTT.

B. 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 [27]. 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 pseudorandom polynomial coefficients can be generated in scheme-specific optimal representations and then stored immediately in the memory of the cryptoprocessor. This strategy helps reducing the overall cycle counts for both Dilithium and Saber.

Saber's public polynomials in $\boldsymbol{A}$ have 13-bit coefficient size and they are generated by the expandable output function SHAKE-128 (Alg. 1 and 2). When these polynomials are multiplied, they are converted into the NTT representation in our unified cryptoprocessor. As described in Sec. IV-A, the NTT unit requires its operand data to be present in 'two coefficients per BRAM word' format. One option for processing the public polynomials will be to generate a continuous bit stream 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 a separate parser hardware. This approach is sequential by nature and results in a bloated cycle count. In order 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. Therefor, 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 (Alg. 1 and 2), 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. Our aim is 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 want to reduce the LUT count by reducing the buffers, but we also need to make the design simple and ensure that it does not increase the timing of the design. The ability of our design to execute data-independent operations in parallel helps us here. It gives us the freedom to slow the Keccak squeeze as it is run in parallel with the NTT and finishes way before NTT. We decide to just make three inter-meditate buffers for zero, two, and four shifts, for both the output buffer and left-over-bits buffer, as shown in Fig. 5. After the Keccak squeeze is done we write the remaining bits to the left-over-bits buffer. In order 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 left. Once the left-over-bits buffer is aligned. We start shifting both the output buffer and left-over-bits buffer towards left by four or two. The values pushed out by the left-over-bits buffer are put in front of the output buffer.

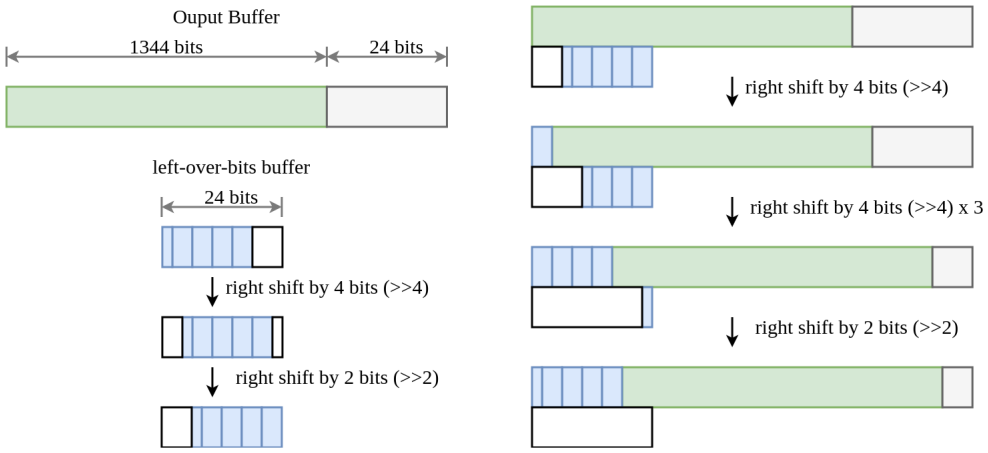

Fig. 5. The output buffer containing the squeeze output in green and the left-over-bits buffer in blue containing the remaining bits of the previous squeeze. This figure shows an example of how the two buffers are shifted when there are 18 remaining bits.

C. Samplers

Saber uses binomial sampler in which integers are sampled from a centered binomial distribution as described in Sec. II. Dilithium requires three different kinds of rejection sampling units for coefficient generation. To use them three different instructions are provided. The uniform rejection sampling used for generating the public matrix vector requires 3 bytes of Keccak output to generate one coefficient, the eta-uniform rejection sampling used for generating secret key coefficients consumes one byte of Keccak output to generate two coefficients, and gamma-rejection sampling used for generating the $\mathbf{y}$ polynomial consumes nine bytes of Keccak output to generate four coefficients. Although we can't make a common block for all the different types of sampling, we try to put in an overall optimization in the cycle count and area.

For the uniform, and the η sampling, we need to use 24 bits and 4 bits respectively. These utilize the Keccak output buffer fully after every squeeze so we need to extract 4 or 24 bit output after Keccak squeeze. The γ sampling outputs 18 bit values and this does not utilize the Keccak output fully, and after every squeeze 8 bits are leftover. The maximum squeezes required for generation are 4 so we increase the size of leftover buffer to 32. The same approach of shifting the output buffer and leftover bit buffer described above can also be used for this sampling. Now we have the big Keccak output buffer giving out 5 different types of outputs 4 bits, 13 bits, 18 bits, 24 bits, and 64 bits. This is controlled using a multiplexer, which in hardware would mean making copies of the big buffer 5 times and based on multiplexer input gives out the required output. This is again very expensive and in order to reduce this we club together the squeezes for 4, 24, and 64 bits into one 192 bit buffer. So now, we have 3 multiplexer outs from the smaller buffer for three types of squeezes but only one 192bit squeeze out instead of the other three. Now our Keccak output buffer only has three different shift outs. Thus we save around 1200 LUTs. We also add the optimized implementation of packing unpacking these polynomials using a common buffer for all the different kinds of packing modes and unpacking modes required by Dilithium.

D. Memory

For the Dilithium variant with the NIST security level 2, the public matrix vector has dimensions 4×4. During signing we need to precompute and store the secret vectors $\mathbf{s}_1$ (4×1), $\mathbf{s}_2$ (4×1), and $\mathbf{t}_0(4 \times 1)$ in the NTT representation, thus requiring storage for 12 polynomials. Storing the entire public matrix in the memory makes the signing operation faster. We now require to store 28 polynomials before the signing starts. During signing operation, we need one storage to store the results temporary results.

Saber has a public matrix with the dimensions 3×3. We will need to have storage for three secret polynomials and nine public polynomials in the NTT representation. It can be seen that the overall memory requirement of the cryptoprocessor is determined by Dilithium as it consumes more memory than Saber. Two coefficients of every polynomial are stored together in one 64 bit word and therefore, one polynomial occupies 128 address spaces. Parallel memory organization is used to ensure efficient load and storage of polynomials. This is especially important for parallel execution of NTT and Keccak.

With all the constraints in consideration and flexibility requirements in place, the implementation of Saber requires only four BRAM36K elements, owing to its small matrix and vector dimensions and small polynomial coefficients (13 bits). However, this was not sufficient for Dilithium as the public matrix and other vectors have huge dimensions. Additionally, Dilithium uses

a coefficient size of 23 bits. So, the memory was split across four major blocks, with each of them having three BRAM36K elements. This is done to support the storage requirement of Dilithium and enable parallel execution of NTT and Keccak.

The constants for NTT and inverse NTT computations are kept in a ROM which is also interpreted using BRAMs in our implementation. Along with this the program controller, for loading all the instructions at once in a separate instruction memory and then handling all the data independent executions in parallel, requires one36k and one18k BRAM.

E. Remaining building blocks

The output given after the polynomial multiplication has now two coefficients per word instead of 4 coefficients per word storage style used in [21]. Therefore, modules UnPack, AddPack, and AddRound individually now take at most twice the number of clock cycles. However, we are able to avoid the extra clock cycles required for pre-processing the input for feeding into these modules. Decompose and Power2Round are implemented as per the specification, consuming 128 clock cycles for processing one polynomial. Since the Decompose module required by MakeHint and UseHint is already implemented, we just implement the equality checkers which return the desired output. BS2POLVEC now converts byte-stream to vector form as required by the NTT module for one polynomial at a time. Modules Copy, Verify, and CMOV work the same way as provided in [21]. We are also able to reduce the overall area consumption of these building blocks by using common addition, subtraction, and polynomial read and write control units, as described in section III-C.

In the coming up sections, we discuss the timing results and area consumption in comparison with the existing works, and the future scope of the design.

F. Parallel processing of Keccak and polynomial arithmetic

In [28], [29] overlapping of data-independent computations at block levels 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 [21] all the instructions, including data-independent, 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. For example, pseudorandom polynomials are generated using the Keccak core and they are immediately consumed by the NTT unit one-by-one to compute polynomial multiplications. 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 which loads all the instructions in an Instruction RAM and then send them one by one to the compute core for processing in parallel or sequence as specified in the instruction. The two types of instructions are shown in Fig. 6 and they are stored together along with 4 control bits in the Instruction RAM.

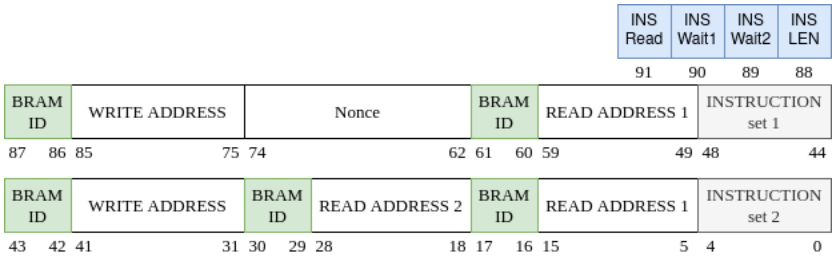

Fig. 6. Format of the instructions stored in Instruction Memory

V. Timing and Utilization Results

The proposed unified cryptoprocessor architecture is described entirely in Verilog and it is implemented for FPGA and ASIC platforms. For FPGA, the proposed architecture is synthesized and implemented using Vivado 2019.1 tool suite for the target platform Zynq Ultrascale+ ZCU102 with area-optimized implementation strategy. The FPGA implementation achieves 200 MHz clock frequency. For ASIC, the proposed architecture is synthesized with UMC 65nm library and it achieves 370 MHz clock frequency.

TABLE I
CYCLE COUNT FOR OPERATIONS IN SABER-KEM AND DILITHIUM-2

Operation	Cycle count	Latency	
		FPGA	ASIC
Saber.KEM.Keygen	10,980	54.9	29.6
Saber.KEM.Encaps	14,504	72.5	39.2
Saber.KEM.Decaps	18,955	94.7	51.2
Dilithium.Gen	15,618	78.0	42.2
Dilithium.Sign$_{pre}$	8,496	42.4	22.9
Dilithium.Sign	20,914	104.5	56.5
Dilithium.Sign$_{post}$	3,595	17.9	9.7
Dilithium.Verify	17,713	88.5	47.8

A. Timing Results

In Table I, we present the cycle count and latency (in μs) for the operations of Saber (key generation, encapsulation, decapsulation) and Dilithium-II (key generation, sign, verify). With 200 MHz clock frequency in FPGA, the CCA-secure key generation, encapsulation and decapsulation operations for Saber take 54.9, 72.5 and 94.7 μs, respectively. The ASIC implementation with 370 MHz clock frequency after the synthesis can perform the key generation, encapsulation and decapsulation operations for Saber in 29.6, 39.2 and 51.2 μs, respectively.

The Dilithium signature generation operation has a loop and it iterates until a valid signature is generated. In Table I, 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 sign part is repeated until a valid signature is generated. For the best-case scenario, the key generation, signature generation and signature verification operations for Dilithium-II take 78.0, 164.8 and 88.5 μs, respectively, in the FPGA platform. The ASIC implementation with 370 MHz clock frequency after the synthesis can perform the key generation, signature generation and signature verification operations for Dilithium-II in 42.2, 89.1 and 47.8 μs, respectively.

B. Utilization Results

In the Table II, we present the detailed utilization of each building blocks in the cryptoprocessor for UltraScale+ ZCU102 platform. The proposed cryptoprocessor achieves 200 MHz clock frequency and it uses 18,040 LUTs (6.5%), 9,101 DFFs (1.6%), 4 DSPs (0.1%) and 14.5 BRAMs (1.5%) 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. The multiplier has two butterfly units and each butterfly unit requires two DSP blocks for performing one 24-bit$\times$24-bit multiplication. Note that we use one BROM to store precomputed powers of $2n$-th root of unity values for NTT and inverse NTT operations. The data memory has four BRAM-sets where each set is a 64-bit wide and 1,536-words deep memory element. They consume 268 LUTs and 12 BRAMs in total.

We also synthesized our design with UMC 65nm ASIC library using Cadence Genus synthesis tool. The proposed cryptoprocessor achieves 370 MHz with 0.301mm^2 area ($\approx$200.6 kGE) excluding on-chip memory for storing data and precomputed powers of $2n$-th root of unity.

C. 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-II as shown in Table III and Table IV, respectively. In the literature, there are several works targeting an unified architecture that supports multiple PQC schemes [11], [12], [30]. In [11], the authors present *Sapphire*, a cryptoprocessor coupled with RISC-V processor implemented in ASIC for various lattice-based PQC schemes. It presents one of the earliest works for ASIC platform and supports parameter sets for NTT-friendly Round 2 candidates in NIST's PQC standardization. It uses constant-geometry NTT algorithm to reduce area cost of the memory blocks. It does not support or provide performance results for Saber while the results provided for Dilithium are using Round-2 specifications. For a fair comparison, we compare Round-2 Dilithium-II with dimensions (4,3) and Round-3 Dilithium-II with dimensions (4,4). Compared to *Sapphire*, our FPGA and ASIC implementations show up to $\times$23 and $\times$42 better performance, respectively.

In [12], the authors present a RISC-V architecture coupled with optimized hardware accelerators for improving the performance of lattice-based PQC algorithms. It provides support for Crystals-Kyber, NewHope and Saber schemes and targets ASIC platform. Compared to the Saber implementation in [12], our FPGA and ASIC implementations show up to 304$\times$ and 564$\times$ better performance, respectively. Also, their implementation consumes more area then our ASIC implementation. The work in [30] 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.

TABLE II
UTILIZATION REPORT OF THE CRYPTOPROCESSOR

Unit	LUTs	FFs	DSPs	BRAMs
ComputeCore	**16,949**	**8,576**	**4**	**13**
⌊AddPack	233	161	0	0
⌊AddRound	350	362	0	0
⌊BS2POLVEC	343	360	0	0
⌊Unpack (Saber)	184	196	0	0
⌊Verify (Saber)	101	208	0	0
⌊CMOV	13	34	0	0
⌊COPY	8	34	0	0
⌊Decompose	453	286	0	0
⌊Power2Round	116	62	0	0
⌊MakeHint	306	119	0	0
⌊UseHint	603	393	0	0
⌊EncodeH	186	231	0	0
⌊Pack/Unapck (Dilithium)	2,015	1,118	0	0
⌊SampleInBall	476	258	0	0
⌊Refresh	4	7	0	0
⌊Verify (Dilithium)	27	69	0	0
⌊Sampler	174	93	0	0
⌊Memory	268	8	0	12
⌊Multiplier	2,454	1,055	4	1
⌊Keccak	8,653	3,514	0	0
ProgramController	1,120	248	0	1.5
Total	**18,040**	**9,101**	**4**	**14.5**

To the best of our knowledge, there are three FPGA-based implementations of Dilithium [31], [32], [33] in the literature. Zhou *et al.* [31] propose a HW/SW co-design solution for improving the performance of Dilithium compared to the full-software implementation. They offload computationally intensive operations such as SHA/SHAKE and polynomial multiplication to the hardware while keeping the rest of the operation in the software. Although their hardware implementation consumes small area, our pure-hardware solution shows up to two order of magnitude better performance compared to their HW/SW co-design solution. In [32], the authors present three high-performance architectures for key generation, signature generation and signature verification operations of Dilithium scheme targeting FPGA platform. Their three implementations can perform key generation, signature generation and signature verification operations in 36, 55 and 66 μs, respectively. Although they show slightly better performance than our implementation, their implementation for signature generation consumes $3.7\times$, $8.6\times$, $241.2\times$ and $10\times$ 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 Saber scheme as well. The work in [33] presents a Dilithium implementation for low-end Artix-7 FPGAs. They target reducing LUT utilization by employing extra DSP units for computations. Our implementation shows slightly better performance and uses less hardware resources. Their implementation uses $1.5\times$ more LUTs and $11.2\times$ more DSPs units. For the best-case scenario, our implementation shows $1.47\times$, $1.08\times$ and $1.36\times$ better performance for the key generation, signature generation and signature verification operations, respectively.

In [34], the authors evaluate the hardware performance of Round-2 PQC signature schemes using high-level synthesis (HLS) tool. They present different architecture for each operation and they apply various HLS directives (i.e., pipelining) to improve the performance of the implementations. Although HLS directives show slight performance improvements, our implementation is superior in terms of both area and performance.

There are several works in the literature implementing Saber in hardware for FPGA [21], [35], [36], [37], [38] and ASIC [39], [40] platforms. Compared to the work in [21], our FPGA implementation uses fewer LUTs even though we support both Saber and Dilithium schemes. However, their implementation shows better performance due to their very fast schoolbook polynomial multiplication unit with 256 processing cores. As we target a compact architecture supporting both Saber and Dilithium, we use an NTT-based polynomial multiplier unit with just two processing elements.

In [35], the authors propose a high-performance FPGA implementation of Saber. Compared to our work, their implementation shows similar performance at the expense of using $64\times$ more DSP units. In [37], the authors present a lightweight FPGA implementation of Saber. Compared to their work, our cryptoprocessor shows up to $6.4\times$ better performance while using $8\times$ less DSP units. Mera *et al.* [38] presents a HW/SW co-design for Saber with small LUT and FF consumption. However, their implementation shows up to $68\times$ worse performance compared to our design implemented entirely in hardware.

In [36], a compact cryptoprocessor for Saber is proposed. The proposed work presents a novel strategy to improve the performance of polynomial multiplication operation. Their implementation shows slightly better performance compared to our work and uses less hardware resources. As we target both Saber and Dilithium which requires NTT-based polynomial multiplier and works with large dimension modules, our architecture consumes more area and memory. In [39] and [40], two high-performance ASIC implementations for Saber are presented. Compared to our ASIC implementation, they show better area consumption and performance. However, their implementations are optimized for Saber scheme as our work targets multiple

TABLE III
COMPARISON TABLE FOR SABER WITH MODULE DIMENSION 3

Work	**Support for Multiple Schemes**	**Platform**	**Performance** (in μs) KG/E/D*	**Freq.** (MHz)	**Area** (mm^2 for ASIC) LUT/FF/DSP/BRAM
[12]	Yes	ASIC/65nm	16.7K/21.9K/26.4K	45.47	0.914 mm^2
[39]	No	ASIC/65nm	7.1/7.1/9.3	1000	0.314 mm^2
[40]	No	ASIC/40nm	2.7/3.6/4.3	400	0.38 mm^2
[30]†	Yes	Artix-7	3.6K/4.9K/5.5K	62.5	20.6K/11.8K/13/36.5
[38]†	No	Artix-7	3.2K/4.1K/3.8K	125	7.4K/7.3K/28/2
[37]	No	Artix-7	–/467.1/527.6	100	6.7K/7.3K/32/0
[36]	No	UltraScale+	48.9/63.2/78.5	250	10.1K/7.7K/0/3
[35]†	No	UltraScale+	-/60/65	322	12.5K/11.6K/256/4
[21]	No	UltraScale+	21.8/26.5/32.1	250	23.6K/9.8K/0/2
This[a]	Yes	UltraScale+ ASIC/65-nm	54.9/72.5/94.7 ≈29.6/39.2/51.2	200 370	18.0K/9.1K/4/14.5 ≈0.821 mm^2

*: KG: Key generation, E: Encapsulation, D: Decapsulation.
†: HW/SW co-design.
[a]: Area of memory (≈0.520 mm^2) is estimated.

TABLE IV
COMPARISON TABLE FOR DILITHIUM-II

Work	**Support for Multiple Schemes**	**Platform**	**Performance** (in μs) KG/S/V*	**Freq.** (MHz)	**Area** (mm^2 for ASIC) LUT/FF/DSP/BRAM
[11][d,f]	Yes	ASIC/40-nm	1.8K/7.1K/2.5K	72	0.28 mm^2
[34][a]	No	HLS/Artix-7	1.4K/-/-	119	17.6K/86.6K/-/-
[34][b,f]			-/10.7K/-	114	21K/90.5K/-/-
[34][c]			-/-/1.8K	114	15.1K/65.2K/-/-
[31]†	No	Zynq-7000	-/8.8K/9.9K	100	2.6K/-/-/-
[32][a]	No	UltraScale+	36/-/-	350	54.1K/25.2K/182/15
[32][b,e]			-/55/-	333	68.4K/86.2K/965/145
[32][c]			-/-/66	158	61.7K/34.9K/316/18
[33][a]	No	Artix-7	84.9/-/-	221	11.0K/7.2K/45/11
[33][b,f]			-/427.9/-	179	18.0K/9.1K/45/15
[33][c]			-/-/98.3	200	12.1K/7.5K/45/11
[33][d,e]	No	Artix-7	115.0/178.3/120.7	163	27.4K/10.6K/45/15
[33][d,f]			115.0/469.8/120.7		
This[d,e,g]	Yes	UltraScale+ ASIC/65-nm	78.0/164.8/88.5 ≈42.2/89.1/47.8	200 370	18.0K/9.1K/4/14.5 ≈0.821 mm^2

*: KG: Key generation, S: Sign, D: Verify.
†: HW/SW co-design.
[a]: Implementation for key generation operation.
[b]: Implementation for sign operation.
[c]: Implementation for verify operation.
[d]: Implementation for all operations.
[e]: Reports sign performance for best-case scenario.
[f]: Reports sign performance for average-case scenario.
[g]: Area of memory (≈0.520 mm^2) is estimated.

schemes.

In the next section, we discuss the future scope of the work and conclude the paper.

VI. EXTENDING SUPPORT FOR OTHER SCHEMES

We intend to include support for all the different variants of Saber and Dilithium. We also intend to include Crystals-Kyber to our cryptoprocessor. Kyber also uses polynomials of coefficients size 256, the same as Saber and Dilithium. Kyber uses NTT-based polynomial multiplication where the prime is 12-bit. A minor datapath adaption of the existing NTT architecture would enable NTTs for Kyber. Furthermore, Kyber also uses the Keccak-based SHA3 and SHAKE functions as Dilithium and Saber do. These hash functions are already present in the proposed cryptoprocessor and thus pseudorandom number generation and hash calculations in Kyber can be supported. Similar to Saber, Kyber also uses binomial sampling. Although the parameters used by the binomial samplers in Saber and Kyber are not always the same, the controller unit and parts of the existing binomial sampler datapath could be adjusted to extend support for Kyber. Although the complementary building blocks used by Kyber are not very similar to those used in the current version of the unified cryptoprocessor, they could be added to the architecture. Owing to the small coefficient size and module dimension, the memory blocks available in the cryptoprocessor are also sufficient for computing all security-levels of Kyber.

VII. Conclusion

By designing a unified hardware architecture for the two finalists Crystals-Dilithium and Saber KEM of the NIST Post Quantum Cryptography Standardization, we showed that it is possible to realize a compact yet fast cryptoprocessor for performing both post-quantum key-exchange and digital signature on ASIC and FPGA platforms.

The optimized cryptoprocessor architecture greatly benefits from the algorithmic and structural similarities in the two implemented cryptographic schemes. The most expensive operations in both Dilithium and Saber are polynomial multiplications, and Keccak-based SHA3 and SHAKE computations. We demonstrated that by instantiating a unified NTT-based polynomial multiplier, we can compute the polynomial multiplications of both schemes. Furthermore, by using a special prime modulus for computing the NTTs of Saber, we can greatly minimized the area overhead of the unified multiplier compared to a Dilithium-only multiplier. Similarly, starting from a high-speed Keccak core, we designed an optimized wrapper around it to pre-process the inputs and post-process the outputs of SHA3 and SHAKE on-the-fly, and by doing so we effectively reduced the number of unnecessary memory read and write cycles. Finally, with all the optimizations, our unified cryptoprocessor on a Xilinx FPGA computes Saber's key generation, encapsulation, and decapsulation in 54.9, 72.5 and 94.7 μs respectively; and Dilithium-II's key generation, signing (best case) and verification in 78.0, 164.8 and 88.5 μs respectively. The designed cryptoprocessor is even faster or smaller than several of the previously published works on Dilithium-only implementations on hardware platforms.

In the future, we intend to integrate more lattice-based schemes while keeping the design lightweight. We also intend to design and implement unified countermeasures for protecting our cryptoprocessor from side-channel and fault attacks in low time and area overheads.

VIII. Acknowledgements

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

Appendix A
Proof of using a 24-bit prime for Saber NTT

The idea for using NTT for Saber's polynomial multiplication, relied on the fact that we can always select a prime big enough to avoid any modular reduction. However, it is important to note that using too big a prime effects the cost of hardware adversely. Secret polynomial have signed coefficients and we get the following two options to deal with them:

- Convert signed coefficients to mod $q(=2^{13})$.
  Required prime $> 2^{13} \cdot 2^{13} \cdot 256 = 2^{34}$.
- Coefficients have signed bit representation.
  Required prime $> 5 \cdot 2^{13} \cdot 256 = 10485760 (\approx 2^{24})$.

While the first option leads to very expensive multiplication and reduction units, the second option requires a special implementation to deal with signed representation to treat these coefficients. In the following proof we show how we can use a 24-bit prime for mod q representation.

A. *Proof*

Let us say we have two polynomials $a(x)$ and $b(x)$ of degrees $n = 256$ each. Coefficients of $a(x)$ are in range $[-5, 5]$ and coefficients of $b(x)$ are in range $[0, 2^{13} - 1]$. Then we define polynomials $c(x)$ as the multiplication of the two polynomials $a(x)$ and $b(x)$ i.e., $c(x) = a(x) \cdot b(x)$. Then the coefficients of $c(x)$ will be of the following form:

$$c_0 = a_0 \cdot b_0 - a_1 \cdot b_{n-1} - a_2 \cdot b_{n-2} \cdots - a_{n-1} \cdot b_1$$

$$c_1 = a_0 \cdot b_1 + a_1 \cdot b_0 - a_2 \cdot b_{n-1} \cdots - a_{n-1} \cdot b_2$$

$$\vdots$$

$$c_{n-1} = a_0 \cdot b_{n-1} + a_1 \cdot b_{n-2} + a_2 \cdot b_{n-3} \cdots + a_{n-1} \cdot b_0$$

Let's consider the cases for c_{n-1} as an example . If we use a prime modulus $p > 5 \cdot 2^{13} \cdot 256 = 10485760 (\approx 2^{24})$, thein the case when all the secret polynomial coefficients are positive,the maximum value for c_{n-1} would be $2^{13}.5.256$ so taking this value mod p won't lead to any modular reduction. In the case when all coefficients are negative we'll convert them to mod p which will give us a maximum possible value as $(p-1) \cdot 2^{13} \cdot 256$ which is clearly much than our prime p. This will lead to modular reductions however, they won't be harmful. We can rewrite c_{n-1} as

$$c_{n-1} = (\sum b_{i_0}) \cdot 0 + (\sum b_{i_1}) \cdot 1 + (\sum b_{i_2}) \cdot 2 + \cdots + (\sum b_{i_{p-2}}) \cdot p - 2 + (\sum b_{i_{p-1}}) \cdot p - 1$$

, where, $b_{i_k} = b_i$ for $a_i = k$.

We can rewrite this as:

$$c_{n-1} = c'_{n-1} + c_\Delta \cdot p$$

, where, c'_{n-1} is the coefficient when the the secret polynomial is not converted to signed representation, i.e,

c'_{n-1}

$$= (\sum b_{i_0}) \cdot 0 + (\sum b_{i_1}) \cdot 1 + (\sum b_{i_2}) \cdot 2 + \cdots + (\sum b_{i_{-2}}) \cdot -2 + (\sum b_{i_{-1}}) \cdot -1$$

So when we take c_{n-1} mod p we will get c'_{n-1} and just get rid of the extra factor c_Δ. Therefore, a small 24 bit prime number greater than 10485760 is sufficient for Saber.

REFERENCES

[1] J. Proos and C. Zalka, "Shor's discrete logarithm quantum algorithm for elliptic curves," *Quantum Info. Comput.*, vol. 3, no. 4, p. 317–344, Jul. 2003.

[2] F. Arute1, K. Arya, R. Babbush, D. Bacon1, J. C. Bardin, R. Barends, R. Biswas, S. Boixo, F. G. S. L. Brandao, D. A. Buell, B. Burkett, Y. Chen, Z. Chen, B. Chiaro, R. Collins, W. Courtney, A. Dunsworth, E. Farhi, B. Foxen, A. Fowler, C. Gidney, M. Giustina, R. Graff, K. Guerin, S. Habegger, M. P. Harrigan, M. J. Hartmann, A. Ho, M. Hoffmann, T. Huang, T. S. Humble, S. V. Isakov, E. Jeffrey, Z. Jiang, D. Kafri, K. Kechedzhi, J. Kelly, P. V. Klimov, S. Knysh, A. Korotkov, F. Kostritsa, D. Landhuis, M. Lindmark, E. Lucero, D. Lyakh, S. Mandrà, J. R. McClean, M. McEwen, A. Megrant, X. Mi, K. Michielsen, M. Mohseni, J. Mutus, O. Naaman, M. Neeley, C. Neill, M. Y. Niu, E. Ostby, A. Petukhov, J. C. Platt, C. Quintana, E. G. Rieffel, P. Roushan, N. C. Rubin, D. Sank, K. J. S. ans Vadim Smelyanskiy, K. J. Sung, M. D. Trevithick, A. Vainsencher, B. Villalonga, T. White, Z. J. Yao, P. Yeh, A. Zalcman, H. Neven, and J. M. Martinis, "Quantum supremacy using a programmable superconducting processor," Nature, 2019, https://doi.org/10.1038/s41586-019-1666-5.

[3] D. J. Bernstein, T. Chou, T. Lange, I. von Maurich, R. Misoczki, R. Niederhagen, E. Persichetti, C. Peters, P. Schwabe, N. Sendrier *et al.*, "Classic mceliece: conservative code-based cryptography," Submission to the NIST Post-Quantum Standardization project, 2017.

[4] 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.

[5] C. Chen, O. Danba, J. Hoffstein, A. Hulsing, J. Rijneveld, J. M. Schanck, P. Schwabe, W. Whyte, Z. Zhang, T. Saito, T. Yamakawa, and K. Xagawa, "NTRU," Proposal to NIST PQC Standardization, Round3, 2021, https://csrc.nist.gov/Projects/post-quantum-cryptography/round-3-submissions.

[6] J.-P. D'Anvers, A. Karmakar, S. S. Roy, F. Vercauteren, J. M. B. Mera, M. V. Beirendonck, and A. Basso, "SABER," Proposal to NIST PQC Standardization, Round3, 2021, https://csrc.nist.gov/Projects/post-quantum-cryptography/round-3-submissions.

[7] 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.

[8] T. Prest, P.-A. Fouque, J. Hoffstein, P. Kirchner, V. Lyubashevsky, T. Pornin, T. Ricosset, G. Seiler, W. Whyte, and Z. Zhang, "FALCON," Proposal to NIST PQC Standardization, Round3, 2021, https://csrc.nist.gov/Projects/post-quantum-cryptography/round-3-submissions.

[9] J. Ding, M.-S. Chen, A. Petzoldt, D. Schmidt, B.-Y. Yang, M. Kannwischer, and J. Patarin, "FALCON," Proposal to NIST PQC Standardization, Round3, 2021, https://csrc.nist.gov/Projects/post-quantum-cryptography/round-3-submissions.

[10] G. Alagic, J. Alperin-Sheriff, D. Apon, D. Cooper, Q. Dang, J. Kelsey, Y.-K. Liu, C. Miller, D. Moody, R. Peralta, R. Perlner, A. Robinson, and D. Smith-Tone, "Status Report on the Second Round of the NIST Post-Quantum Cryptography Standardization Process," NISTIR 8309, 2020, https://doi.org/10.6028/NIST.IR.8309.

[11] U. Banerjee, T. S. Ukyab, and A. P. Chandrakasan, "Sapphire: A configurable crypto-processor for post-quantum lattice-based protocols," *IACR Trans. Cryptogr. Hardw. Embed. Syst.*, vol. 2019, no. 4, pp. 17–61, 2019. [Online]. Available: https://doi.org/10.13154/tches.v2019.i4.17-61

[12] T. Fritzmann, G. Sigl, and J. Sepúlveda, "Risq-v: Tightly coupled risc-v accelerators for post-quantum cryptography," *IACR Transactions on Cryptographic Hardware and Embedded Systems*, vol. 2020, no. 4, p. 239–280, Aug. 2020. [Online]. Available: https://tches.iacr.org/index.php/TCHES/article/view/8683

[13] C.-M. M. Chung, V. Hwang, M. J. Kannwischer, G. Seiler, C.-J. Shih, and B.-Y. Yang, "Ntt multiplication for ntt-unfriendly rings: New speed records for saber and ntru on cortex-m4 and avx2," *IACR Transactions on Cryptographic Hardware and Embedded Systems*, vol. 2021, no. 2, p. 159–188, Feb. 2021. [Online]. Available: https://tches.iacr.org/index.php/TCHES/article/view/8791

[14] J.-P. D'Anvers, A. Karmakar, S. S. Roy, and F. Vercauteren, "SABER," Proposal to NIST PQC Standardization, Round2, 2019, https://csrc.nist.gov/Projects/Post-Quantum-Cryptography/round-2-submissions.

[15] V. Lyubashevsky, "Fiat-shamir with aborts: Applications to lattice and factoring-based signatures," in *International Conference on the Theory and Application of Cryptology and Information Security*. Springer, 2009, pp. 598–616.

[16] National Institute of Standards and Technology. 2015., "SHA-3 standard: Permutation-Based Hash and Extendable-Output Functions," FIPS PUB 202, 2015.

[17] J. Bos, L. Ducas, E. Kiltz, T. Lepoint, V. Lyubashevsky, J. M. Schanck, P. Schwabe, G. Seiler, and D. Stehlé, "Crystals-kyber: a cca-secure module-lattice-based kem," in *2018 IEEE EuroS&P*. IEEE, 2018, pp. 353–367.

[18] A. A. Karatsuba and Y. P. Ofman, "Multiplication of many-digital numbers by automatic computers," *Doklady Akademii Nauk*, vol. 145, no. 2, pp. 293–294, 1962.

[19] A. L. Toom, "The complexity of a scheme of functional elements realizing the multiplication of integers," *Soviet Mathematics Doklady*, vol. 3, no. 4, pp. 714–716, 1963.

[20] D. Knuth, *The Art of Computer Programming, Volume 2. Third Edition*. Addison-Wesley, 1997.

[21] 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

[22] C. M. Chung, V. Hwang, M. J. Kannwischer, G. Seiler, C. Shih, and B. Yang, "NTT multiplication for ntt-unfriendly rings new speed records for saber and NTRU on cortex-m4 and AVX2," *IACR Trans. Cryptogr. Hardw. Embed. Syst.*, vol. 2021, no. 2, pp. 159–188, 2021. [Online]. Available: https://doi.org/10.46586/tches.v2021.i2.159-188

[23] 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

[24] 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

[25] F. Yaman, A. C. Mert, E. Öztürk, and E. Savas, "A hardware accelerator for polynomial multiplication operation of CRYSTALS-KYBER PQC scheme," in *Design, Automation & Test in Europe Conference & Exhibition, DATE 2021, Grenoble, France, February 1-5, 2021*. IEEE, 2021, pp. 1020–1025. [Online]. Available: https://doi.org/10.23919/DATE51398.2021.9474139

[26] S. S. Roy, F. Vercauteren, N. Mentens, D. D. Chen, and I. Verbauwhede, "Compact ring-lwe cryptoprocessor," in *Cryptographic Hardware and Embedded Systems – CHES 2014*, L. Batina and M. Robshaw, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2014, pp. 371–391.

[27] K. Team, "Keccak in VHDL: High-speed core," https://keccak.team/hardware.html, Accessed on November 2019.

[28] K. Gaj, "Implementation and Benchmarking of Round 2 Candidates in the NIST Post-Quantum Cryptography Standardization Process Using FPGAs," *NIST PQC Round 3 Seminars*, October 2020, https://csrc.nist.gov/projects/post-quantum-cryptography/workshops-and-timeline/round-3-seminars.

[29] 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.*, vol. 2020, p. 795, 2020. [Online]. Available: https://eprint.iacr.org/2020/795

[30] T. Fritzmann, M. Van Beirendonck, D. B. Roy, P. Karl, T. Schamberger, I. Verbauwhede, and G. Sigl, "Masked accelerators and instruction set extensions for post-quantum cryptography." *IACR Cryptol. ePrint Arch.*, vol. 2021, p. 479, 2021.

[31] 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

[32] 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

[33] 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

[34] D. Soni, K. Basu, M. Nabeel, and R. Karri, "A hardware evaluation study of nist post-quantum cryptographic signature schemes," in *Second PQC Standardization Conference*. NIST, 2019.

[35] V. B. Dang, F. Farahmand, M. Andrzejczak, and K. Gaj, "Implementing and benchmarking three lattice-based post-quantum cryptography algorithms using software/hardware codesign," in *2019 International Conference on Field-Programmable Technology (ICFPT)*, 2019, pp. 206–214.

[36] P. He, C.-Y. Lee, and J. Xie, "Compact coprocessor for kem saber: Novel scalable matrix originated processing."

[37] A. Abdulgadir, K. Mohajerani, V. B. Dang, J.-P. Kaps, and K. Gaj, "Lightweight implementation of saber resistant against side-channel attacks."

[38] J. Maria Bermudo Mera, F. Turan, A. Karmakar, S. Sinha Roy, and I. Verbauwhede, "Compact domain-specific co-processor for accelerating module lattice-based kem," in *2020 57th ACM/IEEE Design Automation Conference (DAC)*, 2020, pp. 1–6.

[39] M. Imran, F. Almeida, J. Raik, A. Basso, S. S. Roy, and S. Pagliarini, "Design space exploration of saber in 65nm asic," 2021.

[40] Y. Zhu, M. Zhu, B. Yang, W. Zhu, C. Deng, C. Chen, S. Wei, and L. Liu, "Lwrpro: An energy-efficient configurable crypto-processor for module-lwr," *IEEE Transactions on Circuits and Systems I: Regular Papers*, vol. 68, no. 3, pp. 1146–1159, 2021.