Efficient Polynomial Arithmetic Architecture for Saber and Dilithium

Aikata
aikata@iaik.tugraz.at

Institute for Applied Information
Processing and Communications (IAIK)
Graz University of Technology
Inffeldgasse 16a
8010 Graz, Austria

Master Thesis

Supervisor: Sujoy Sinha Roy

December, 2021

AFFIDAVIT

I declare that I have authored this thesis independently, that I have not used other than the declared sources/resources, and that I have explicitly indicated all material which has been quoted either literally or by content from the sources used. The text document uploaded to TUGRAZonline is identical to the present master‘s thesis.

Date: 15.12.2021

Signature: 

Acknowledgements

I would like to acknowledge all my colleagues and researchers. This thesis would not have been possible without their support and collaboration. I would like to thank my supervisor, Prof. Dr. Sujoy Sinha Roy. His friendly guidance and expert advice have been invaluable throughout all stages of the work. I would also wish to express my gratitude to Post. Doc. Dr. Ahmet Can Mert for extended discussions and valuable suggestions that have contributed greatly to the improvement of the thesis. This work would not be in its best stage without the several brainstorming sessions with them.

Special thanks are due to my parents for their continuous support and understanding throughout the research.

Thank You

Abstract

The tech giants of the world are competing for quantum supremacy- the fight for who will make the world's first strong quantum computer. In 2003, Shor's algorithm showed that these quantum computers can be used to break the public key schemes based on the hard problems of integer factorization and discrete logarithmic problems like RSA and Elliptic Curve Cryptography. The cut-throat competition and the significant progress made towards making the Quantum computers, has led the researchers to believe that it is not very far from being a reality. This implies a need to come up with secure public-key cryptosystems to avoid any breach in the confidentiality and integrity of the communications.

Realizing this the National Institute of Standards and Technology (NIST) launched a project in 2019- Post-Quantum Cryptography (PQC) Standardization for development of key encapsulation and public key encryption scheme and digital signature algorithm. This competition is currently in its final stage. Several designers have presented implementations for the candidate schemes. We realized that it is important to design a compact cryptoprocessor that can efficiently implement both kinds of schemes, the major bottleneck being a polynomial arithmetic unit and other comparatively simpler building blocks.

In this thesis, we aim to design an efficient yet lightweight polynomial arithmetic unit for the two NIST's PQC standardization project finalists- Saber (key encapsulation and public key encryption scheme) and Dilithium(digital signature scheme). We exploit the synergies between these two lattice-based designs and overcome the differences between the two schemes to successfully implement a flexible, compact, and lightweight architecture capable of performing all the polynomial arithmetic operations for all the variants of Saber and Dilithium based on different security levels. The proposed architecture is an instruction set architecture. The implementation is done on Xilinx Ultrascale+ ZCU102 and the polynomial arithmetic unit alone consumes 2,460 LUTs, 1,084 FFs, 4 DSPs, and 1 BRAM. One NTT/INTT operation takes 522 clock cycles and one pointwise addition/subtraction/multiplication requires 138 clock cycles.

Keywords: Polynomial Arithmetic, Dilithium, Saber, Hardware Implementation, Lattice-based Cryptography, Post-quantum cryptography

Kurzfassung

Die Technologiegiganten der Welt konkurrieren um die Quantenvorherrschaft – ein heißer Kampf darum, wer den ersten starken Quantencomputer der Welt bauen wird. Im Jahr 2003 zeigte der Algorithmus von Shor, dass diese Quantencomputer dazu missbraucht werden können, um die öffentlichen Schlüsselschemata basierend auf der Problematik der ganzzahligen Faktorisierung und den diskreten logarithmischen Problemen wie RSA und Elliptische Kurven Kryptografie zu knacken. Der starke Konkurrenzkampf und die erheblichen Fortschritte bei der Entwicklung der Quantencomputer lassen die Forscher glauben, dass dies nicht mehr weit von der Realität entfernt ist. Dies impliziert die Notwendigkeit, sichere Krypto-Systeme mit öffentlichem Schlüssel zu entwickeln, um jede Verletzung der Vertraulichkeit und Integrität der Kommunikation zu vermeiden.

Als das Nationale Institute für Standards and Technologien (NIST) sich der Problematik bewusst wurde, startete es 2019 ein Projekt – Post-Quanten Kryptographie (PQK) Standardisierung zur Entwicklung der Schlüsselkapselung, des Verschlüsselungsschemas mit öffentlichem Schlüssel sowie des digitalen Signaturalgorithmus. Dieser Wettbewerb befindet sich derzeit in der finalen Phase. Mehrere Designer haben Implementierungen für die verschiedenen Kandidatenschemata präsentiert. Wir haben erkannt, dass es wichtig ist, einen kompakten Krypto-Prozessor zu entwickeln, der beide Arten von Schemata effizient implementieren kann, wobei die größten Engstellen die polynomische Arithmetik-Einheit und andere vergleichsweise einfachere Bausteine sind.

In dieser Arbeit zielen wir darauf ab, eine effiziente und dennoch leichte polynomiale Recheneinheit für die beiden Finalisten des PQC-Standardisierungsprojekts des NIST - Saber (Schlüsselkapselung und Verschlüsselungsschema mit öffentlichem Schlüssel) und Dilithium (digitales Signaturschema) - zu entwerfen. Wir nutzen die Synergien zwischen diesen beiden gitterbasierten Designs und überwinden die Unterschiede zwischen den beiden Schemata, um erfolgreich eine flexible, kompakte und leichtgewichtige Architektur zu implementieren, die in der Lage ist, alle polynomialen arithmetischen Operationen für alle Varianten von Saber und Dilithium auf unterschiedlichen Sicherheitsebenen durchzuführen. Die vorgeschlagene Architektur ist eine Befehlssatzarchitektur. Die Implementierung erfolgt auf Xilinx Ultrascale+ ZCU102 und die polynomiale Recheneinheit allein verbraucht 2.460 LUTs, 1.084 FFs, 4 DSPs und 1 BRAM. Eine NTT/INTT-Operation benötigt 512 Taktzyklen und eine punktweise Addition/Subtraktion/Multiplikation erfordert 138 Taktzyklen.
Stichwörter: Polynomische Arithmetik, Dilithium, Saber, Hardwareimplementierung, gitterbasierte Kryptographie, Post-Quanten Kryptographie

Contents

List of Figures

List of Tables

List of Algorithms

Chapter 1

Introduction

In the year 2003, Peter Williston Shor proposed the famous Shor's scheme [59] which could solve the integer factorization and discrete logarithm problems in polynomial time using quantum computers. Thus posing a direct threat to some of the most widely used public-key cryptosystems - RSA and Elliptic curve cryptography. The tech giants- IBM, Google, Amazon, etc., are already working towards making a powerful quantum computer for the past five years and have achieved quite great success. A problem that would take 10,000 years to be solved by a state-of-the-art supercomputer was solved in 200 seconds by Google's 'Sycamore' [15], a 54-bit quantum computer proposed in October 2019. Although they are not yet powerful enough the break the RSA and Elliptic curve-based public-key cryptography, but owing to the super fast design progress that is being made, in the near future the present day cryptographic schemes won't be secure anymore. This would adversely affect the confidentiality and integrity of the communications.

Fearing the inevitable, the National Institute of Standards and Technology (NIST) launched a project in 2019- 'Post-Quantum Cryptography (PQC) Standardization' to initiate the development and standardization of the schemes which would be secure even after the advent of quantum computers and can function with the existing communications protocols and networks. NIST also shared a report discussing the status of quantum computing and post-quantum cryptography and press on the design being crypto-agile. Researchers from all-over-the world contributed to this project with various design strategies using the algorithms which would remain unbreakable by the quantum computers. After the first round, 17 public-key encryption and key-establishment algorithms, and 9 digital signature candidates moved to the second round. On July 2020 this competition moved to it's final stage with four finalists in the public-key encryption (PKE) or key encapsulation mechanism (KEM) category: Classic-McEliece [21], CRYSTALS-Kyber [65], NTRU [25], and Saber [34]; and three finalists in the digital signature category : CRYSTALS-Dilithium [16], Falcon [58], and Rainbow [35]. Amongst the finalists, most of the schemes are lattice-based constructions except Classic-McEliece and Rainbow which are code-based and oil-vinegar-based constructions.

After the second round, the project report [13] published by NIST states that one of the lattice-based schemes will be chosen as the winner and eventually standard-

ized. From the report and the ongoing discussion on the standardization project more and more emphasis is placed on improving the implementations and side-channel analysis of these finalists, since it is a challenging task to efficiently implement these schemes on existing platforms which greatly vary in terms of memory and performance constraints. Therefore, in this thesis we aim to improve the implementation aspects of these schemes and realize a unified polynomial arithmetic unit that is compact, lightweight, and yet high-speed.

After the first round of NIST's PQC standardization itself in November 2017, various hardware implementations of the PQC candidates started showing up in 2018. While most of them focused on giving the results for one particular scheme, some of them also gave results for unified cryptoprocessors which provided support for 2 or more lattice-based PQC schemes. In 2019 Sapphire[17] was proposed, which could execute multiple lattice-based PQC schemes, namely Frodo, NewHope, qTESLA, CRYSTALS-Kyber, and CRYSTALS-Dilithium. However, it does not support the most latest scheme specifications of the two finalists CRYSTALS-Kyber [65] and CRYSTALS-Dilithium [16]. Then in 2020 a tightly-coupled RISC-V extension[41], 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. For Saber, due to limited hardware support, the cycle count is arguably not superior to optimized software implementations on an ARM Cortex M4 micro controller [27]. No hardware support for the lattice-based signature schemes Dilithium or Falcon is available in RISQ-V.

From all the previous implementations of these schemes, we conclude that polynomial arithmetic and Keccak based hash and pseudo-random number generation functions are most expensive and complex to implement. There are many efficient implementations of Keccak based on different requirements which exist in literature and because the functionality of the module stays the same for different designs, not many changes are required in the existing implementations of Keccak. Therefore, in this thesis, we propose a compact and fast polynomial arithmetic architecture for performing the multiplication, addition, and subtraction of two polynomials. We realized this unified architecture completely in hardware for the lattice-based finalist PKE/KEM candidate Saber [34] and the signature candidate Dilithium [16].

In the coming up sections we list the concrete contributions made in the thesis and the organization of the thesis.

1.1 Thesis Contribution

In this thesis the following contributions are made:

- The most area-consuming components in the PQC scheme are Keccak based hash and pseudo-random number generators, and Polynomial Arithmetic unit- for performing multiplication, addition, and subtraction of Polynomials. A unified Polynomial Arithmetic Architecture is designed for the two NIST finalists: Saber and Dilithium. Special care is taken to ensure that the architecture is efficient, scalable, and lightweight.

- The NIST PQC schemes have also provided different parameters for different security levels. The proposed unified architecture supports all the different security variants of Saber and Dilithium. Thus, making the architecture easily suitable for being plugged into any existing implementation of Saber or Dilithium or both.

- The most expensive components in the Polynomial Unit are the multiplication unit and modular reduction unit. We use a DSP multiplier for the multiplication unit. An efficient modular reduction unit is designed for three different 'special' prime candidates. This gives flexibility to the users to choose the design option based on the application requirements. Both the DSP multiplier and modular reduction unit are well-pipelined to achieve a higher clock frequency.

- By designing an efficient and lightweight unified Polynomial Arithmetic Architecture for a lattice-based Key Encapsulation scheme(Saber) and Signature scheme(Dilithium), we move a step forward towards designing a unified cryptoprocessor which supports both types of schemes, and multiple such schemes like Kyber, NTRU, etc..

1.2 Organization of the Thesis

Chapters 2-4 provide the background on the design of the two schemes, and various design methodologies existing in the literature. We discuss our design strategy and implementation details, along with results and future scope of the work, in Chapters 5-8. In Chapter 9 we conclude our work.

Chapter 2 introduces the design of Saber and Dilithium. We discuss in brief various building blocks of the two schemes along with algorithms. We also list the different parameters required for different security levels. The algorithms showcase the importance of a polynomial arithmetic architecture which can support different parameters.

We describe various methods to implement the polynomial multiplication unit, which is a major component of a polynomial arithmetic unit, in Chapter 3. The different methods used for the implementation of the PQC schemes and their algorithms are discussed. The run-time complexity is given for each of the algorithms.

For this architecture, we use the NTT-based multiplication method as the common method for both schemes.

In Chapter 4 - Modular Reduction methods, we list various efficient modular reduction methods and algorithms. Some of these are used in the official implementations of the schemes. From this section, we can conclude that the 'special' primes based reduction method is the best in terms of both time and area consumption. Therefore, our design also uses this efficient method for an efficient and lightweight architecture.

In Chapter 5, we discuss the design decisions we took towards the design of the unified polynomial arithmetic architecture. We discuss the different requirements of Saber and Dilithium and how we deal with them to make a common module for different parameters sets.

The implementation strategies are discussed in Chapter 6. We start from the naive methodologies and progress towards better solutions. We provide the algorithms and architecture diagrams for the modular reduction unit and arithmetic unit. We also discusses various trade-offs we considered while implementing the design.

In Chapter 7 we discuss the results of our implementation in hardware and in Chapter 8 we discuss the future scope of the work.

We finally conclude the work in Chapter 9. Some abbreviations, algorithms, and figures used in the thesis are given in Chapter A- Appendix. The end of the thesis has references to the works which greatly helped in designing the architecture.

Chapter 2

PQC schemes

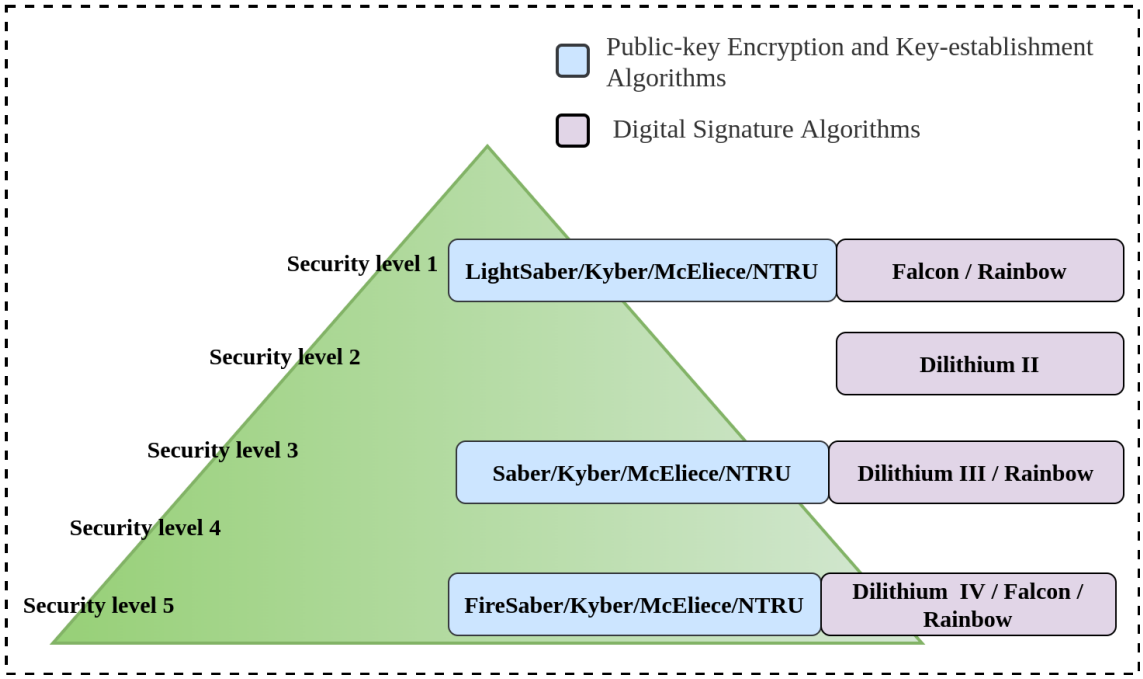

Figure 2.1: NIST finalists grouped according to security levels supported

Figure 2.1 shows the finalist candidates and different security levels for which they can be parameterized. In this chapter, we discuss the design specifications of the two lattice-based schemes: Saber(public-key encryption and key-establishment algorithms) and Dilithium (Digital Signature scheme). From the figure, it can be seen that both Saber and Dilithium provide support for at least 3 security levels out of which 2 are common, which is the most between any other two sets of key-encapsulation and signature schemes. Saber and Dilithium are both lattice-based schemes and are based on similar design principles and we aim to exploit these similarities to make a very efficient and lightweight hardware polynomial arithmetic unit that supports all the variants of both the schemes.

2.1 Saber scheme description

The finalist candidate for Public-key Encryption and Key-establishment algorithms-Saber [34] is an IND-CCA secure scheme. Its security relies on the hardness of Module learning with rounding problem(MLWR), which is presumed to be computationally infeasible, both on classical and quantum computers. Saber uses this hard problem in conjunction with two moduli p and q, both of them being powers-of-two, to construct a Chosen Ciphertext Attack (CCA) secure key encapsulation mechanism (KEM).

Based on the different security levels, as shown in Fig.2.1, it has three variants: LightSaber, Saber, and FireSaber targeting low, medium, and high security levels respectively. Even though they are for different security levels, they 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 difference between the three variants lies in their use of different module-dimensions and secret-distributions. These different parameters for both public key encryption and key encapsulation are listed in Tab. 2.1 and Tab. 2.2.

Security level	1(LightSaber)	3(Saber)	5(FireSaber)
Failure Probability	2^{-120}	2^{-136}	2^{-165}
pk (Bytes)	672	992	1312
sk (Bytes)	832	1248	1664
ct (Bytes)	736	1088	1472
l	2	3	4
n	256	256	256
q	2^{13}	2^{13}	2^{13}
p	2^{10}	2^{10}	2^{10}
T	2^3	2^4	2^6
μ	10	8	6

Table 2.1: Saber.PKE Parameters

Security level	1(LightSaber)	3(Saber)	5(FireSaber)
Failure Probability	2^{-120}	2^{-136}	2^{-165}
pk (Bytes)	672	992	1312
sk (Bytes)	1568	2304	3040
ct (Bytes)	736	1088	1472
l	2	3	4
n	256	256	256
q	2^{13}	2^{13}	2^{13}
p	2^{10}	2^{10}	2^{10}
T	2^3	2^4	2^6
μ	10	8	6

Table 2.2: Saber.KEM Parameters

Here, n is the degree of the polynomial ring, and l is the dimension of the matrix. Increasing l increases the security level but lowers the correctness. p, q are the moduli 2^{10} and 2^{13} as discussed earlier, which can be seen from the table, stay the same for different security variants. T is the parameter such that $\epsilon_q > \epsilon_p > \epsilon_T$ where, $p = 2^{\epsilon_p}$,$q = 2^{\epsilon_q}$, and $T = 2^{\epsilon_T}$. So $T|p|q$. Higher T implies lower security but higher correctness. μ is the range in which the coefficient of the secret polynomial is sampled using a binomial distribution. Higher μ implies higher security but lower correctness. Next, we describe the design of the Saber scheme. It is a two-step process. The first step is to establish a public key encryption scheme and it is IND-CPA secure.

The public key encryption scheme- `Saber.PKE` consists of three algorithms:

- `Saber.PKE.KeyGeneration`() specified in Alg.1

Algorithm 1: `Saber.PKE.KeyGen`() [33]

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

- `Saber.PKE.Encryption`() specified in Alg.2

Algorithm 2: `Saber.PKE.Enc`$(pk = (seed_{\boldsymbol{A}}, \boldsymbol{b}), m \in R_2; r)$ [33]

$\boldsymbol{A} = \mathtt{gen}(\mathrm{seed}_{\boldsymbol{A}}) \in \mathcal{R}_q^{l \times l}$
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 $c := (c_m, \boldsymbol{b'})$

- `Saber.PKE.Decryption`() specified in Alg.3

Algorithm 3: `Saber.PKE.Dec`$(sk = \boldsymbol{s}, c = (c_m, \boldsymbol{b'}))$ [33]

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

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-CCA algorithms used in Saber-KEM are:

- `Saber.KEM.KeyGeneration`() as specified in Alg. 4.

Algorithm 4: `Saber.KEM.KeyGen`() [33]

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

- `Saber.KEM.Encapsulation`()) as specified in Alg. 5.

Algorithm 5: `Saber.KEM.Encaps`$(pk = (seed_{\boldsymbol{A}}, \boldsymbol{b}))$ [33]

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

- `Saber.KEM.Decaps`() as specified in Alg. 6.

Algorithm 6: `Saber.KEM.Decaps`$(sk = (\boldsymbol{s}, z, pkh), pk = (seed_{\boldsymbol{A}}, \boldsymbol{b}), c)$ [33]

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

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 pseudo-random 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.

2.2 Dilithium scheme description

The digital signature finalist candidate Dilithium [16] is built upon the well-known Fiat-Shamir with aborts framework [52]. Its security is based on the computational hardness of the Module Learning With Errors (MLWE) and Module Short Integer Solution (MSIS) problems, i.e., finding short vectors in lattices.

As mentioned earlier Dillithium also provides support for three different security levels. 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 [16]. The parameters are listen in Table 2.3. 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.

NIST Security Level	2	3	5
q [modulus]	8380417	8380417	8380417
d [dropped bits from t]	13	13	13
τ [# of $\pm$ 1's in c]	39	49	60
challenge entropy	192	225	257
γ_1 [**y** coefficient range]	2^{17}	2^{19}	2^{19}
γ_2 [low-order rounding range]	(q-1)/88	(q-1)/32	(q-1)/32
(k,l) [dimensions of A]	(4, 4)	(6, 5)	(8, 7)
η [secret key range]	2	4	2
β [$\tau \cdot \eta$]	78	196	120
ω [max. # of 1's in the hint h]	80	55	75

Table 2.3: Dilithium Parameters

Dilithium signature scheme has the following three basic procedures as discussed in [11]:

- **Key Generation**: 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. The random polynomial-matrix $\boldsymbol{A} \in R_q^{k\times l}$ is generated by expanding a 256-bit seed ρ. After computing the vector $\boldsymbol{t} \in R_q^k$, it is split into two vectors $\boldsymbol{t}_0$ and $\boldsymbol{t}_1$ in line 6 using $\texttt{Power2Round}_q()$. Finally, tr is computed in line 7 by applying $\texttt{CRH}()$ to the concatenated string $\rho \parallel \boldsymbol{t}_1$. The key generation as shown in Algorithm 7, generates a $k \times l$ matrix $\mathbf{A}$ each of whose entries is a polynomial in the ring $R_q = Z_q[X]/(X_n + 1)$. Afterwards, the algorithm samples random secret key vectors $\mathbf{s}_1$ and $\mathbf{s}_2$. Each coefficient of these vectors is an element of R_q with small coefficients of size at most η. Finally, the second part of the public key is computed as $\mathbf{t} = \mathbf{A}\mathbf{s}_2 + \mathbf{s}_2$. All algebraic operations in this scheme are assumed to be over the polynomial ring R_q .

- **Verfication** : The verification operation (Alg. 8) is cheaper than key-generation and signing. It accepts a signature if all the three conditions specified in line 5 are satisfied.

Algorithm 7: `Dilithium.Gen()` [16]

1 $\zeta \leftarrow \{0,1\}^{256}$
2 $(\rho, \varsigma, K) \in \{0,1\}^{256\times 4} := \mathcal{H}(\zeta)$ ▷ $\mathcal{H}$ 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** $(pk = (\rho, \boldsymbol{t}_1), sk = (\rho, K, tr, \boldsymbol{s}_1, \boldsymbol{s}_2, \boldsymbol{t}_0))$

Algorithm 8: `Dilithium.Verify`$(pk, M, \sigma = (\boldsymbol{z}, \boldsymbol{h}, \tilde{c}))$ [16]

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

- **Signing procedure**: The signing operation (Alg. 9) 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 to start from line 14 to 19. Signing as shown in Algorithm 9, generates a masking vector of polynomials $\mathbf{y}$ with coefficients less than γ_1 . The signer then computes $\mathbf{Ay}$ and sets $\mathbf{w}_1$ to be the "high-order" bits of the coefficients in this vector. The output c is a polynomial in R_q with exactly $\tau\pm1$'s and the rest 0's.

These algorithms use various building blocks which are described below:

- `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 $\boldsymbol{s}_2 \in S_\eta^\ell \times S_\eta^k$. For each polynomial the seed ς and a 16-bit nonce are fed

Algorithm 9: $\texttt{Dilithium.Sign}(sk, M)$ [16]

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

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 $\mathbb{Z}_q$ 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 $r_1 = (r - r_0)/2^d$. This is shown in Alg. 10

Algorithm 10: $\texttt{Power2Round}_q(r, d)$ [16]

$r \leftarrow mod^+ q$
$r_0 \leftarrow mod^{\pm} q$
return $((r - r_0)/2^d, r_0)$

- $\texttt{HighBits}_q()$ and $\texttt{LowBits}_q()$: Let α be a divisor of $q - 1$. The function $\texttt{Decompose}_q()$ as shown in Alg. 11 is defined in the same way as $\texttt{Power2Round}()$ with α replacing 2^d in $\texttt{Power2Round}()$. Thus $\texttt{Decompose}_q()$ breaks an input

$r \in \mathbb{Z}_q$ 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 $\texttt{LowBits}_q()$. This is shown in Alg. 12 and Alg. 13.

Algorithm 11: $\texttt{Decompose}_q(r, \alpha)$ [16]

$r \leftarrow mod^+ q$
$r_0 \leftarrow mod^{\pm} \alpha$
if $r - r_0 = q - 1$ **then**
  $r_1 \leftarrow 0; r_0 \leftarrow r_0 - 1$
else
  $r_1 \leftarrow (r - r_0)/\alpha$
return (r_1, r_0)

Algorithm 12: $\texttt{HighBits}_q(r, \alpha)$ [16]

$(r_1, r_0) \leftarrow Decompose_q(r, \alpha)$
return r_1

Algorithm 13: $\texttt{LowBits}_q(r, \alpha)$ [16]

$(r_1, r_0) \leftarrow Decompose_q(r, \alpha)$
return r_0

- $\texttt{MakeHint}_q()$: It uses $\texttt{Decompose}_q()$ to produce a hint $\boldsymbol{h}$. This is shown in Alg. 14.

Algorithm 14: $\texttt{MakeHint}_q(z, r, \alpha)$ [16]

$r_1 \leftarrow \texttt{HighBits}_q(r, \alpha)$
$v_1 \leftarrow \texttt{HighBits}_q(r + z, \alpha)$
return $[[r_1 \neq v_1]]$

- $\texttt{UseHint}_q()$: It use the hint $\boldsymbol{h}$ produced by $\texttt{MakeHint}_q()$ to recover the high-bits. This is shown in Alg.15.

Algorithm 15: $\texttt{UseHint}_q(h, r, \alpha)$ [16]

$m \leftarrow (q - 1)/\alpha$
$(r_1, r_0) \leftarrow \texttt{Decompose}_q(r, \alpha)$
if $h = 1$ *and* $r_0 > 0$ **then**
  return $(r_1 + 1) mod^+ m$
if $h = 1$ *and* $r_0 \leq 0$ **then**
  return $(r_1 - 1) mod^+ m$

- $\texttt{CRH}()$: This is a collision resistant hash function which utilizes 384 bits of the output of SHAKE-256.

- $\texttt{SampleInBall}$: It uses SHAKE-256 to expand μ and bit-packed $\boldsymbol{w}_1$ as the seed. Out of the first 64 output bits, 60 bits are taken as random sign bits and the

remaining 4 bits are simply discarded. Then in every next iteration, rejection sampling is performed on every next byte of the squeeze output and out of 256 values, only τ values are non-zero with value either $+1$ or -1 depending on the sign bit. This is described in Alg. 16.

Algorithm 16: `SampleInBall`(ρ) [16]

```
initialize c = c_0 c_1 · s c_255 = 00···0
for i = 256 - τ to 255 do
    j ← {0, 1, ··· , i}
    s ← {0, 1}
    c_i = c_j
    c_j = (−1)^s
return c
```

- `ExpandMask`(): This function is used to generate the polynomial vector $\boldsymbol{y}$ used for deterministically generating the randomness of the signing procedure. It maps $\acute{\rho}||\kappa$ to $\boldsymbol{y} \in \tilde{S}^l_{\gamma_1}$. The squeeze output is broken into a sequence of positive integers in the range $\{0, \cdots, 2\gamma_1 - 1\}$ by breaking down the stream into chunks of 18 bits or 20 bits depending on the value of γ_1. These values are then subtracted from γ_1 and to obtain the integers comprising $\boldsymbol{y}$.

- `NTT`: Polynomial multiplications are performed using the Number Theoretic Transform (NTT) method.

From the algorithms, it is clear that a major portion of the computation is spent performing polynomial arithmetic computation. Therefore, it is very important to build an efficient polynomial arithmetic unit.

In the next chapter, we discuss various strategies to build a polynomial multiplier.

Chapter 3

Polynomial Multiplication

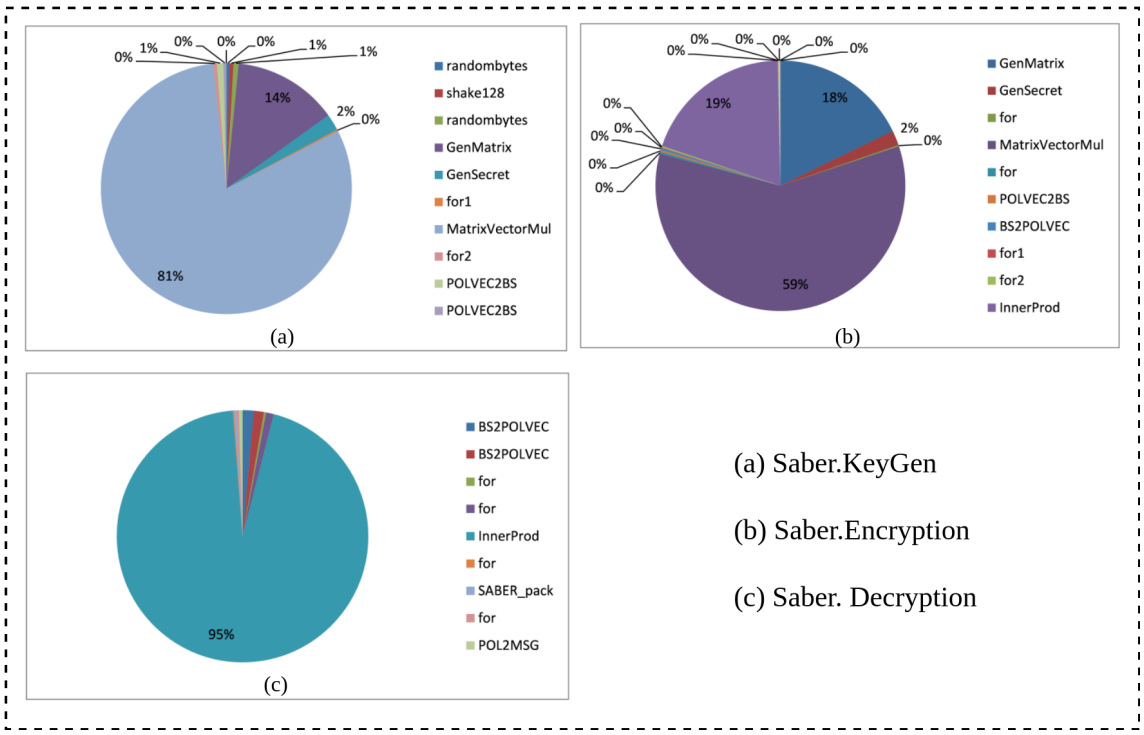

Figure 3.1: Function usage in Key Generation, Encryption, and Decryption for Saber [61]

Fig. 3.1 shows that a major part of the computation for Saber is spent on polynomial arithmetic operations, especially polynomial multiplication. The same can also be concluded for Dilithium from the description given in the previous chapter. Therefore, it is very important that we chose a good multiplication algorithm for our design after carefully examining all the multiplication algorithms that have been used for implementing the PQC schemes.

In this chapter, we describe all of these algorithms. Later in the Design and Implementation Chapters, discuss which one is the best for our architecture, based on our design goals.

3.1 SchoolBook multiplication

Schoolbook multiplication is the simplest and most basic method for multiplying two polynomials. Every polynomial coefficient is multiplied with every other coefficient and then accumulated. For e.g. Let's say we have two polynomials $2x^3 + x + 1$ and $2x^3 + x^2 + 2$ in $\mathbb{Z}_3[x]/(x^4 + 1)$ then the multiplication will be done in the following way: Since the polynomial multiplication is in $\mathbb{Z}_3[x]/(x^4 + 1)$, the coefficients are

			$2x^3$	$0x^2$	x	1
		$\times$	$2x^3$	x^2	$0x$	2
$4x^6$	$2x^5$	$2x^4$	$7x^3$	$2x^2$	$2x$	2
mod 3 $\downarrow$						
x^6	$2x^5$	$2x^4$	x^3	$2x^2$	$2x$	2
mod $(x^4 + 1)$ $\downarrow$						
			x^3	x^2		

always *mod* 3 and the resultant polynomial is *mod* $(x^4 + 1)$. For the latter we use the property $x^4 \equiv -1$, as the product of the two polynomials is a negative convolution. Therefore, accumulation is not a simple addition and instead has a signed bit $(-1)^{\lfloor (i+j)/n \rfloor}$, which is 1 if $i + j < n$, otherwise -1.

Algorithm 17: `SchoolBookMultiplication`(a, b, c) [51]

Input : a, b Polynomials in $\mathbb{Z}_p[x]/(x^n + 1)$
Output : $c = a \cdot b$ Polynomial in $\mathbb{Z}_p[x]/(x^n + 1)$
for *i =0 to n-1* **do**
  $sum \leftarrow 0$
  for *j =0 to n-1* **do**
    $ab \leftarrow (a[j] \cdot b[(i - j) mod\ n]) mod\ q$
    if *i<j* **then**
      $ab \leftarrow q - ab$
    $sum \leftarrow (sum + ab) mod\ q$
  $c[i] \leftarrow sum$
return c

This algorithm is simple has a time complexity of $\mathcal{O}(n^2)$ as can be seen in Equation 3.1. Even though the time complexity is high compared to the other schemes as discussed in the next sections, due to its simplicity it's used to design various efficient polynomial arithmetic architecture for post-quantum schemes [51],[63]. In [63] show implement a $\mathcal{O}(n)$ time-complexity schoolbook polynomial multiplier for

Saber by cascading n MAC units thus paying a high cost in terms of the area while saving time.

$$c = ab \ mod \ (x^n + 1) = \sum_{i=0}^{n-1} \sum_{j=0}^{n-1} (-1)^{\lfloor (i+j)/n \rfloor} a_i b_j x^{(i+j) mod \ n} \tag{3.1}$$

3.2 Karatsuba multiplication

Karatsuba is a divide-and-conquer algorithm which reduces the complexity of polynomial multiplication from $\mathcal{O}(n^2) \rightarrow \mathcal{O}(n^{log_2 3}) \approx \mathcal{O}(n^{1.58})$ when n is a power of two. It was proposed for single-degree polynomials in the beginning. It has 5 steps - Splitting, Evaluation, Pointwise multiplication, Interpolation, and Re-composition as shown in the Wikipedia page [8]. Here I give a simplified explanation for the two polynomials :

$$f(x) = f_1 x + f_0 \ and \ g(x) = g_1 x + g_0$$

Then the coefficients of multiplied resultant polynomial $h(x)$ can be expressed in terms of three multiplications as:

$$h_0 = f_0 \cdot g_0$$

$$h_1 = f_1 \cdot g_1$$

$$h_{0,1} = (f_0 + f_1) \cdot (g_0 + g_1)$$

Thus the multiplied polynomial can be written as:

$$h(x) = h_1 x^2 + (h_{0,1} - h_0 - h_1)x + h_0$$

For the same polynomials, schoolbook multiplication would take 4 multiplications and 1 addition. However, here we require 3 multiplications, 2 additions, and 2 subtractions. Note that for large integers or large numbers addition is many times faster and simpler to implement than multiplications which is a much more area-consuming circuit, so we can trade one multiplication operation for several addition operations and still save significant hardware cost and complexity. This was an example for a 1-degree polynomial, however, we can do the same with an $n-1$ degree polynomial. We can split the polynomial into two parts (n/2 coefficients each) and then use the same formulas above. We will have to do 3 n/2-degree schoolbook multiplications and 2 additions and 2 subtractions. Let $A(x)$ and $B(x)$ be two polynomials of degree n-1 such as:

$$A(x) = a_{n-1} x^{n-1} + \cdots + a_2 x^2 + a_1 x + a_0$$

$$B(x) = b_{n-1} x^{n-1} + \cdots + b_2 x^2 + b_1 x + b_0$$

These two polynomials are divided into two equal parts A_0, A_1 and B_0, B_1 such that $A = A_0 + x^{n/2} A_1$ and $B = B_0 + x^{n/2} B_1$:

$$A_0(x) = a_{n-1} x^{n-1} + \cdots + a_{(n/2)+2} x^{(n/2)+2} + a_{(n/2)+1} x^{(n/2)+1} + a_{n/2}$$

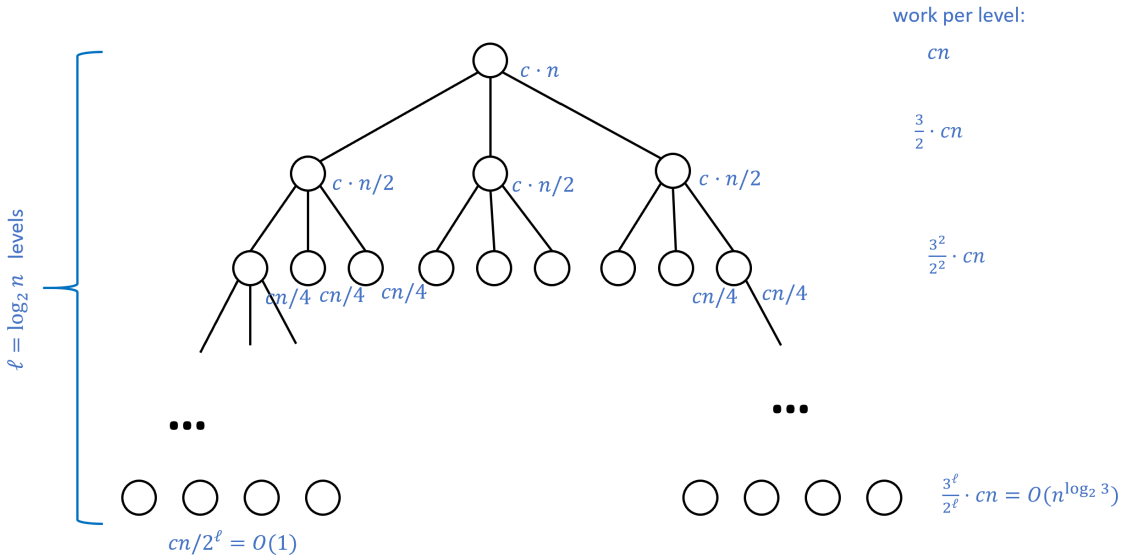

Figure 3.2: Karatsuba's algorithm reduces an n-bit multiplication to three $n/2$-bit multiplications, which in turn are reduced to nine $n/4$-bit multiplications and so on. We can represent the computational cost of all these multiplications in a 3-ary tree of depth $log_2 n$. where at the root the extra cost is cn operations, at the first level the extra cost is $c(n/2)$ operations, and at each of the 3^i nodes of level i, the extra cost is $c(n/2^i)$.[1]

$$B_0(x) = b_{n-1}x^{n-1} + \cdots + b_{(n/2)+2}x^{(n/2)+2} + b_{(n/2)+1}x^{(n/2)+1} + b_{n/2}$$
$$A_1(x) = a_{(n/2)-1}x^{(n/2)-1} + \cdots + a_2x^2 + a_1x + a_0$$
$$B_1(x) = b_{(n/2)-1}x^{(n/2)-1} + \cdots + b_2x^2 + b_1x + b_0$$

We then recursively keep on splitting these polynomials until we reach the 1 degree polynomials and then start multiplying the two parts from the lowest recursion level. If $T(n)$ is the time complexity to multiply $A(x)$ and $B(x)$ then:

$$T(n) = 3 \cdot T(n/2) + c \cdot n$$

$$T(n) = O(n^{log_2 3}) \approx 0(n^{1.58})$$

, where n is an even number representing degree of the two polynomials. This is called a *2-way Karatsuba Algorithm* because we are splitting it into 2 parts every time and then multiply them once we have 1-degree polynomials. The algorithm is given in Alg. 18. Similar optimization is possible for a *3-way Karatsuba Algorithm* as well. In this, we split it into 3 parts and break it down to degree 2 polynomials. In this case the time-complexity will be $\mathcal{O}(n^{1.63})$. A generalized *k-way Karatsuba Algorithm* can be constructed similarly. For polynomials of degree $n \cdot m$ we multiply polynomials of degree m coefficients and in the recursive step, the Karatsuba is applied to the polynomials with n coefficients and merged at the end as shown in [74].

Algorithm 18: `KaratsubaMultiplication`(a, b, c)

Input : a, b Polynomials of degree at most $n-1$ with $n = 2^k$ and $k \in \mathbb{N}$
Output : $c = a \cdot b$
if *n=1* **then**
  return $c \leftarrow a \cdot b$
$c_1 \leftarrow a_0 \cdot b_0$ $(recursively)$
$c_2 \leftarrow a_1 \cdot b_1$ $(recursively)$
$c_3 \leftarrow a_0 + a_1$
$c_4 \leftarrow b_0 + b_1$
$c_5 \leftarrow c_3 \cdot c_4$ $(recursively)$
$c_6 \leftarrow c_5 - c_1 - c_2$
$c \leftarrow c_1 + c_6 x^{n/2} + c_2 x^n$
return c

3.3 ToomCook multiplication

Proposed by Andrei Toom and Stephen Cook, Toom-Cook multiplication can be considered a generalization of Karatsuba with reduced complexity. It is also a divide-and-conquer scheme following the same 5 major steps as mentioned for Karatsuba. A Toom-k algorithm splits a polynomial $A(x)$ of degree $n-1$ into k polynomials each having degree $n/k-1$ and n/k coefficients. Then these polynomials are evaluated for $2k-1$ different values and recombined. Below an example for polynomial $A(x)$ of degree $n-1$ is shown [55].

$$\begin{aligned} A(x) &= a_{n-1}x^{n-1} + \cdots + a_1 x + a_0 \\ &= x^{3n/4} \sum_{i=3n/4}^{n-1} a_i x^{i-3n/4} + \cdots + x^{n/4} \sum_{i=n/4}^{2n/4-1} a_i x^{i-n/4} + \sum_{i=0}^{n/4-1} a_i x^i \\ &= x^{3n/4}\alpha_3 + x^{2n/4}\alpha_2 + x^{n/4}\alpha_1 + \alpha_0 \\ \mathcal{A}(\mathcal{X}) &= \mathcal{X}^3\alpha_3 + \mathcal{X}^2\alpha_2 + \mathcal{X}\alpha_1 + \alpha_0, where\ \mathcal{X} = n/4 \end{aligned}$$

A similar conversion for another polynomial $B(x)$ is done and we get $\mathcal{A}(\mathcal{X})$ and $\mathcal{B}(\mathcal{X})$. Next step is to evaluate these two resultant polynomials for $2k-1$ values of $\mathcal{X}$. We trivially choose the points for Toom-3 as $\{0, \pm 1, -2, \infty\}$, for Toom-4 as $\{0, \pm 1, \pm 1/2, 2, \infty\}$ and evaluate the polynomial as shown below.

$$\begin{bmatrix} \mathcal{A}(0) \\ \mathcal{A}(1) \\ \mathcal{A}(-1) \\ \mathcal{A}(1/2) \\ \mathcal{A}(-1/2) \\ \mathcal{A}(2) \\ \mathcal{A}(\infty) \end{bmatrix} = \begin{bmatrix} 0 & 0 & 0 & 1 \\ 1 & 1 & 1 & 1 \\ -1 & 1 & -1 & 1 \\ 1/8 & 1/4 & 1/2 & 1 \\ -1/8 & 1/4 & -1/2 & 1 \\ 8 & 4 & 2 & 1 \\ 1 & 0 & 0 & 0 \end{bmatrix} \cdot \begin{bmatrix} \alpha_3 \\ \alpha_2 \\ \alpha_1 \\ \alpha_0 \end{bmatrix} \tag{3.2}$$

Next we multiply these $2k-1$ evaluated points of both the polynomials and compute $\mathcal{C}(i) = \mathcal{A}(i) \cdot \mathcal{B}(i)$ for $i \in \{0, \pm 1, \pm 1/2, 2, \infty\}$.

$$\begin{bmatrix} \mathcal{C}(0) \\ \mathcal{C}(1) \\ \mathcal{C}(-1) \\ \mathcal{C}(1/2) \\ \mathcal{C}(-1/2) \\ \mathcal{C}(2) \\ \mathcal{C}(\infty) \end{bmatrix} = \begin{bmatrix} \mathcal{A}(0) \\ \mathcal{A}(1) \\ \mathcal{A}(-1) \\ \mathcal{A}(1/2) \\ \mathcal{A}(-1/2) \\ \mathcal{A}(2) \\ \mathcal{A}(\infty) \end{bmatrix} \cdot \begin{bmatrix} \mathcal{B}(0) \\ \mathcal{B}(1) \\ \mathcal{B}(-1) \\ \mathcal{B}(1/2) \\ \mathcal{B}(-1/2) \\ \mathcal{B}(2) \\ \mathcal{B}(\infty) \end{bmatrix} \tag{3.3}$$

Depending on the size of the polynomials α_i different schemes can be used to do the polynomial multiplication required in the above step. Next we need to invert the pointwise evaluation of $\mathcal{C}(\mathcal{X})$ into polynomial $C(x)$. For this optimized interpolation as shown below.

$$\begin{bmatrix} \theta_0 \\ \theta_1 \\ \theta_2 \\ \theta_3 \\ \theta_4 \\ \theta_5 \\ \theta_6) \end{bmatrix} = \begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 1 & \\ 1 & 1 & 1 & 1 & 1 & 1 & \\ 1 & -1 & 1 & -1 & 1 & -1 & 1 \\ 1/64 & 1/32 & 1/16 & 1/8 & 1/4 & 1/2 & 1 \\ 1/64 & -1/32 & 1/16 & -1/8 & 1/4 & -1/2 & 1 \\ 64 & 32 & 16 & 8 & 4 & 2 & 1 \\ 1 & 0 & 0 & 0 & 0 & 0 & \end{bmatrix}^{-1} \cdot \begin{bmatrix} \mathcal{C}(0) \\ \mathcal{C}(1) \\ \mathcal{C}(-1) \\ \mathcal{C}(1/2) \\ \mathcal{C}(-1/2) \\ \mathcal{C}(2) \\ \mathcal{C}(\infty) \end{bmatrix} \tag{3.4}$$

The Toom-Cook algorithm is summarised in Alg. 19. It has a time-complexity

Algorithm 19: $\texttt{Toom} - \texttt{Cook}(a, b, c, k)$ [55]

Input : Polynomials a, b and integer k
Output : $c = a \cdot b$
$[\mathcal{A}_\prime(\mathcal{X}), \cdots, \mathcal{A}_{2k-2}(\mathcal{X})] \leftarrow$ `Evaluation` of $A(x)$
$[\mathcal{B}_\prime(\mathcal{X}), \cdots, \mathcal{B}_{2k-2}(\mathcal{X})] \leftarrow$ `Evaluation` of $B(x)$
for $i = 0$ *to* $2k-2$ **do**
  $\mathcal{C}_\backslash(\mathcal{X}) = \mathcal{A}_\backslash(\mathcal{X}) \cdot \mathcal{B}_\backslash(\mathcal{X})$
$C(x) \leftarrow$ `Interpolation` of $[\mathcal{C}_\prime(\mathcal{X}), \cdots, \mathcal{C}_{2k-2}(\mathcal{X})]$
return c

of $\mathcal{O}(n^{\frac{log(2k-1)}{logk}})$. So, for Toom-3 the complexity is $\mathcal{O}(n^{1.46})$, and for Toom-4 it is $\mathcal{O}(n^{1.40})$. Karatsuba and Toom-Cook are often use din combination with each other for optimized implementations of polynomial multiplications. In [54] Toom-Cook-4-way is coupled with Karatsuba for polynomial multiplication for Saber as shown in Fig. 3.3. They also couple these further with Schoolbook multiplication for computing multiplication of the polynomials.

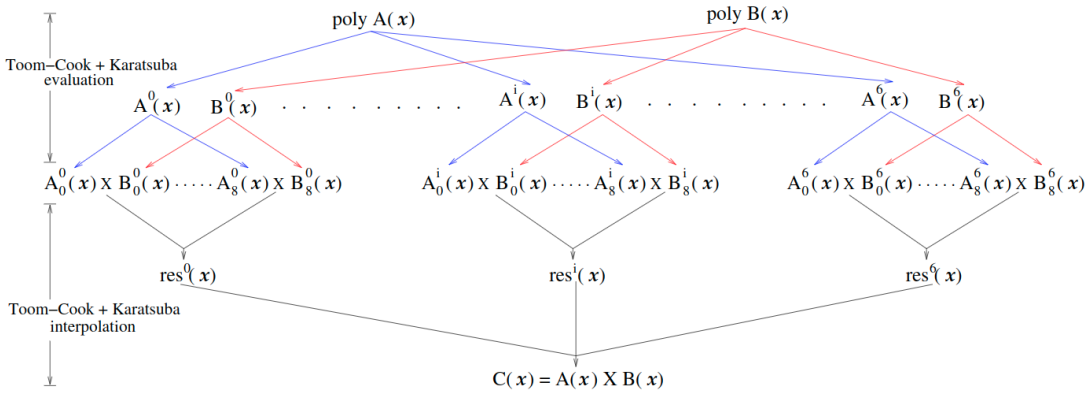

Figure 3.3: The Toom-Cook-4-way and Karatsuba multiplication used in Saber [54]

3.4 Toeplitz Matrix Vector Product(TMVP)

In [56]. Paksoy and Cent use Toeplitz Matrix-Vector Product for computing polynomial multiplication for Saber. TMVP was first proposed in [38] for multiplying elements in binary fields. This involves the construction of a Toeplitz matrix. One such matrix is shown below.

$$T = \begin{bmatrix} a_0 & a'_1 & a'_2 & \cdots & \cdots & \cdots & a'_{n-1} \\ a_1 & a_0 & a'_1 & a'_2 & \cdots & \cdots & \vdots \\ a_2 & a_1 & a_0 & a'_1 & \cdots & \cdots & \vdots \\ \vdots & a_2 & a_1 & \cdots & \cdots & \cdots & \vdots \\ \vdots & \ddots & \ddots & \ddots & \ddots & a'_1 & a'_2 \\ \vdots & & \ddots & \ddots & a_1 & a_0 & a'_1 \\ a_{n-1} & \cdots & \cdots & \cdots & a_2 & a_1 & a_0 \end{bmatrix} \tag{3.5}$$

It is symmetric against it's inverse diagonal and has $2n-1$ different elements. Thus addition of two $n \times n$ Toeplitz matrices only require $2n-1$ additions instead of n^2 additions required for normal matrices. Another interesting property is that every sub-matrix of a Toeplitz matrix is also a Toeplitz matrix, thus making divide-and-conquer work better over the naive multiplication method for matrix-vector multiplication. Suppose we want to multiply a Toeplitz matrix T with a vector $B = (b_0, b_1, \cdots, b_n - 1)$, then the product will look as shown below:

$$T \cdot B = \begin{bmatrix} a_0 & a'_1 & \cdots & a'_{n-2} & a'_{n-1} \\ a_1 & a_0 & \cdots & a'_{n-3} & a'_{n-2} \\ \vdots & \vdots & \ddots & \vdots & \vdots \\ a_{n-2} & a_{n-3} & \cdots & a_0 & a'_1 \\ a_{n-1} & a_{n-2} & \cdots & a_1 & a_0 \end{bmatrix} \begin{bmatrix} b_0 \\ b_1 \\ \vdots \\ b_{n-1} \\ b_{n-1} \end{bmatrix} \tag{3.6}$$

Now for a 2-way TMVP we will split the matrix T of dimension $n \times n$ into 4 smaller smaller sub-matrices, each of dimension $n/2 \times n/2$. Because of the structure of the Toeplitz matrix we get 3 distinct sub-matrices $T_0, T_1 and\ T_2$. We split vector B of size n into two smaller vectors B_0, B_1 of size $n/2$. Now the multiplication takes the following form:

$$T \cdot B = \begin{bmatrix} T_1 & T_0 \\ T_2 & T_1 \end{bmatrix} \cdot \begin{bmatrix} B_0 \\ B_1 \end{bmatrix} = \begin{bmatrix} P_1 - P_2 \\ P_1 - P_3 \end{bmatrix} \tag{3.7}$$

$$\begin{aligned} where\ P_1 &= T_0(B_0 + B_1), \\ P_2 &= (T_0 - T_1)B_1, \\ P_3 &= (T_1 - T_2)B_0 \end{aligned}$$

This gives a time-complexity $T(n) = 3M(n/2)+3n-1$, where M(n/2) is the complexity of computing polynomial multiplication for polynomials of degree $n/2$. Similarly for a 3-way TMVP we can divide the T in 9 sub-matrices out of which only 5 are distinct, and we divide B in 3 parts as shown below.

$$T \cdot B = \begin{bmatrix} T_2 & T_1 & T_0 \\ T_3 & T_2 & T_1 \\ T_4 & T_3 & T_2 \end{bmatrix} \cdot \begin{bmatrix} B_0 \\ B_1 \\ B_2 \end{bmatrix} = \begin{bmatrix} P_1 + P_4 + P_5 \\ P_2 + P_4 + P_6 \\ P_3 + P_5 + P_6 \end{bmatrix} \tag{3.8}$$

$$\begin{aligned} where\ P_1 &= (T_0 + T_1 + T_2)B_2, \\ P_2 &= (T_1 + T_2 + T_3)B_1, \\ P_3 &= (T_2 + T_3 + T_4)B_0, \\ P_4 &= T_1(B_1 + B_2), \\ P_5 &= T_2(B_0 + B_2), \\ P_6 &= T_3(B_0 + B_1) \end{aligned}$$

This gives a time-complexity $T(n) = 6M(n/3)+5n-1$, where M(n/3) is the complexity of computing polynomial multiplication for polynomials of degree $n/3$. Similar formulae can be constructed for a generalized k-way TMVP. For using this scheme for Saber one of the two polynomials to be multiplied, is converted to a Toeplitz matrix T and the other is taken as it is as vector B of coefficients. The modular reduction by prime and x^n+1 is done after every k-degree polynomial multiplication and addition/subtraction. In [56] 3-way/4-way TMVP is used and the 3/4 degree polynomials are multiplied using Toom-3/Toom-4 multiplication.

3.5 NTT-based multiplication

NTT- based multiplication [28] further reduces the complexity of polynomial multiplication to $\mathcal{O}$(n(log n)). A very easy-to-follow explanation of NTT based multiplication is provided in [69] by Daan Sprenkels. The multiplication is performed in

the ring $\mathcal{R}_q = \mathbb{Z}_q[x]/(x^n+1)$, where degree n is a power of two and $q \equiv 1 mod\ 2n$ for complete NTT/INTT. This is similar to Karatsuba and Toom-Cook with respect to the fact that it also involves evaluation of a $n-1$ degree polynomial on n-points, their pointwise multiplication with the n evaluated points of the other polynomial, and then reconstructing the multiplied polynomial back from the n multiplied points. However, the difference in complexities is because of the conversion to point domain and reconstruction. It makes use of the Fast Fourier Transform(FFT) for this. NTT is FFT in a discrete field instead of a real field.

A complete understanding of NTT requires knowledge of the Chinese Remainder Theorem(CRT) [6] and Residue Number System(RNS) [7].

3.5.1 Chinese Remainder Theorem(CRT)

It gives a unique remainder for a number divided by product of several pairwise co-prime integers, given that the different remainders of the number divided by the each of these integers is known. For example, consider a system of congruence:

$$\begin{aligned} x &\equiv a_1 (mod\ n_1) \\ x &\equiv a_2 (mod\ n_2) \\ &\vdots \\ x &\equiv a_k (mod\ n_k) \end{aligned}$$

where the n_i are pairwise co-prime, and let $N = \prod_{i=0}^{k} n_i$. Let's say for $k = 3$, and $n_1 = 3$,$n_2 = 4$, $n_3 = 5$, and $N = 3 \cdot 4 \cot 5 = 60$. Then the equations look like:

$$\begin{aligned} x &\equiv 0 (mod\ 3) \\ x &\equiv 3 (mod\ 4) \\ x &\equiv 4 (mod\ 5) \end{aligned}$$

To find a solution it is sufficient to iterate it for all the values in the 0 *to* N and find that $x \equiv 39 (mod\ 60)$ satisfies all the 3 congruences. More efficient ways for finding out x exist in literature. Here $\{a_1, a_2, \cdot, a_k\}$ is the RNS representation of x in terms of pairwise co-prime integers $\{n_1, n_2, \cdot, n_k\}$. The beauty of CRT lies in the fact that it allows the construction of RNS for larger values of x based on similar smaller values of a_i. For example, if we multiply all the a_i for the above example by 3 as follows:

$$\begin{aligned} x &\equiv 3.0 (mod\ 3) \equiv 0 (mod\ 3) \\ x &\equiv 3.3 (mod\ 4) \equiv 1 (mod\ 4) \\ x &\equiv 3.4 (mod\ 5) \equiv 2 (mod\ 5) \end{aligned}$$

then the solution for x would be $x \equiv 3.39 (mod\ 60) \equiv 57 (mod\ 60)$. Using this we can break down big multiplications into very small multiplications and use these results reconstruct back the bigger multiplied result.

Now that we have established the background we can proceed to its use in the design of this efficient polynomial multiplication algorithm. As mentioned earlier, the idea behind NTT-based multiplication is similar to the Toom-Cook or Karatsuba, i.e., we wish to split the polynomials of degree n into 0-degree polynomials, which we can then pointwise multiply and then use CRT to reconstruct back the original polynomial, as shown in Alg. 20.

Algorithm 20: `NTT_based_multiplication`(a, b, c)

Input : a, b Polynomials in ring $\mathcal{R}_q$
Output : $c = a \cdot b$ Polynomial in ring $\mathcal{R}_q$
$\mathcal{A}_{ntt} \leftarrow \texttt{NTT}(a)$
$\mathcal{B}_{ntt} \leftarrow \texttt{NTT}(b)$
$\mathcal{C}_{ntt} \leftarrow \mathcal{A}_{ntt} \cdot \mathcal{B}_{ntt}$ (pointwise multiplication (mod q))
$c \leftarrow \texttt{INTT}(\mathcal{C}_{ntt})$
return c

NTT transformation is used to split the polynomial into N points and INTT transformation is used to reconstruct back the multiplied polynomial. The complexity of these transformations are $\mathcal{O}$(n(log n)) and that of pointwise multiplication is $\mathcal{O}(n)$, thus the overall complexity is $\mathcal{O}$(n(log n)). For ring $\mathcal{R}_q = \mathbb{Z}_q[x]/(x^{256}+1)$, we try to split the reduction polynomial into smaller parts. We will compute the multiplications in terms of these smaller parts and then reconstruct the solution back for the $x^{256}+1$. So, first, we try to find polynomials such that:

$$\begin{aligned}(x^{256}+1) &= (x^{128}-\eta_0)(x^{128}+\eta_0)\\ &= (x^{64}-\eta_1)(x^{64}+\eta_1)(x^{64}-\eta_1^3)(x^{64}+\eta_1^3)\\ &\vdots\\ &= (x-\eta)(x+\eta)(x-\eta^{129})(x+\eta^{129})\cdots(x-\eta^{127})(x+\eta^{127})(x-\eta^{255})(x+\eta^{255})\end{aligned}$$

This splitting can be visualized as a binary tree and therefore after $logn$ steps of splittings our polynomials will be in their irreducible forms. When we solve the right hand side for η_i, we get $\eta_0 = \eta^{128}$ and using the first inequality we get $\eta_0^2 = -1$ or $\eta_0^4 = 1$, thus we get $\eta^{2\cdot 256} = 1$. We can conclude that η is the $2n$th root-of-unity. In the forward transform the we reduce the polynomials by $\{eta_0, eta_1, \cdots, \eta\}$ step by step. In the inverse transform we use inverse-root-of-unity to reconstruct the polynomial.

For example, let us say we have two polynomials a, b. After NTT transformation they map to points $\{2, 3, 5, 2\}$ and $\{3, 4, 2, 7\}$ as shown in Fig 3.4. Then we multiply these pointwise and we get polynomial c having points $\{6, 12, 10, 14\}$. Next, we will perform INTT transformation on this polynomial c, to reconstruct the multiplied polynomial.

Here a complete NTT/INTT is described however, certain schemes like Kyber [65] use incomplete NTT/INTT, where they stop at a higher degree polynomial and uses schoolbook like multiplication for the small polynomial multiplication.

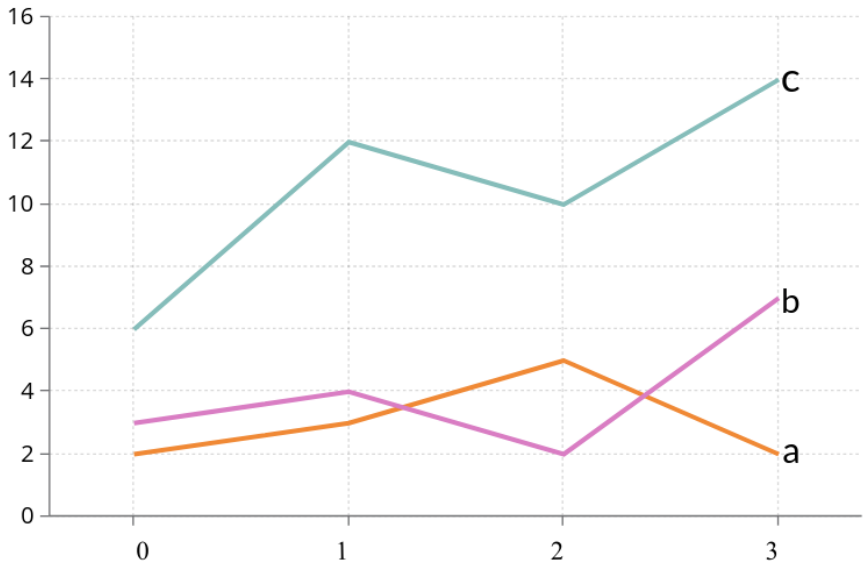

Figure 3.4: Example showing NTT intermediate values during NTT-multiplication

3.6 Summary of the multiplication methods

In this chapter, we discussed various multiplication strategies, which have different complexities and based on the optimizations can be used efficiently for different PQC schemes.

For multiplying two polynomials of degree n, Schoolbook multiplication has a time-complexity of $\mathcal{O}(n^2)$, but if we implement it using n multipliers the complexity becomes $\mathcal{O}(n)$, which is very fast. Karatsuba and Toom Cook use the divide-and-conquer strategy and further reduce the complexity from $\mathcal{O}(n^2) \rightarrow \mathcal{O}(n^{1.58}), \mathcal{O}(n^{1.46})$ for 2-way Karatsuba and Toom-3 respectively. Toeplitz Matrix vector multiplication also provided a similar advantage while multiplying different polynomials with a divide-and-conquer-based approach. NTT-based multiplication further reduced the complexity to $\mathcal{O}$(n(log n)).

A very important and expensive part of the multiplication unit is the Modular-reduction unit. We discuss various existing design methods for an efficient modular reduction unit in the next chapter.

Chapter 4

Modular Reduction methods

Modular reduction implies we have an integer number c and modulus p, then the modular reduction of c by p computes the following:

$$\tilde{c} = c \; mod \; p = c - \lfloor \frac{c}{p} \rfloor . p$$

The naive method to perform this will involve multiplying these values and then performing a modulo operation. Although it looks simple it is computationally very expensive. For such an operation, hardware will be required to compute the division of two numbers and then subtract the *dividend* $\times$ p from the multiplied value. This method was further improved using an interleaved method. A very simple interleaved Modular Reduction algorithm for two numbers a, b which are to be multiplied is given in Alg. 21. In this method, a division is used along with flooring operation. This is a naive very time-consuming operation. Using fast estimation techniques its performance has been improved in literature. However, not that division is more expensive than multiplication, and therefore such an algorithm will reduce the clock-frequency of the design and also consume a lot of area.

Algorithm 21: `Interleaved_Modular_Reduction`(a, b, c) [5]

Input : a, b Integers , primes p such that $r^{n01} < p < r^n$ for radix-r
Output : $c = (a \times b) mod \; p$
$t \leftarrow 0$ **for** $i = 0$ *to* $n - 1$ **do**
    $t \leftarrow r + b_i \cdot a$ (here b_i refers to ith digit of b)
    $t \leftarrow t - \lfloor \frac{t}{p} \rfloor \cdot p$
return c

Many algorithms like RSA and DSA require the multiplication of two large numbers modulo another large number. They will suffer if the implementation is not efficient. For Dilithium, we have a prime modulus, which requires an efficient modular reduction unit during polynomial multiplication. Therefore, we investigate more efficient modular reduction techniques in the next few sections of the Chapter.

4.1 Montgomery Reduction

Peter L. in 1985 [37] introduced Montgomery reduction- an efficient method for computing the multiplication of two integers modulo p while avoiding division by p. This is used in the official implementations of various PQC schemes like Dilithium to avoid the expensive computational cost of performing 'naive' modular reduction.
Let us say we have two numbers a, b, and a modulus p, and we need to compute c

Algorithm 22: `Montgomery_Reduction`(a, b, c) [5]

Input : a, b Integers
Output : $c = (a \times b) mod\ p$
Choose $R \in \mathbb{N}, s.t., R > p\ and\ gcd(R, p) = 1$
$k = \frac{R(R^{-1}\ mod\ p)-1}{p}$
$\bar{a} = aR\ mod\ p$
$\bar{b} = bR\ mod\ p$
$x = \bar{a}\bar{b}$
$s = xk\ mod\ R$
$t = x + sp$
$u = \frac{t}{R}$
if $u < p$ **then**
    $\bar{c} = u$
else
    $\bar{c} = (u - p)$
$c = (\bar{c}R^{-1}\ mod\ p)$
return c

such that:

$$c \equiv (a \times b) mod\ p$$

Now, for Montgomery reduction, instead of multiplying a and b, we chose a number $R \in \mathbb{N}$ that is greater than p and is co-prime with p, i.e., gcd(R, p)=1, and we use R to compute *residues* of a and b given as:

$$\bar{a} = aR\ mod\ p$$

$$\bar{b} = bR\ mod\ p$$

Generally, R is chosen as a power-of-two to simplify the multiplication and division operations, as multiplication by 2^k is just shifting the value by k bits and division by 2^k is taking the least-significant k bits of the number. R^{-1} is defined as the Modular inverse of R such that:

$$RR^{-1} \equiv 1\ mod\ p$$

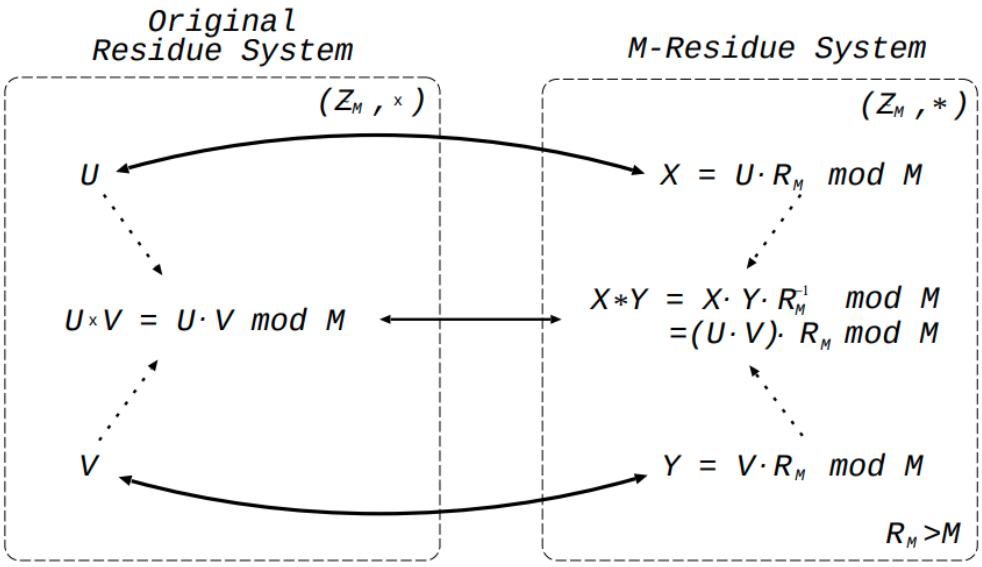

Figure 4.1: Mapping between the original residue system and the Montgomery residue system [37]

. Thus we can express $\bar{c}$(residue of c) in terms of $\bar{a}$ and $\bar{b}$ and recompute c int he following way:

$$\begin{aligned}\bar{c} &\equiv (a \times b)R \\ &\equiv (aR \times bR)R^{-1} \\ &\equiv (\bar{a} \times \bar{b})R^{-1} \ mod \ p\end{aligned}$$

Since R^{-1} can be easily computed and stored, it can be seen that the above step is just a few multiplications and additions. The algorithm for Montgomery reduction is defined in 22 and a figure to explain the transformation is shown in Fig/ 4.1. It uses $k = \frac{R(R^{-1} \ mod \ p)-1}{p}$ for computing $xR^{-1} \ mod \ p$ as shown below:

$$\begin{aligned}s &= (x \ mod \ R)k \ mod \ R \\ u &= (x + sp)/R \\ if \ u \geq p \ then \ return \ u - p \ else \ u\end{aligned}$$

4.2 Barrett Reduction

Paul Barrett introduced the Barrett reduction in 1987 [18], aiming to implement RSA on a standard DSP. It works on the assumption that for two integers a, b that we wish to multiply, the modulus $p \in \mathbb{N}$, $p \geq 3$ is constant and $0 \leq ab < p^2$. Similar to Montgomery reduction, we will pre-compute a factor R using division and then perform basic multiplication, subtractions, and shifts which are computationally

much less expensive and faster than division operation. We chose a value k such that $2^k > p$ and compute $R = \lfloor \frac{4^k}{p} \rfloor$. The Algorithmic description of the Barrett reduction is given in Alg. 23.

Algorithm 23: `Barrett_Reduction`(a, b, c) [4]

Input : a, b Integers
Output : $c = (a \times b) mod\ p$
Choose $k \in \mathbb{N}, s.t., 2^k > p$
$R = \lfloor \frac{4^k}{p} \rfloor$
$x = ab$
$t = x - \lfloor \frac{xR}{4^k} \rfloor$p
if $t < p$ **then**
    $c = t$
else
    $c = (t - p)$
return c

It bears many similarities with the Montgomery reduction discussed earlier. They are both used for making the modular reduction fast and require pre-computing of certain constants for a modulus p. They are both performing two multiplications and the divisions are always performed by powers of two, which is simple truncation by some bits. Although the multiplication in Barrett reduction multiplies 2 k-bit numbers with another k-bit number whereas in Montgomery reduction we multiply a k-bit number with another k-bit number, thus the former is more expensive to implement than the latter. Another major difference lies in the fact that Montgomery reduction requires conversion of integers to and from Montgomery form as residues, whereas Barrett reduction works on normal values as can be seen in the two Algorithms Alg. 22 and Alg. 23. Montgomery deals with congruence and exact computations whereas Barrett reduction is performing approximate computations with a bound on precision. Both of these schemes require multiplications and transformations, however for constraint devices that require high-speed computation, we can use memory and perform some look-ups instead of multiplications as shown in the next section.

4.3 Lookup-table-based modular reduction methods

Basic Lookup-table-based modular reduction is a naive method in which the first step is to pre-compute the look-up table for a modulus p. We choose an integer k such that $2^{k-1} < p < 2^k$. Then the pre-computation table looks like Tab. 4.1. The algorithm is given in Alg. 24.

l	$R[l]$
$2k-1$	$2^{2k-1}\ mod\ p$
$2k-2$	$2^{2k-1}\ mod\ p$
$\vdots$	$\vdots$
k	$2^{k}\ mod\ p$

Table 4.1: Pre-computation table for a modulus p

Next we compute the binary string of integer $c = ab$ as $Bin(c)$, then:

$$c = \sum_{i=0}^{k-1} c_{k+i} R[k+i] + \sum_{j=0}^{k-1} z_j 2^j \ mod\ p$$

where $z_j \in \{0,1\} \forall j$. For example let us say $p = 97 = (1100001)_2$, $k = 7$, $c = 3135 = (110000111111)_2$, and $l = 12$. We will pre-compute the look-up table for $R[10] = 2^{10}\ mod\ p = 11$ and $R[11] = 2^{11}\ mod\ p = 54$. Then :

$$c = 3135 = R[11] + R[10] + (111111)_2 = 11 + 54 + 63 \equiv 31\ mod\ 97$$

Algorithm 24: `Basic LUT − based modular reduction` (p, k, T, c, l, r) [23]

Input : Modulus p, $k = Bitlength(p)$, $c, s.t., 0 \leq c < 2^{2k}$, $l = Bitlength(c)$, Look-up table T
Output : $r = c\ mod\ p$
if $c < p$ **then**
  return c
if $l = k$ **then**
  return $c - p$
$s \leftarrow Bin(c)$
$r \leftarrow 0$
for $i = l - 1\ to\ k$ **do**
  if $s[i] = 1$ **then**
    $r \leftarrow r + T[i]$
$r \leftarrow r + \sum_{j=0}^{k-1} s[j]2^j$
while $r \geq p$ **do**
  $r \leftarrow r - p$
while $r < 0$ **do**
  $r \leftarrow r + p$
return r

This method gives bad run-time complexity for strings like $11 \cdots 1$ and to improve this another method is known as *Run-length-based modular reduction* was proposed,

which involves flipping bits of c described in [23]. However, the latter is less efficient than the former for strings like $1010\cdots10$. One way to deal with both types of the string is that we divide $Binary(c)$ into segments of length $l-k, k$, and count the number of 1's in the left segment. If it is greater than the runs in the left segment then we use Run-length based method otherwise we use the Basic method. This combination of the two reduction methods is further refined for a new type of reduction method called *Fast reduction method*[23]. The fast reduction method is twice as fast as Barrett's reduction. The memory requirement is also not huge making it portable and suitable for resource constraint devices.

All of these methods perform modular reduction for general moduli. However, if we chose the modulus carefully, we can use the structure for an even more efficient modular reduction method as described in the next section.

4.4 Efficient Reduction unit for special primes

In [72] Lim et.al. discuss that for different special primes like - Mersenne primes, pseudo-Mersenne primes, and generalized Mersenne primes/Solinas primes[67], the modular reduction can be expressed in terms of simple additions and subtractions.

Mersenne primes are prime numbers having structure $2^n - 1$ for some integer n. However such primes occur less frequently in a long-range. Pseudo-Mersenne primes have the structure $2^n - t$ for some integer n and a small integer t. These are comparatively more frequent than the Mersenne primes. Generalized Mersenne primes/Solinas primes have structure $2^n \pm 2^k \pm 1$ amongst these $2^n - 2^k - 1$ or 40% more frequent than $2^n + 2^k + 1$.

For pseudo-Mersenne prime $p = 2^n - t$, given an integer c, we can express it as follows:

$$\begin{aligned} c &= c''2^n + c' \\ &\equiv c''t + c' \ mod\ p, \text{ since } 2^n \equiv \ (mod\ p) \end{aligned}$$

By applying this method recursively on c'', we can obtain $c\ mod\ p$ using a few shifts and additions as multiplication by 2^n is left shift by n. The algorithm is given in Alg. 25. This can be further optimized considering that since c is a product of two numbers a, b, in the same finite field, the product can never be greater than 2^2n. This is shown in [70]. An efficient modular reduction for Solinas primes[67] is given in [70]. This work shows that using such a dedicated modular reduction unit for special primes leads to improved performance over other traditional modular reduction methods like Montgomery reduction. This improvement increase with the increase in the field size. The only disadvantage is its scalability, as it requires a dedicated unit if we change the modulus p the structure of the unit will also need to be changed.

For e.g. let us consider the case when prime is ($p = 12289 = 2^{14} - 2^{12} + 1$)[77]. The the integer c is the multiplication of two numbers a, b in the same field, therefore

Algorithm 25: `Reduction by pseudo − Mersenne prime` (c, p, r) [70]

Input : Modulus $p = 2^n - t$, Integer c to be reduced
Output : $r = c \ mod \ p$

$q \leftarrow c/2^n$
$r \leftarrow c \ mod \ 2^n$
$i \leftarrow 0$
while $q^{[i]} > 0$ **do**
    $r \leftarrow r + (tq^{[i]} \ ^{mod \ 2^n})$
    $q^{[i+1]} \leftarrow (tq^{[i]/2^n})$
    $i \leftarrow i + 1$
while $r \geq p$ **do**
    $r \leftarrow r - p$
return r

c can be expressed in 28 bits. The modular reduction of c can be expressed as following:

$$
\begin{aligned}
c &\equiv 2^{14}c[27:14] + c[13:0] \\
&\equiv 2^{12}c[27:14] - c[27:14] + c[13:0] \\
&\equiv 2^{14}c[27:16] + 2^{12}c[15:14] - c[27:14] + c[13:0] \\
&\vdots \\
&\equiv 2^{12}(c[27:26] + c[25:24] + c[23:22] + c[21:20] + c[19:18] + c[17:16] + c[15:14]) \\
&- (c[27:26] + c[27:24] + c[27:22] + c[27:20] + c[27:18] + c[27:16] + c[27:14]) \\
&+ c[13:0] \\
&\equiv 2^{12}x - y + c[13:0] \ (mod \ p)
\end{aligned}
$$

where $x = c[27:26]+c[25:24]+c[23:22]+c[21:20]+c[19:18]+c[17:16]+c[15:14]$, and $y = c[27:26]+c[27:24]+c[27:22]+c[27:20]+c[27:18]+c[27:16]+c[27:14]$. Now we can further simplify $2^{12}c$ suing the special property of modulus p as shown in [77]. Thus computing the entire modular reduction using some additions and subtractions without even a single multiplication required by the Barrett or Montgomery reduction. It also does not require any pre-computation of constants or LUTs and is, therefore, memory efficient as well. For a chosen special prime it is the most efficient way and finishes in constant time.

4.5 Summary of the modular reduction methods

In this chapter, we discussed various methods to reduce an integer number modulus to another integer number. We saw how the naive modular reduction method involves division which is very expensive to implement in hardware. Therefore in the next sections, we discussed Montgomery and Barrett reduction algorithms which

replace this division with a few multiplication and additions. We see how Montgomery and Barrett reduction can perform the modular reduction with very few pre-computation steps. We also see the differences and similarities between these two reduction methods. Next, we discuss lookup-table-based reduction methods where we spend most of our time in pre-computation and therefore, only require many additions and subtraction units and no multiplication units.

All of these schemes are generalized for all the integer moduli, however, next, we see that if we carefully choose are moduli that have a special structure like that of 'Mersenne primes' or 'Solinas primes' then we can further optimize our modular reduction unit with very few additions and subtractions. However, the only disadvantage is that the implementation is modulus-specific and cannot be generalized to all the moduli. Therefore, if the moduli change the design will also change.

We have established the background for our work. We discussed various methods to implement the polynomial multiplier and modular reduction unit. In the next chapter, we will discuss our design decisions.

Chapter 5

Design strategies

In this chapter, we discuss the different strategies taken into consideration for implementing an efficient unified polynomial arithmetic unit for Saber and Dilithium.

As described in Chapter 2, 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. Note that in the ring or module lattice-based post-quantum public-key schemes, polynomial multiplications, hash calculations, and pseudo-random number generations are the most expensive operations.

The design of the two schemes, as discussed in Chapter 1, motivated us to investigate efficient implementation techniques such that we could design a unified polynomial arithmetic architecture for accelerating the polynomial arithmetic functions in the two schemes. We discussed in Chapter 3 that polynomial arithmetic has the major functional usage ($> 50\%$) compared to other functions, and is, therefore, very important to make it fast and yet lightweight. Having a compact as well as a unified implementation of this architecture for two schemes is a step towards making a common cryptoprocessor that supports lattice-based KEM and digital signature on resource-constrained platforms. In the following part of this section, we discuss the challenges in implementing a unified polynomial arithmetic architecture for Dilithium and Saber.

We would like to remark that synergies also exist in other lattice-based schemes. For example, CRYSTALS-Kyber [22] shows great similarities with Saber [34] as well as Dilithium as all 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 architecture implementation typically has a long design cycle, in this work we stay focused on unifying the polynomial arithmetic unit for Saber with Dilithium, and by doing so we show that a compact and unified polynomial arithmetic architecture for post-quantum digital signature and key exchange is feasible.

A polynomial arithmetic unit consists of three major operations between two polynomials- addition, subtraction, and multiplication. Based on the complexity of implementation we divide this explanation into the following two parts:

- Polynomial Addition and Subtraction unit
- Polynomial multiplication unit

5.1 Polynomial Addition and Subtraction unit

Polynomial addition and subtraction are very simple pointwise operations as shown in Alg. 26 and Alg. 27.

Algorithm 26: `AdditionOf2polynomials` (a, b, c)

Input : a, b Polynomials in $\mathcal{R}_q$
Output : $c = a + b$ Polynomial in $\mathcal{R}_q$
$N \leftarrow LengthOfPolynomial(a\ or\ b)$
for *i=0 to N-1* **do**
    $t \leftarrow a[i] + b[i]$
    if $t < q$ **then**
        $c[i] = t$
    else
        $c[i] = t - q$
return c

Algorithm 27: `SubtractionOf2polynomials` (a, b, c)

Input : a, b Polynomials in $\mathcal{R}_q$
Output : $c = a - b$ Polynomial in $\mathcal{R}_q$
$N \leftarrow LengthOfPolynomial(a\ or\ b)$
for *i=0 to N-1* **do**
    $t \leftarrow q + a[i] - b[i]$
    if $t < q$ **then**
        $c[i] = t$
    else
        $c[i] = t - q$
return c

In addition, we take two coefficients, add them and then check if the sum is higher than our modulus q or not. If it is we subtract q from the sum otherwise we return the original sum. Similarly, for subtraction, we subtract a value from

another value that is pre-added with prime q to avoid signed computations. Next, we perform a similar check as addition and return the subtracted value in mod q.

5.2 Polynomial multiplication unit

In Chapter 3 we discussed various algorithms that we can use for polynomial multiplication. Schoolbook multiplication is the most commonly used method, owing to the simplicity of its implementation, however, it has a very high time complexity of $O(n^2)$. This is further reduced to $\mathcal{O}(n^{\log_2(3)})$ if we Karatsuba method [48], and to $\mathcal{O}(c(k) \cdot n^e)$ if we use the Toom-Cook method [73], where $e = \log(2k-1)/\log(k)$. The Fast Fourier Transform or Number Theoretic Transform method [49], is the fastest with the time complexities of $\mathcal{O}(n \log n)$ as discussed in Chapter 7. Different hybrid polynomial multiplication techniques which combine the above-mentioned algorithms are also available in the literature.

Dilithium [16] has prime moduli and therefore makes Number Theoretic Transform (NTT) method an integral part of the protocol to compute polynomial multiplications in the least time. It transforms the two polynomials to be multiplied in the NTT domain and then performs pointwise multiplication as shown in Alg. 28. Then this multiplied polynomial is reconstructed ted back using Inverse Number Theoretic Transform (INTT).

Algorithm 28: `PointwiseMultiplicationOf2polynomials` (a, b, c)

Input : a, b Polynomials in $\mathcal{R}_q$
Output : $c[i] = a[i] \cdot b[i] \forall i \in \{0, N\}$ Polynomial in $\mathcal{R}_q$
$N \leftarrow LengthOfPolynomial(a \text{ or } b)$
for *i=0 to N-1* **do**
    $t \leftarrow a[i] \cdot b[i]$
    $c[i] = ModReduce(t)$
return c

However, Saber [34] has power-of-two moduli. It cannot use the NTT method directly for its application. However, it can use the other multiplication algorithms or a hybrid of multiple algorithms like Karatsuba and Toom-Cook, or Karatsuba and schoolbook, etc. Since the coefficient-moduli are powers-of-two in Saber, the modular reductions become free of cost if schoolbook or Karatsuba or Toom-Cook or any combination of them is used. In [63] the hardware implementation of Saber uses a highly parallel schoolbook multiplier that computes one polynomial multiplication in just 256 cycles. If we still try to implement Saber using NTT-based polynomial multiplication [26], it will require 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 [63]) 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.

We might be able to reduce the size of the modulus required based on certain assumptions. We will see how to do this in the next Chapter along with implementation details.

Chapter 6

Implementation in Hardware

As described in the previous section, if we wish to make a common Polynomial arithmetic unit for Saber and Dilithium we will have to use the NTT-based multiplication unit. However, for Saber designing such a unit might become very expensive owing to large modulus size requirements. In the next section, we discuss how we overcome this problem.

6.1 Prime selection for NTT in Saber

As discussed in Chapter 2, the secret polynomial coefficients of Saber are signed values in the range [-3,3] for FireSaber, [-4,4] for Saber, and [-5,5] for LightSaber. For the positive coefficients a modulus of order $2^3 \times 2^{13} \times 256 = 2^{24}$ is sufficient, however, for negative coefficients it is small. If we convert the negative coefficients to unsigned values by performing modular reduction by $q = 2^{13}$, then the required modulus size increases to $2^{13} \times 2^{13} \times 2^8 = 2^{34}$. If we don't convert then we will have to provide support for signed arithmetic which is also expensive to deal with. Implementing a common multiplier that supports 2^{34} order modulus will be very costly as well in comparison to the one supporting 2^{23} order modulus required by Dilithium.

In [40, 26], the designers also discuss a similar problem and mention that a 24-bit modulus can be used along with a special provision for signed number representation. However, we observe that if we still take a prime p' of the order of 2^{25} 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. Thus, our aim to make an efficient and common polynomial multiplication unit can be achieved by implementing an NTT-based multiplier with support for 2^{25} order modulus. But the question arises is this the best we can do? What will happen if we take a smaller modulus, or let's say Dilihtium's modulus? Will it completely fail or will there be some low probability of failure?

We performed various experiments to answer this question and concluded that for a prime or size 25-bit the failure probability is 0, as expected. For a 24-bit prime, the failure probability is 2^{-350}, and for Dilithium's 23-bit prime the failure probability is 2^{-100}. These probabilities were obtained for the worst possible cases when the input

Prime	Experimental Failing Probability for Saber
$2^{25} - 2^{14} + 1$	0
$2^{24} - 2^{14} + 1$	2^{-350}
$2^{23} - 2^{13} + 1$	2^{-100}

Table 6.1: Proposed primes with experimental failing probabilities

is generated using a poor random number generator. For uniformly or binomially distributed inputs as mentioned in the scheme specification, these primes never fail. However, going by Murphy's law that the worst will happen, we provide an implementation for all the 3 different sizes of primes modulus. So, the user can decide which one is more suitable for his application. If we use Dilithium's prime we can have the same multiplication and modular reduction unit however for the other two primes we will have to design a unified NTT/INTT unit as well as a unified modular reduction unit. We provide all three different kinds of implementations.

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.

6.2 Efficient modular reduction unit

The original software source code of Dilithium [16] uses the Montgomery modular reduction. As we discussed in Chapter3, the modular reduction using special primes is the most optimized and least expensive to implement. The Dilithium prime $p = 2^{23} - 2^{13} + 1$ has a sparse structure. If we chose similar 24-bit and 25-bit primes having sparse structure for Saber as well then their modular reduction circuits can be unified very well to reduce the area overhead. Therefore, we choose $2^{24} - 2^{14} + 1$ as the 24-bit prime and $2^{25} - 2^{14} + 1$ as the 25-bit prime. After carefully choosing sparse and reduction-friendly primes for Saber, we followed the add-shift-based method [76] and used a similar fast modular reduction technique. The final primes with failing probabilities are listed in Tab. 6.1.

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 $\{0, \ldots, q-1\}$ as shown in Alg. 35. Fig. 6.1 shows the modular reduction unit which uses a carry-save adder tree to reduce the critical path. Thus, we choose special primes for our implementation which do not need to change as long the design specification does not change and implement a dedicated unified modular reduction unit. Which just performs very few additions and subtractions. Such efficient modular reduction units are widely used in literature like in the implementation for NewHope discussed in Chapter 3.

In the next section we discuss how we implement an efficient unified NTT-based multiplier for Saber and Dilithium .

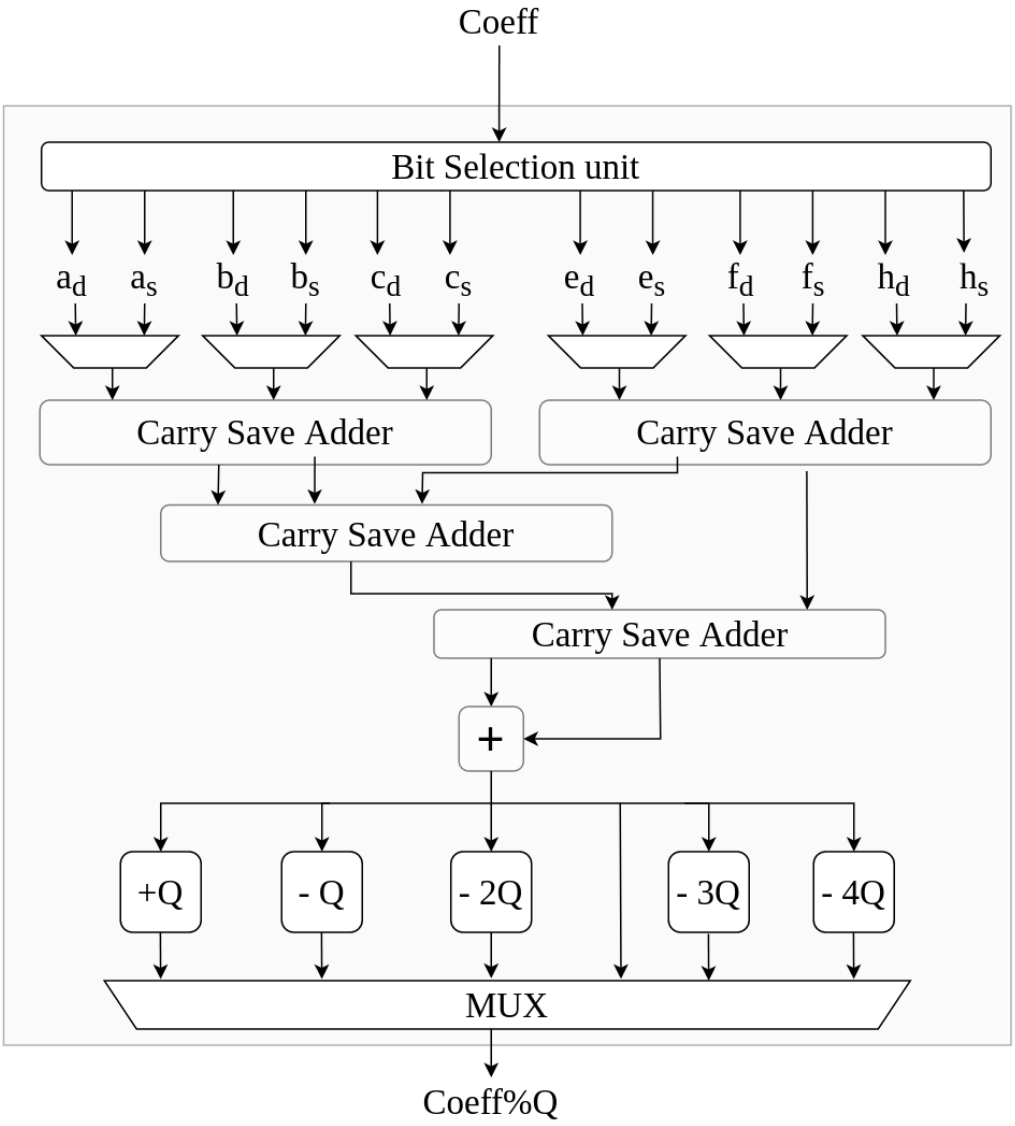

Figure 6.1: Unified modular reduction unit

6.3 NTT-based unified polynomial multiplier

After carefully choosing three different prime candidates for Saber, based on their failing probabilities we now describe the implementation decisions we make for designing the NTT-based polynomial multiplier for both Saber and Dilithium. Fig. 6.2 gives an overview of NTT based multiplication described in Chapter 3. We first convert the two polynomials to be multiplied into the NTT domain, then perform pointwise multiplication, and the multiplied polynomial back to the normal domain using Inverse NTT(INTT).

We have already discussed how to implement pointwise multiplication in Chapter 5. Now we discuss how to transform a polynomial into NTT/INTT and its implementation aspects. The NTT/INTT transformation can be visualized in terms of

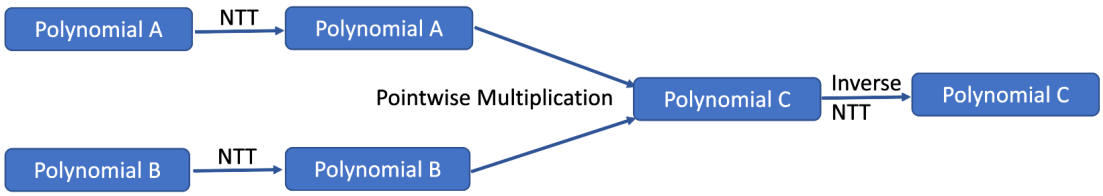

Figure 6.2: NTT-based multiplication flow

$$\begin{bmatrix} f_0 \\ f_1 \\ \vdots \\ f_{n-1} \end{bmatrix} = \begin{bmatrix} 1 & 1 & 1 & \cdots & 1 \\ 1 & \alpha & \alpha^2 & \cdots & \alpha^{n-1} \\ 1 & \alpha^2 & \alpha^4 & \cdots & \alpha^{2(n-1)} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & \alpha^{n-1} & \alpha^{2(n-1)} & \cdots & \alpha^{(n-1)(n-1)} \end{bmatrix} \begin{bmatrix} v_0 \\ v_1 \\ \vdots \\ v_{n-1} \end{bmatrix}$$

Figure 6.3: NTT transformation using matrix-vector multiplication

$$\begin{bmatrix} v_0 \\ v_1 \\ \vdots \\ v_{n-1} \end{bmatrix} = \frac{1}{n} \begin{bmatrix} 1 & 1 & 1 & \cdots & 1 \\ 1 & \alpha^{-1} & \alpha^{-2} & \cdots & \alpha^{-(n-1)} \\ 1 & \alpha^{-2} & \alpha^{-4} & \cdots & \alpha^{-2(n-1)} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & \alpha^{-(n-1)} & \alpha^{-2(n-1)} & \cdots & \alpha^{-(n-1)(n-1)} \end{bmatrix} \begin{bmatrix} f_0 \\ f_1 \\ \vdots \\ f_{n-1} \end{bmatrix}$$

Figure 6.4: INTT transformation using matrix-vector multiplication

matrix-vector multiplication as shown in Fig. 6.3 and Fig. 6.4. Here, vector V consists of the n polynomial coefficients in the normal domain. After multiplication pre-generated matrix of various powers of primitive-root-of-unity, we get the vector f. Now we can use this for pointwise multiplication. Then when we have to convert back a polynomial f in the NTT domain we again multiply it by a pre-generated matrix of various powers of primitive-root-of-unity as shown in Fig. 6.4.

This approach although seems simple will lead to a very expensive hardware implementation. As instead of requiring $\mathcal{O}(n(logn))$ it requires $\mathcal{O}(n^2)$ multiplication and also requires the two hug $n \times n$ matrices pre-computed stored in hardware. We can improve this by generating these values on the fly however, it will cost us more multiplication units. Thus, we need to find a better way to perform this transformation as discussed in the next section.

6.4 NTT/INTT transformation method

Following the official reference code of Dilithium, we also decide to use the iterative NTT/INTT algorithms- Cooley-Tukey (Alg. 30) and Gentleman-Sande (Alg. 29) butterfly configurations for the NTT and inverse NTT respectively. Cooley-Tukey as shown in Fig. 6.5 is a decimation-in-time flow whereas Gentleman-Sande as shown in Fig. 6.6 is a decimation-in-frequency flow. These are the most common FFT algorithms. Cooley-Tuley breaks DFTs into smaller DFTs recursively, and similarly, Gentleman-Sande can reconstruct the DFT from smaller DFTs. More details about the development and use cases of these algorithms can be easily found in literature and various online sources[2, 3].

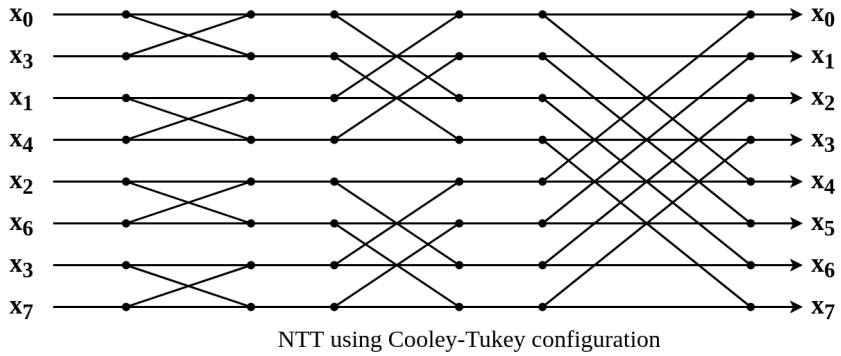

Figure 6.5: Cooley-Tukey NTT/INTT flow diagram

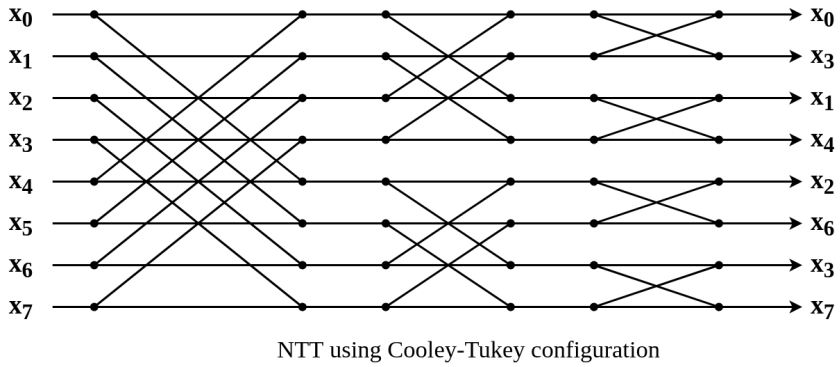

Figure 6.6: Gentleman-Sande NTT/INTT flow diagram

These iterative approaches have a run time complexity $\mathcal{O}(n(log\ n))$, as can be seen for the three nested loops. The inner part of the loop which is performing addition, subtraction, and multiplication is referred to as the Butterfly unit. We decided to separate the control unit, for the nested loops, and the data operations in the butterfly unit to make the implementation simple and readable. However, other than the multiplier in the butterfly unit we will require one more multiplier to generate the ζ values on the fly for different iterations. However, to avoid extra

multiplier cost, since the number of ζ values required are fixed, we can also pre-generate and store them in a BROM(Block Read Only Memory), using the Alg. 31. Since our design goal is lightweight and efficient architecture, we decided to go with the latter. It can be seen that in every iteration we only perform one multiplication each for NTT/INTT. Thus reducing the design cost significantly.

Post-processing elimination

In the Inverse NTT (INTT) operation the resulting coefficients are scaled by $1/n$, which requires extra n multiplications. Dilithium's official software implementation [16] uses an extra loop at the end of the inverse NTT for scaling the output coefficients, as shown in [66]. In our implementation, these extra scaling-related multiplications are removed by processing the coefficients using the following equation [76] during INTT.

$$x/2 \mod q = (x \gg 1) + x[0] \times ((q+1)/2) \tag{6.1}$$

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. This optimization helps us reduce the area consumption as well as clock cycle significantly.

Algorithm 29: The Gentleman-Sande InverseNTT algorithm [66, 76]

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 : Pre-computed table of 2n-th roots of unity ζ^{-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**
        $S \leftarrow$
        $k \leftarrow k + 2t$
    $t \leftarrow 2t$
    $m \leftarrow m/2$
return

Algorithm 30: The Cooley-Tukey NTT algorithm [66]

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 : Pre-computed table of 2n-th roots of unity ζ, 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**
      $S \leftarrow$
      $k \leftarrow k + 2t$
    $t \leftarrow t/2$
    $m \leftarrow 2m$
  return

Algorithm 31: `Zeta_generation` $(\zeta, \zeta^{-1}, zeta\_table, zeta\_Inv\_table, p)$

Input : ζ primitive root-of-unity,ζ^{-1} Inverse primitive root-of-unity, prime modulus p
Output : $zeta\_table, zeta\_Inv\_table$ tables containing the pre-generated constants

$c[0] \leftarrow 1$
$d[0] \leftarrow 1$
for $i = 0 to N - 1$ **do**
  $a[i] \leftarrow BitReverse(i)$
  $b[i] \leftarrow BitReverse(i) + 1$
  if $i! = 0$ **then**
    $c[i] \leftarrow (c[i-1] \cdot \zeta)\%p$
    $d[i] \leftarrow (d[i-1] \cdot \zeta^{-1})\%p$
$c[0] \leftarrow 0$
$d[0] \leftarrow 0$
for $i = 0 to N - 1$ **do**
  $\zeta\_table[i] \leftarrow c[a[i]]$
  $\zeta\_Inv\_table[i] \leftarrow d[b[i]]$
return $zeta\_table, zeta\_Inv\_table$

Internal architecture of NTT

To further reduce the multiplication cost we decided to implement a common butterfly unit for both NTT and INTT transformations as shown in Alg. 35. Fig. 6.7 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 subtractor, are all pipelined in a total of ten stages to achieve high clock frequency. The multiplier is pipelined in four stages and the modular reduction unit is pipelined in three stages. There are three more pipeline stages other than this in the butterfly unit to achieve a high-clock frequency.

Algorithm 32: Butterfly(alg,p1_in,p2_in,ζ,p1_out,p2_out,p)

Input : alg- 0 for NTT and 1 for INTT, (p1_in,p2_in) Input coefficients, ζ, prime p
Output : (p1_out,p2_out) Output coefficients
$t \leftarrow$(p1_in - p2_in)%p
$t\_mux \leftarrow$ alg ? t : p2_in
$u \leftarrow ModReduce(\zeta \cdot t\_mux)$
$v \leftarrow$ alg ? p2_in : u
$w \leftarrow$ (p1_in - u)%p
p1_out $\leftarrow$ (p1_in + v)%p
p2_out $\leftarrow$ alg ? u : w
return (p1_out,p2_out)

An example of the memory pattern across a single BRAM for NTTon a polynomial of coefficient size- 8 is shown in Fig. A.2 for an NTT iteration when $n = 8$. The memory pattern for INTT is shown in Fig.A.1.

As one butterfly core consumes two coefficients of size at most 25-bit each and simultaneously produces two coefficients, also of size 25-bit each, every cycle, we always keep two coefficients in a single 64-bit memory-word following [64]. That enables accessing two coefficients by just one memory-read and storing two coefficients by just one memory-write. This also helps reduce the memory requirement by half for storing a polynomial, as now we store the coefficients in pairs instead of separately. Using one such butterfly unit for NTT/INTT would require 1024 clock cycles. This is too high since we will be performing this operation for matrices up to an order of 8×7 for Dilithium. We will be spending a considerably huge amount of time doing NTT/INTT. We note that we can also unroll this NTT/INTT operation into as many butterfly units as we want to depend on the area available and design goals.

By using k butterfly units we will reduce the complexity of the NTT/INTT from $\mathcal{O}$(n(log n)) to $(\mathcal{O}(n(\log n)))/k$. However, butterfly units are also expensive units as they consist of a multiplier and quite a few additions and subtractions, including the ones from the modular reduction unit. Therefore, special care must be taken and the trade-off between the latency area must be considered before choosing an appropriate number of butterfly units. Also, note that this decision also depends on the read-write latency of coefficients from BRAM. In our case, we can only read 2 pairs of coefficients at a time. Even if we use more butterfly units they won't be

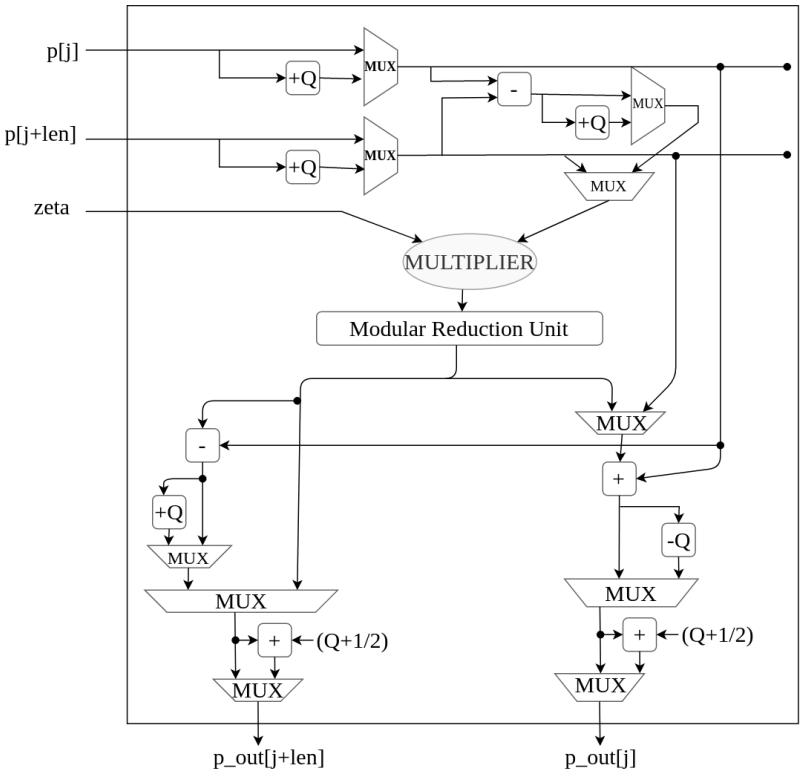

Figure 6.7: Internal architecture of the butterfly unit for unified Cooley-Tukey NTT and Gentleman-Sande INTT

fed in parallel. Since our butterfly unit is efficiently pipelined, it can already start processing the next start of coefficients read without waiting for the first pair of coefficients to finish.

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 can feed only both the butterfly cores partially, for the next set of values we will have to wait for another cycle before we can start the butterfly, due to the limitations in the number of read/write ports. Instead, if we use two BRAM each having their own read/write ports, and spread the coefficients in a way that the two coefficients used for a butterfly operation are either stored across the two BRAMs or together in the same BRAM. Thus making the read-write latency one. 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 is $\{N, N/2, N/4, \ldots, 4, 2\}$ during the outermost NTT-loops (Alg. 30), and $\{2, 4, \ldots, N/4, N/2, N\}$ during the outermost inverse NTT-loops (Alg. 29). 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.

With this approach, NTT or inverse NTT operation takes 512 clock cycles only.

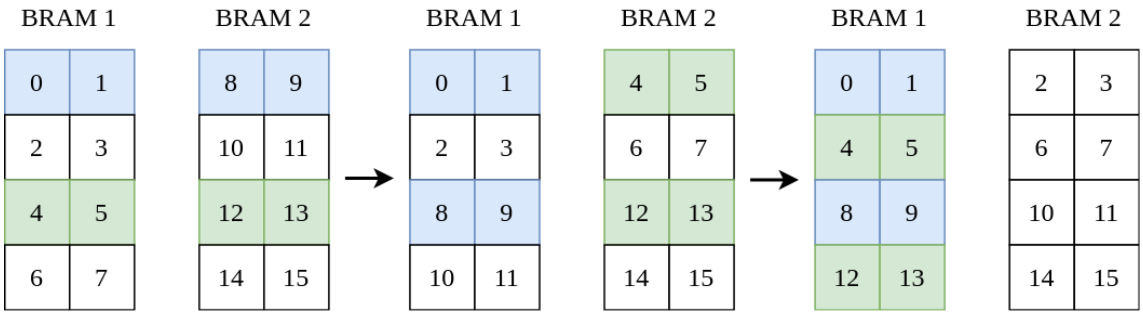

Figure 6.8: Coefficients storage for first 3 steps of the first iteration of NTT transformation on a polynomial of coefficient size 16

Fig. 6.8 shows the arrangement of coefficients in memory words during NTT loop-iterations using a toy example.

Using the efficient storage of coefficients across the two BRAMs, in every iteration of the outermost NTT-loop, we process the coefficients stored at the same address in both the BRAMs. For simplicity when we refer to coefficient i we refer to coefficient stored at index $\lfloor i/2 \rfloor$. For a polynomial with 16 coefficients, the first three steps in the first interactions are shown in Fig/6.8. We can see the coefficients are stored pairwise across the two BRAMs.

In the first iteration, we read coefficients zero-and-one, and eight-and-nine. The coefficients 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 is so that in the next iterations the coefficients required are stored across the two BRAMs. Because coefficients four-and-five have not been processed yet, we can not write new values to their memory location.

We solve the above problem in the following way. To be able to write processed coefficients to this address, we read the coefficients four-and-five, and similarly twelve-and-thirteen immediately after reading the coefficients zero-and-one, and similarly eight-and-nine. But what about the read/write conflicts that will arise when we write the processed coefficients with index values eight-and-nine in the memory location of the coefficients with index values four-and-five and simultaneously read coefficients four and five for the next step? This won't be a problem because everything is pipelined using 10 pipeline stages. So the processed output will be produced after 10 clock cycles. By this time we would have already read the next 10 coefficients, thus avoiding any read-write conflict.

Now that we have discussed how to efficiently perform NTT/INTT transformations, in the next section, we discuss how to efficiently implement the pointwise arithmetic operations.

6.5 Pointwise addition, subtraction, and multiplication

The pointwise arithmetic operations are much simpler to implement and also require fewer clock cycles. We just need to load coefficient[i] of both the polynomials and add, subtract, or multiply depending on the instruction. Addition and subtraction can be performed using simple adders and subtractors, followed by adding a simple check to keep the coefficients positive and less than the modulus. The multiplication unit however will require a similar modular reduction unit that we used for the NTT/INTT transformation. If we make fresh implementations for each of these, the addition and subtraction unit won't be very expensive. It will have a control part responsible for loading the coefficients of two polynomials and writing the added or subtracted values back. However, the multiplication block will not only have the control part but also an extra multiplier and modular reduction unit, which are more expensive than simple addition and subtraction.

In the previous section, we have decided to store the two coefficients in a single memory word to save up memory and read more coefficients at a time. WE can use this to optimize our implementation of the pointwise arithmetic operations. We can process two additions, subtractions and multiplications at once, as shown in Alg.37 for addition, Alg. 36 for subtraction, and Alg. 33 for multiplication . To optimize this further, we can implement a common control unit for reading and writing the coefficients. We just choose which operations are to be performed on the coefficients depending on the instructions.

Algorithm 33: `PointwiseMultiplicationOf2polynomials` (a, b, c)

Input : a, b Polynomials in $\mathcal{R}_q$
Output : $c[i] = a[i] \cdot b[i] \forall i \in \{0, N\}$ Polynomial in $\mathcal{R}_q$
$N \leftarrow LengthOfPolynomial(a \text{ or } b)$
for *i=0 to N-1,i=i+2* **do**
    $t1 \leftarrow a[i] \cdot b[i]$
    $t2 \leftarrow a[i] \cdot b[i]$
    $c[i] = ModReduce(t1)$
    $c[i+1] = ModReduce(t2)$
return c

However, since we already have adders, subtractors, and multipliers in the butterfly unit we can use them to give us the desired result for simple addition, subtraction, and multiplication by modifying the input in the way shown in Alg. 34. Thus, we only spend the resources required for a common control unit. For performing operations on data, we use the arithmetic units which were already of the butterfly unit.

Algorithm 34: Butterfly_data_control(op,p_in[N],p_out[N],ζ_in[N],p,j,len,k)

Input : (j,len,k)-loop variables, op- 0 for NTT, 1 for INTT, 2 for Pointwise Multiplication, 3 for Addition, and 4 for Subtraction, p_in[N]in Input polynomial, ζ_in[N]- table containing pre-generated powers of primitive root-of-unity , prime p

Output : p_out Output polynomial

alg $\leftarrow$ op=0 ? 0 : 1

p1_in $\leftarrow$ op=2 ? 0 : p_in[j]

p2_in $\leftarrow$ p_in[j+len]

$\zeta \leftarrow$ op=0 or op=1 ?ζ_in[j] : op==2 ? p_in[j] : 1

Butterfly(alg,p1_in,p2_in,ζ,p1_out,p2_out,p)

return (p1_out,p2_out)

6.6 Memory

Four our proposed architecture we decide to pre-generate all the roots-of-unit to avoid computing them on the fly, which would require extra multipliers. For storing the pre-generated root-of-unity for both Saber and Dilithium we will require 1 BRAM. Next, we also require some storage for storing the polynomials which will be read for different polynomial arithmetic operations. The NTT/INTT implementation is an in-place operation. Therefore, it does not require extra storage, and 2 BRAMs each having their read and write ports are sufficient.

With 2 BRAMs we can store half the polynomial in each BRAM and implement the simplified NTT/INTT transformation described above. For pointwise operations, we can always fetch 2 coefficients from each polynomial at once if the two polynomials are alternately stored in the two BRAMs as shown in Fig. 6.9. Thus, improving the cycle count. We also require one 18k BRAM or 0.5 BRAM to store the few instructions we receive. So in total, our proposed polynomial arithmetic architecture requires 3.5 BRAMs.

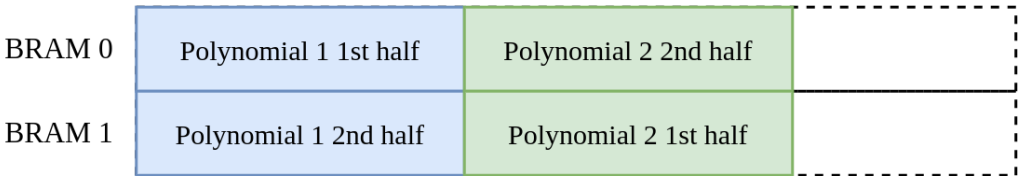

Figure 6.9: Storage of two polynomials across the two BRAMs to facilitate the parallel access for pointwise arithmetic operations

Instruction code	Operation
0	Null instruction
1	NTT
2	Inverse NTT
3	Coefficient-wise multiply
4	Coefficient-wise add
5	Coefficient-wise subtract
6	Null instruction
7	End instruction

Table 6.2: Table showing the instructions in two columns. An instruction from the first column can be run in parallel with an instruction from the second column.

6.7 Program Controller for Instruction set Architecture

The proposed design is an instruction set architecture. The user has to first send load all the polynomials into the memory. Then he sends the instructions for the desired operation on these polynomials and waits for the done signal. Once he receives the done signal he then fetches the result back from the memory. The instruction can be one of the polynomial arithmetic instructions listed in Tab.6.2. Instruction is sent in the format given in Fig. 6.10. The instruction contains a 3-bit instruction to specify which operation to perform and the two read addresses and 1 write address each of size 8-bits. The NTT/INTT is an in-place operation- meaning they read from and write to the same addresses. Making them write to another address will complicate the implementation. As they require the same coefficient to be fetched, processed and written back multiple times. So NTT/INTT requires only Read_address_1/ as this will also be the address they will be writing to. The pointwise arithmetic operations use the two read addresses to read the two polynomial coefficients and write the result to the given write address. This can be used as a partial in-place operation as well- implying that the resultant values can be written to one of the polynomial addresses as well.

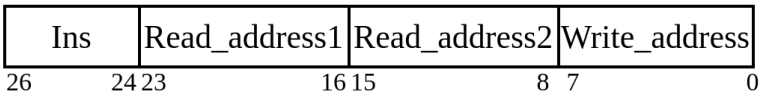

Figure 6.10: Format of the instructions for the designed cryptoprocessor

These instructions are sent one by one to the Instruction memory. Once all the instruction are received the special program controller reads and send the first instructions to the polynomial arithmetic unit and waits for the dance signal. Once the first instruction is done it read the next instruction, sends it, and waits for the done signal. This continues until all the instructions are executed, following which the program controller sends a done signal to the user. The user is supposed to send

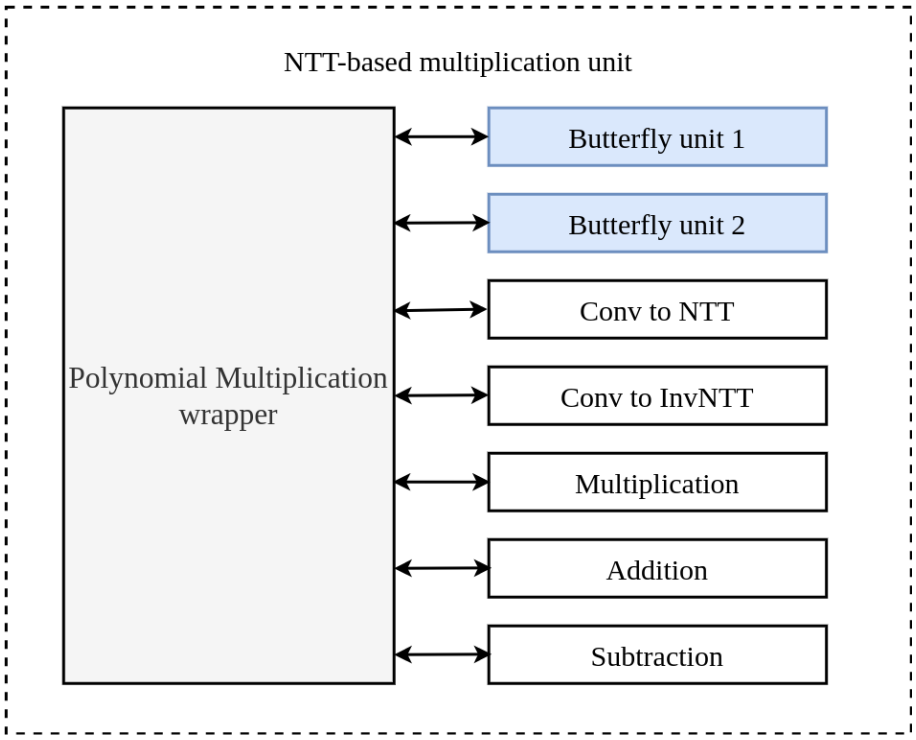

Figure 6.11: Architecture diagram of the Polynomial Arithmetic Unit

all the instructions at once followed by 'end instruction' in the end. The program controller gets to know this is the last instruction by reading the 'end instruction'. This is discussed in brief in [11].

To provide a summary of our implementation, we started by selecting a 'special' prime modulus for Saber to implement a common polynomial arithmetic unit. Then we designed an efficient and unified modular reduction unit, which is an integral part of the polynomial multiplier. Next, we chose the Cooley-Tukey NTT transformation, and Gentleman-Sande INTT transformation as the best methods for our design and perform post-processing elimination on the INTT transformation algorithm to make the NTT and INTT unit have the same cost. We saw the internal architecture of the unified butterfly unit for both NTT/INTT. We then use the adders, subtractors, and multipliers present in the butterfly unit to perform the pointwise arithmetic operations, thus saving area. Finally, we discussed the required Memory and how we make our design an instruction-set architecture. Thus, we realize an efficient architecture for the polynomial arithmetic unit. The overview of the design components in the proposed architecture is shown in Fig. 6.11.

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.

Chapter 7

Results

In this chapter, we discuss our results of implementation on the FPGA and then in the next section compare them with other implementations.

7.1 Our Results

Operation	Cycle count
NTT/INTT	522
Pointwise addition/subtract/multiplication	138
One full polynomial multiplication	1704
⌊ 2 polynomial NTT	⌊522×2
⌊ 1 pointwise multiplication	⌊138
⌊ 1 polynomial INTT	⌊522

Table 7.1: Cycle count for different operations

The proposed efficient polynomial arithmetic architecture is described entirely in hardware definition language- Verilog. Vivado 2019.1 is used to synthesize and implement the proposed architecture for the three different primes for Saber - 23-bit prime($2^{23} - 2^{13} + 1$), 24-bit prime($2^{24} - 2^{14} + 1$), and 25-bit prime($2^{25} - 2^{14} + 1$). We target the platform **Zynq Ultrascale+ ZCU102** with the strategy being performance and area explorer. The utilization numbers are given in Tab. 7.2 and the utilization percentage is given in Fig.7.1. Fig. 7.2 shows the area occupied on the FPGA. The proposed architecture runs at a clock frequency of $\approx 270MHz$.

From the results, we can see that there is not much difference in terms of area consumption for the three different primes. Even though the modular reduction unit for the Dilithium prime is not a unified unit and instead it's dedicated to only one prime. Also, the input-output size for the butterfly unit is 23-bits only. The clock cycle count for different operations in the architecture is given in Tab. 7.1. The clock cycles include the extra 10 clock cycles consumed by pipelining stages to increase the clock frequency of the design.

In the next section, we compare these results with the other implementations of polynomial multipliers for Saber and Dilithium in the literature.

$2^{23} - 2^{13} + 1$	LUT	FF	DSP	BRAM
Wrapper	3,347	1,594	4	3.5
⌊Program controller	⌊617	⌊350	⌊0	⌊0.5
⌊Compute wrapper	⌊2,730	⌊1,244	⌊4	⌊3
⌊⌊Polynomial Arithmetic unit	⌊⌊ 2,430	⌊⌊ 1,044	⌊⌊ 4	⌊⌊1
⌊⌊ BRAM for polynomial storage	⌊⌊ 0	⌊⌊ 0	⌊⌊ 0	⌊⌊ 2
$2^{24} - 2^{14} + 1$	LUT	FF	DSP	BRAM
Wrapper	3,371	1,605	4	3.5
⌊Program controller	⌊617	⌊350	⌊0	⌊0.5
⌊ Compute wrapper	⌊2,754	⌊1,255	⌊4	⌊3
⌊⌊Polynomial Arithmetic unit	⌊⌊ 2,454	⌊⌊ 1,055	⌊⌊ 4	⌊⌊ 1
⌊⌊ BRAM for polynomial storage	⌊⌊ 0	⌊⌊ 0	⌊⌊ 0	⌊⌊ 2
$2^{25} - 2^{14} + 1$	LUT	FF	DSP	BRAM
Wrapper	3,377	1,634	4	3.5
⌊Program controller	⌊617	⌊350	⌊0	⌊0.5
⌊Compute wrapper	⌊2,760	⌊1,284	⌊4	⌊3
⌊⌊Polynomial Arithmetic unit	⌊⌊ 2,460	⌊⌊ 1,084	⌊⌊ 4	⌊⌊1
⌊⌊ BRAM for polynomial storage	⌊⌊ 0	⌊⌊ 0	⌊⌊ 0	⌊⌊ 2

Table 7.2: Implementation utilization results

7.2 Comparison with other results

We compare our work with different related implementation works in terms of both performance and area consumption. The comparisons are made against hybrid architectures which support multiple PQC schemes including at least one of the two schemes used in our design, and dedicated implementations of Saber and Dilithium. In the works [17, 41, 42] architectures supporting multiple schemes are proposed. Sapphire [17] supports multiple schemes which made it to Round-2 of the NIST's standardization project, however, the specifications are old for Dilithium implementation and it does not provide results for Saber.

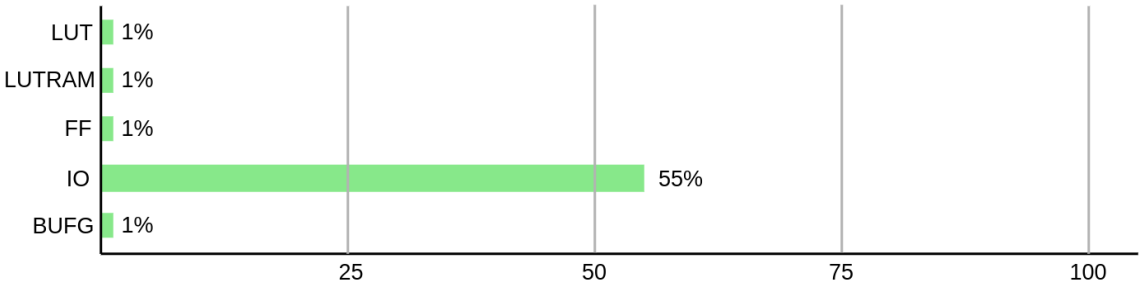

Figure 7.1: Utilization % on the FPGA

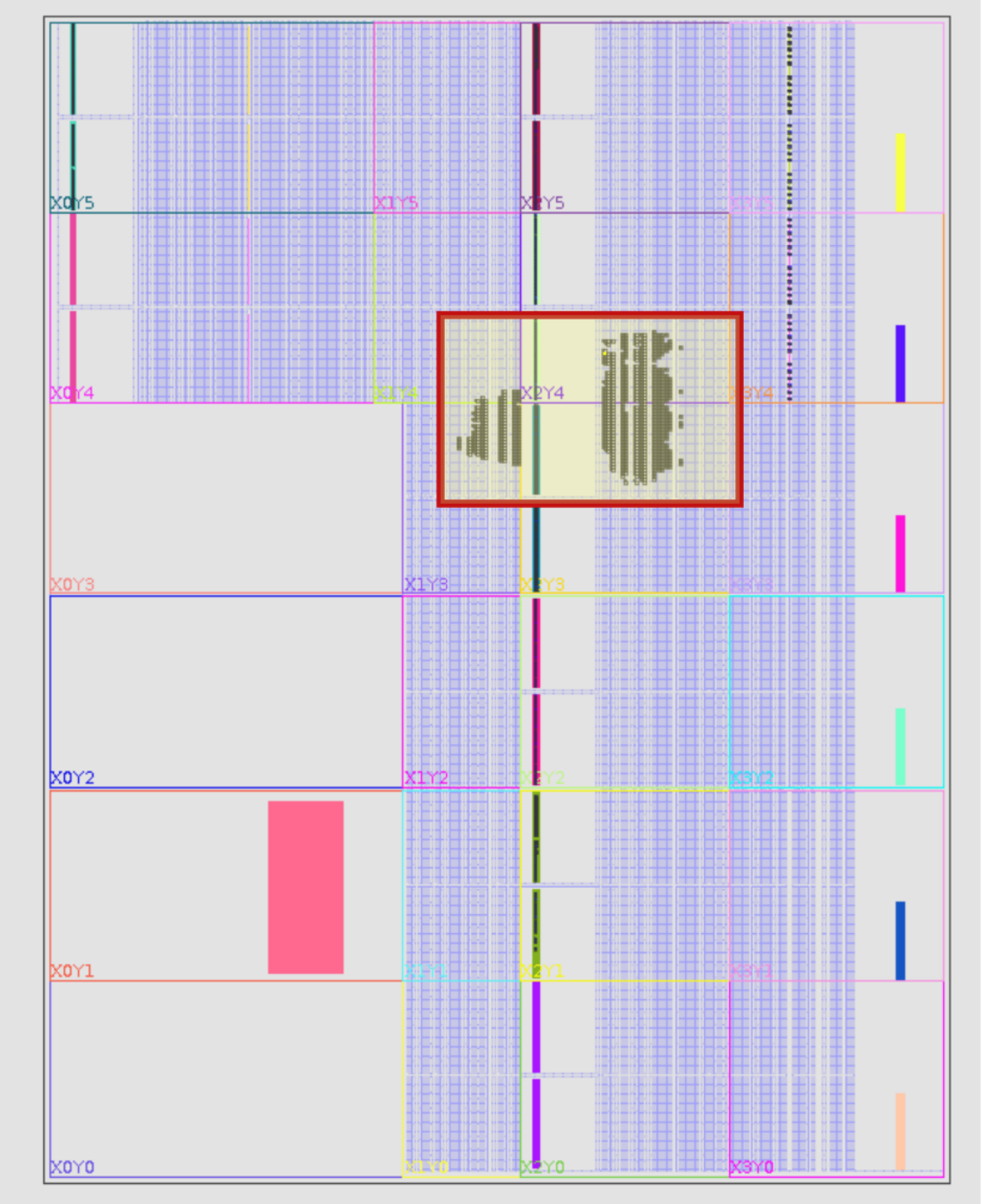

Figure 7.2: Implementation Diagram

In [41] the authors implement multiple key encapsulation and public key encryption schemes, one of which is Saber. They use three different primes for Saber and break down the polynomial into three residue polynomials, perform polynomial multiplication and convert them back using CRT. Note that this kind of approach is used for homomorphic implementations when the polynomials and moduli are huge. This is not the case for PQC schemes, and therefore, we conclude that this optimization although nice and novel, leads to an unnecessary blowup of both area and clock cycle counts. Their single accelerator for just NTT and modular arithmetic consumes 2,908 LUTs and 9 DSPs which is much higher compared to our entire Polynomial Arithmetic unit consumption. They perform the implementation for Xilinx Zynq-7000 and achieve a very low clock frequency of $\approx 77MHz$ on ASIC which is very low compared to our design. A Hardware/Software co-design is proposed in [42] for implementing Saber and Kyber. They prefer using Karatsuba/Toom-Cook over NTT based polynomial multiplication and end up spending a lot of area and cycle count giving our architecture a huge advantage in terms of both area and performance.

Various works implementing the KEM-PKE scheme Saber are present in the literature. In [63] the authors present a fast implementation of Saber, using the Schoolbook multiplication method for polynomial multiplication. They use $n = 256$ MAC units and thus lower the complexity from $\mathcal{O}(n^2)$ to $\mathcal{O}(n)$. Thus they only require 256 clock cycles plus some additional pipeline stages to perform one polynomial multiplication. They have a much better cycle count compared to our implementation however, our area consumption is much less, as we only require 2 multipliers. Thus, making our implementation a better option for resource-constraint devices. Dang et. al. [29] present a hardware/software co-design implementation of three different Round 2 PQC submissions FrodoKEM, Round5 , and Saber. For Saber, they also use the same approach as [63] and therefore spend 256 DSP units which is very high compared to our architecture. In [44] the authors implement Saber and propose a Scalable Matrix Originated Processing (SMOP) strategy for polynomial multiplication. Their area consumption and performance of the unified architecture are almost comparable to our design. However, we support all variants of Dilithium as well with the same architecture. Their design does not perform NTT transformation and therefore will not be able to support Dilithium. Thus we have an advantage considering the use case of the architecture.

[10] proposed both unmasked and masked implementation of Saber on Artix-7. They reduce the MAC units from 256 to 32 for their schoolbook multiplication based polynomial multiplier, thus, decreasing the area consumption but in turn increases the cycle count significantly to 2,048 for one polynomial multiplication. Our polynomial arithmetic architecture not just consumes less area but also has a smaller cycle count of 1704 clock cycles for single polynomial multiplication. This work helps us see a clear difference in performance and area between the high-speed optimized schoolbook based polynomial multiplication and NTT based polynomial multiplication depending on the number of MAC units and butterfly units. Toom-cook multiplication is used in [53] and the design is implemented for FPGA Zynq-7000. Our results are better, cause even though they show low area but they have a much

lower clock-frequency. Our design outperforms both [10, 53] and shows similar performance compared to the architectures in [29, 44]. The high-speed implementations [46, 79, 63] although report a better performance for Saber, the design is not scalable for Dilithium as our design does. Thus, we get the advantage of supporting multiple schemes for multiple primes.

Dilithium is not very widely implemented in literature because the fixation on the use of NTT leaves less room for exploration. Nevertheless, [78, 62, 50, 20] present in the literature provide efficient implementations of Dilithium. Zhou et al. [78] proposed a hardware-software co-design of Dilithium. Their NTT unit consumes 2,044 LUTs, 16 DSPs, 6 BRAMs, and 1,170 clock cycles at a clock frequency of 216 MHz. They do not pre-generate all the twiddle factors and do not use the Post-processing elimination. From the area consumption, it can be seen that they not only consume a higher area but also require more clock cycles for NTT transformation. Ricci et. al. [62] present a high-performance architecture which consumes 2,547 LUTs, 3889 FFs, 84 DSPs, and 3.5 BRAMs. They achieve a very high frequency of 637 MHz. Their design consumes much more area than our architecture and is therefore not suitable for a resource constraint device. The work in [50] presents a Dilithium implementation for FPGA. They target reducing LUT utilization by employing extra DSP units for computations. They also make use of the efficient modular reduction unit. However, they end up consuming 45 DSPs which is much higher compared to our design. DSP units aside, the LUT, FF, BRAM, and clock cycle count are almost the same as our design. In [20], a high performance Dilithium implementation is presented. It consumes 9018 LUTs, 6,292 FFs, and 16 DSPs. This is almost four times more than our area consumption with a clock cycle count almost comparable to our design. They do not mention a separate clock frequency for the multiplication unit but their overall design frequency for Dilithium V is 256 MHz.

Thus, from all these comparisons we can conclude that our design provides the best performance and least area consumption for resource-constraint devices in the case of Saber and a better area and performance in general compared to Dilithium implementations.

Chapter 8

Future scope

There is scope for improving the clock cycle count for pointwise polynomial operations-addition, subtraction, and multiplication, as well as the NTT/INTT transformations. Currently, two coefficients are processed at a time. If we store the polynomials across 4 BRAMs instead of 2 BRAMs and use two more butterfly units then the clock cycle count for all of these operations will be reduced by half. However, this will increase the area, For devices that have enough resources available, this is an apt solution. An overview of the proposed architecture is given in Fig. 8.1.

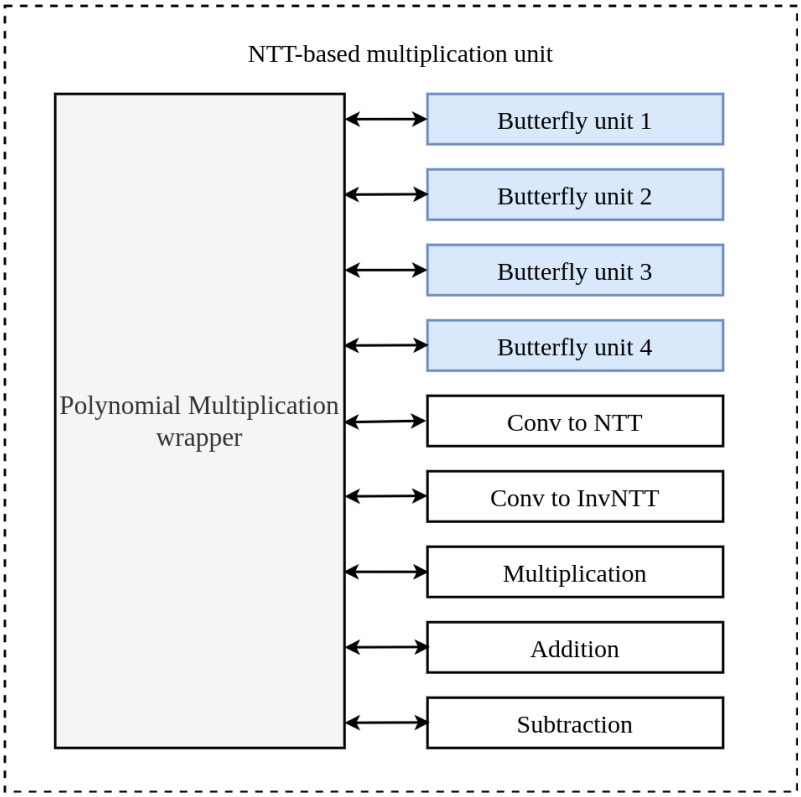

Figure 8.1: Proposed new Architecture for the Polynomial Arithmetic Unit

There are many different directions in which this work can be extended. The

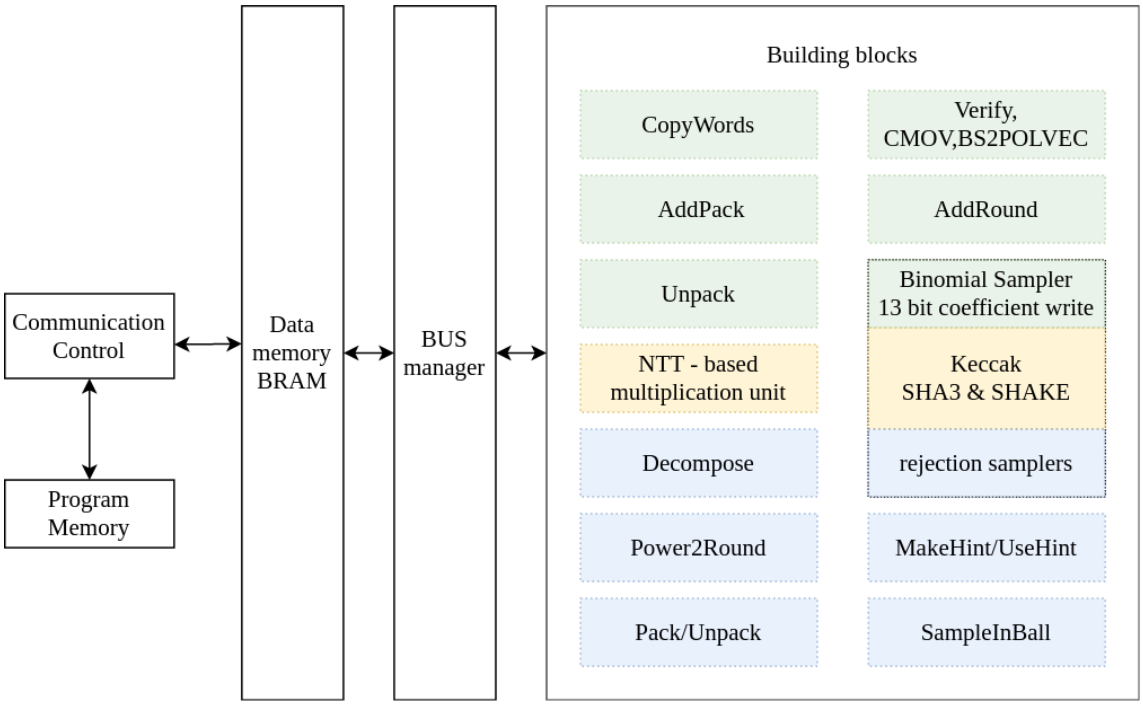

Figure 8.2: Unified cryptoprocessor for Saber and Dilithium

first scope is to include the other building blocks of Saber and Dilithium and make a complete cryptoprocessor as shown in Fig. 8.2. This would require adding another expensive module- Keccak. Both Saber and Dilithium require Keccak for generating pseudo-random numbers and hash. Saber uses a binomial sampler whereas Dilithium uses a rejection sampler of multiple types for generating different polynomials. Dilithium also requires many different types of packing which will require several big buffers. This might lead to a significant increase in the design cost and lower the clock frequency of the design as well. The two major blocks- Keccak and polynomial multiplication will be common. The remaining building blocks having complexity $\mathcal{O}(n)$ are not expected to be expensive to implement. However, synergies between the design of the two schemes should be exploited to maintain the lightweightedness of the architecture.

Another direction of extension is including support for Kyber's polynomial multiplication. Kyber also is a lattice-based design with polynomials of coefficients size $n = 256$. It uses a small prime moduli $q = 3{\cdot}2^{10}{+}2^8{+}1$ even smaller than Dilithium's prime. For Kyber we can store 4 coefficients in a 64-bit word, and use the existing 2 butterfly core as 4 butterfly cores. Thus processing one full NTT in half the clock cycles as that required by Saber or Dilithium, without a huge increase in the area. This will only require some changes in the control unit of the multiplier. The structure of the Kyber prime is sparse and it stays the same for all Kyber variants. Therefore, we can again make a highly optimized dedicated unified modular reduction unit for the three schemes. This would work fine for a normal case, but Kyber computes incomplete NTT/INTT, i.e., it stops at an earlier level during NTT and starts from this higher level during INTT. This is because the Kyber prime contains

primitive 256-th roots of unity but not primitive 512-th roots. So, in Kyber after the NTT transformation we get 1 degree polynomials, i.e., polynomials having two coefficients and therefore, our existing pointwise multiplication approach won't work directly. But with simple modifications in the control unit, this can be supported by using the Karatsuba optimization.

To summarize the discussion, there are many possible use cases and also future scopes of this work. It involves making this a full-fledged cryptoprocessor that supports both key encapsulation and public key encryption scheme and digital signature algorithm. Another way is to include more such schemes to provide for different applications.

Chapter 9

Conclusions

In this work, we use two finalists of the NIST' PQC standardization project- Saber, a Public Key Encryption and Encapsulation scheme, and Dilithium, a digital signature algorithm. The two schemes are lattice-based and have many similarities discussed earlier, however, while Saber has a power-of-two modulus, Dilithium has a prime modulus. Thus implementing a common polynomial arithmetic unit was a challenging and much-needed task to show that these two schemes can co-exist in a single processor without occupying much area.

The most expensive operations in both Saber and Dilithium are polynomial arithmetic operations and Keccak based hash and pseudo-random number generation. Both the schemes can use the same Keccak code with different wrappers. For the polynomial multiplication unit, Saber's official implementation uses a high-speed well-optimized schoolbook multiplier, however, Dilithium, because of its prime moduli, efficiently use the NTT based polynomial multiplication unit. Having two different expensive polynomial multiplication units will increase the area consumption. We solved this problem by designing an NTT-based polynomial multiplication unit for both the schemes and for this we use a special prime modulus for computing the NTTs of Saber. This greatly reduces the area overhead of the unified multiplier.

The most expensive unit in a polynomial multiplication unit is the modular reduction operation. We discussed several strategies which are used for implementing an efficient modular reduction unit in literature and also the ones used in the official implementations of these schemes. We selected the special primes for Saber in a way that an efficient and dedicated unified modular reduction unit can be designed. We showed the three different prime candidates for Saber based on the acceptable failing probabilities and discussed the advantages and disadvantages of using them. We implemented the polynomial multiplier for all three primes to give a fair comparison. We also removed the post-processing elimination iteration given in the Gentleman-Sande INTT transform by using simple addition and shift in every INTT iteration.

After implementing all the above carefully chosen and optimized design strategies, our polynomial arithmetic architecture consisting of the polynomial arithmetic unit and also an instruction program controller utilizes 3,377 LUTs, 1,634 FFs, 4 DSPs, and 3.5 BRAMs on a Xilinx FPGA. The polynomial arithmetic unit alone

consumes 2,460 LUTs, 1,084 FFs, 4 DSPs, and 1 BRAM. The unit has 10 pipeline stages and results in a high clock frequency of $\approx$ 270Mhz. One NTT/INTT operation takes 522 clock cycles and one pointwise addition/subtraction/multiplication requires 138 clock cycles. The proposed design is much faster compared to many other Dilithium-only implementations and consumes much less area compared to many Saber-only implementations on hardware platforms. It supports all the variants of Saber and Dilithium based on different security levels.

In the future, we intend to add the remaining building blocks of the two schemes to make a full-fledged cryptoprocessor implementation while maintaining the design goal of the architecture being lightweight. We also intend to add the polynomial arithmetic unit of other PQC schemes like Kyber. Kyber being lattice-based is expected to share some similarities with these two lattice-based schemes we have already implemented.

In this thesis, we showed how to design a flexible, compact, and lightweight polynomial arithmetic architecture for computing the lattice-based post-quantum finalist schemes Crystals-Dilithium for digital signature, and Saber for key encapsulation mechanism. To reduce the area consumption, we implemented a common NTT-based multiplication unit and made both algorithmic and structural optimizations to reduce the amount of memory and avoid the read-write conflicts. This is a completely instruction-level architecture, thus giving the users flexibility to perform the instruction that they wish to perform instead of performing a complete NTT-based multiplication and other combinations. The design supports all the variants of Saber and Dilithium and consumes only a small portion of present-day FPGAs.

Appendix A

Appendix

A.1 Abbreviations

PQC	Post-Quantum cryptography
NIST	National Institute of Standards and Technology
MLWR	Module Learning With Rounding
MLWE	Module Learning With Errors
MSIS	Module Short Integer Solution
KEM	Key Encapsulation Mechanism
PKE	Public Key Encryption
CCA	Chosen Ciphertext Attack
CPA	Chosen Plaintext Attack
XOF	expandable output function
TMVP	Toeplitz Matrix Vector Product
NTT	Number Theoretic Transform
MAC	Multiply And Accumulate
INTT	Inverse Number Theoretic Transform
FFT	Fast Fourier Transform
DFT	Discrete Fourier Transform
RNS	Residue Number System
CRT	Chinese Remainder Theorem
RSA	Rivest–Shamir–Adleman
DSA	Digital Signature Algorithm
DSP	Digital Signal Processor(s)
LUT	Look-up-tables
FF	Flip Flops
BRAM	Block Random Access Memory
BROM	Block Read-Only Memory

A.2 Algorithms and Figures

Algorithm 35: Unified_Modular_Reduction(ctr,A,B,p)

Input : ctr- 0 23-bit prime or 1 25-bit prime, prime p, Input value A
Output : Output value B

$v1 \leftarrow$ ctr ? A[24:0] : A[22:0]
$v2 \leftarrow$ ctr ? A[49:25] : A[45:23]
$v3 \leftarrow$ ctr ? A[35:25]—— A[49:36] : A[32:23], A[45:33]
$v4 \leftarrow$ ctr ? A[46:36],11'b0, A[49:47] : A[42:33],10'b0, A[45:43]
$v5 \leftarrow$ ctr ? A[49:47],14'b0 : A[45:43],13'b0
$v6 \leftarrow$ ctr ? 0xDFFBFFB : 0xF7FDFFB
$R \leftarrow v1 + v2 + v3 + v4 + v5 + v6$
$w1 \leftarrow p+$ ctr ? A[24:0] : A[22:0] $w2 \leftarrow$ R[27:0] - p
$w3 \leftarrow$ R[27:0] - $2p$
$w4 \leftarrow$ R[27:0] - $3p$
$w4 \leftarrow$ R[27:0] - $4p$
if *R[27]!=0* **then**
    $B\_temp \leftarrow w1[24:0]$
else if $w5[27] == 0$ **then**
    $w5[24:0]$
else if $w4[27] == 0$ **then**
    $w4[24:0]$
else if $w3[27] == 0$ **then**
    $w3[24:0]$
else if $w2[27] == 0$ **then**
    $w2[24:0]$
else
    $R[24:0]$
B $\leftarrow$ ctr ? B_temp[24:0] : B_temp[22:0]
return B

Algorithm 36: `SubtractionOf2polynomials` (a, b, c)

Input : a, b Polynomials in $\mathcal{R}_q$
Output : $c = a - b$ Polynomial in $\mathcal{R}_q$
$N \leftarrow LengthOfPolynomial(a\ or\ b)$
for *i=0 to N-1, i=i+2* **do**
    $t1 \leftarrow q + a[i] - b[i]$
    $t2 \leftarrow q + a[i] - b[i]$
    if $t1 < q$ **then**
        $c[i] = t1$
    else
        $c[i] = t1 - q$
    if $t2 < q$ **then**
        $c[i+1] = t2$
    else
        $c[i+1] = t2 - q$
return c

Algorithm 37: `AdditionOf2polynomials` (a, b, c)

Input : a, b Polynomials in $\mathcal{R}_q$
Output : $c = a + b$ Polynomial in $\mathcal{R}_q$
$N \leftarrow LengthOfPolynomial(a\ or\ b)$
for *i=0 to N-1,i=i+2* **do**
    $t1 \leftarrow a[i] + b[i]$
    $t2 \leftarrow a[i] + b[i]$
    if $t1 < q$ **then**
        $c[i] = t1$
    else
        $c[i] = t1 - q$
    if $t2 < q$ **then**
        $c[i+1] = t2$
    else
        $c[i+1] = t2 - q$
return c

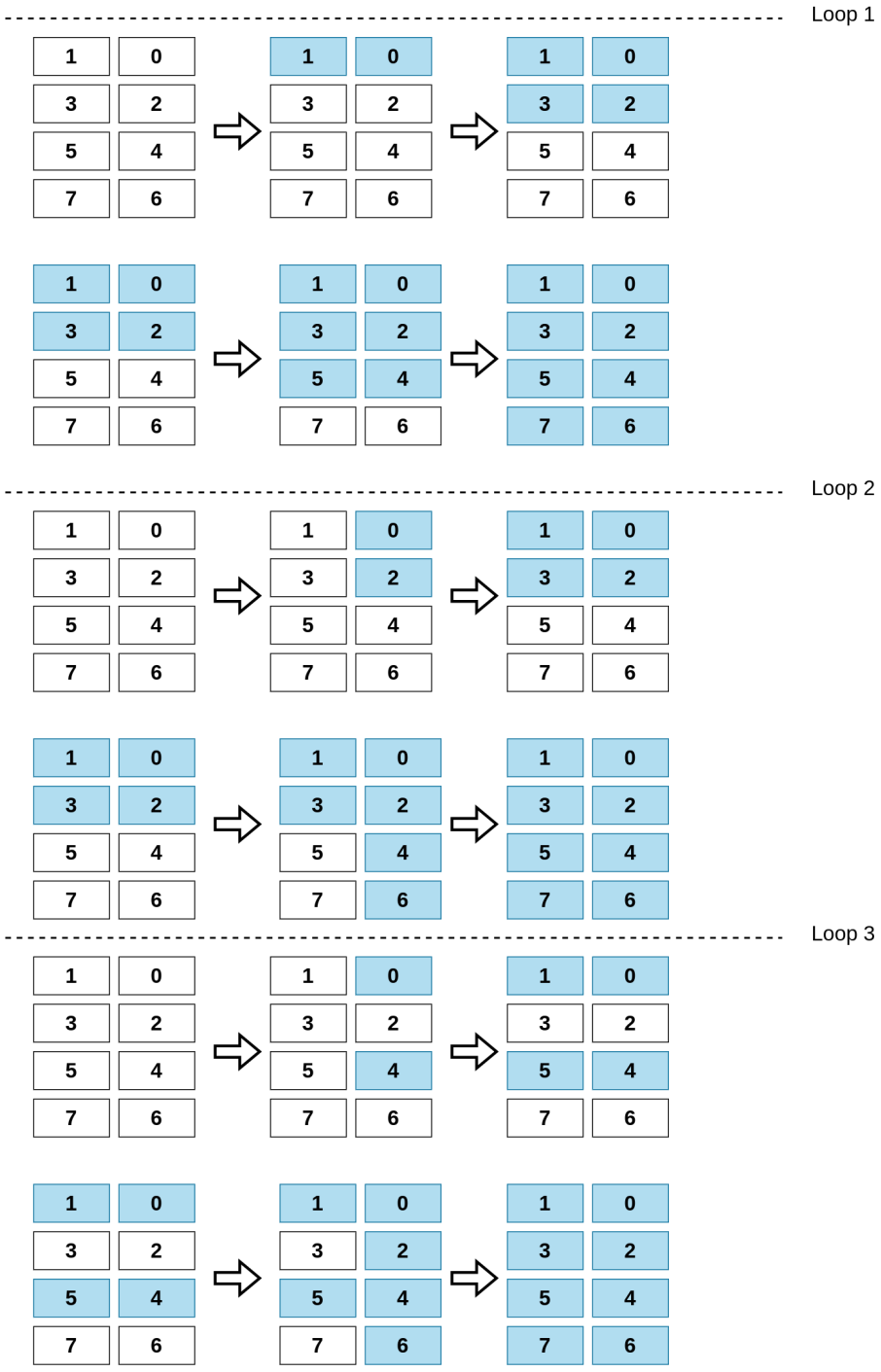

Figure A.1: Coefficients storage in single BRAM for full iterations of INTT on a polynomial having 8 coefficients

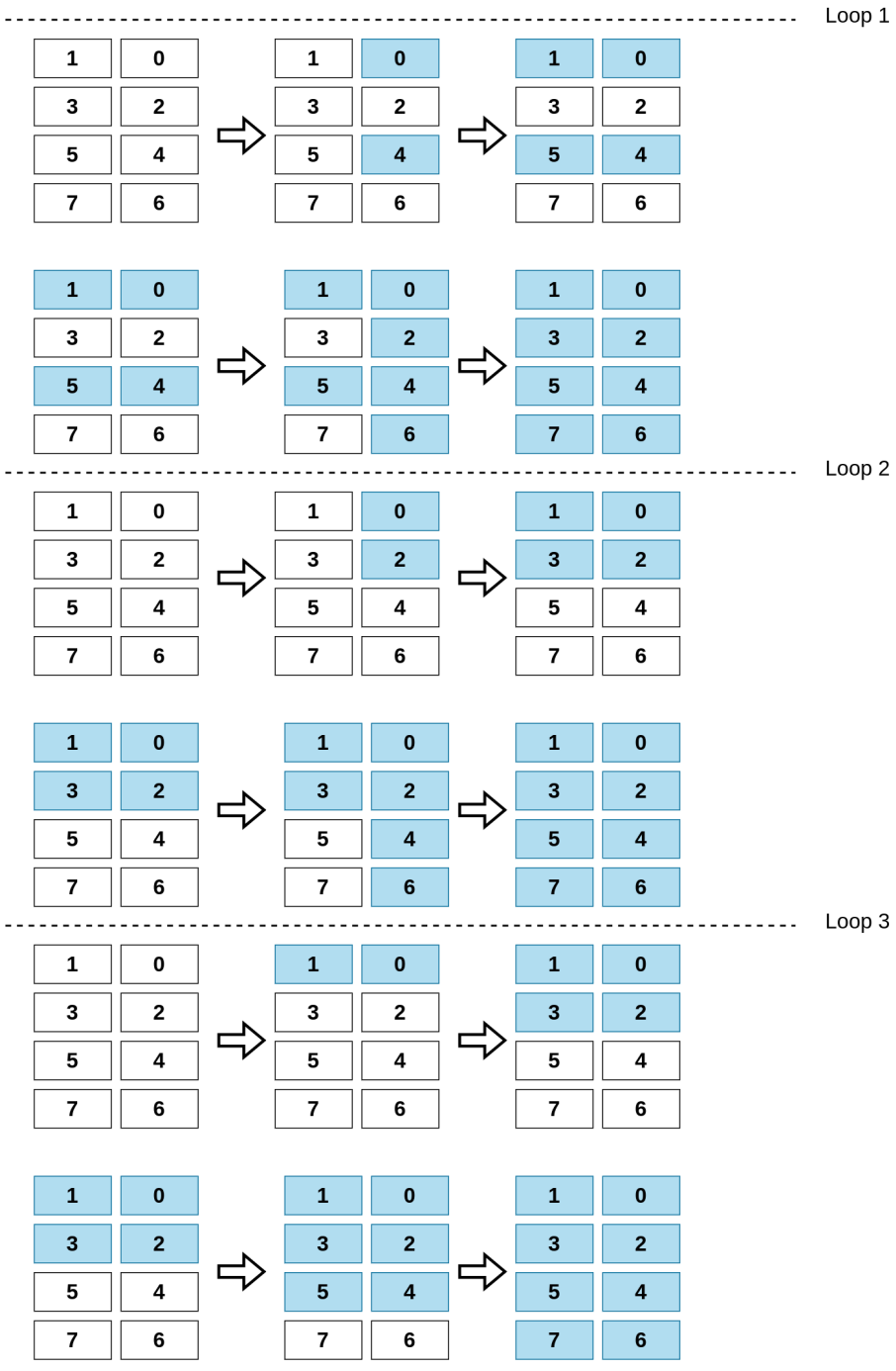

Figure A.2: Coefficients storage in single BRAM for full iterations of NTT on a polynomial having 8 coefficients

Bibliography

[1] Karatsuba vs grade multiplication explained. https://introtcs.org/public/lec_01_introduction.html.

[2] Online page explaining development and uses of cooley-tukey transformation. https://en.wikipedia.org/wiki/Cooley-Tukey_FFT_algorithm.

[3] Online page explaining development and uses of gentleman-sande transformation. http://dsp-book.narod.ru/FFTBB/0270_PDF_C03.pdf.

[4] Pages explaining barrett reduction in brief. https://www.nayuki.io/page/barrett-reduction-algorithm,https://en.wikipedia.org/wiki/Barrett_reduction.

[5] Pages explaining montgomery reduction in brief. https://www.nayuki.io/page/montgomery-reduction-algorithm,https://cryptography.fandom.com/wiki/Montgomery_reduction.

[6] Website explaining the chinese remainder theorem with examples. https://brilliant.org/wiki/chinese-remainder-theorem/.

[7] Wikipedia article on residue number system, note = https://en.wikipedia.org/wiki/residue_number_system.

[8] Wikipedia page for toom-cook multiplication. https://en.wikipedia.org/wiki/Toom-Cook_multiplication.

[9] NIST. 2015. SHA-3 standard: Permutation-Based Hash and Extendable-Output Functions. FIPS PUB 202, 2015.

[10] A. Abdulgadir, K. Mohajerani, V. B. Dang, J.-P. Kaps, and K. Gaj. A lightweight implementation of saber resistant against side-channel attacks. Cryptology ePrint Archive, Report 2021/1452, 2021.

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

[12] M. Ajtai. Generating hard instances of lattice problems (extended abstract). In G. L. Miller, editor, *Proceedings of the Twenty-Eighth Annual ACM Symposium on the Theory of Computing, Philadelphia, Pennsylvania, USA, May 22-24, 1996*, pages 99–108. ACM, 1996. https://doi.org/10.1145/237814.237838.

[13] 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.

[14] E. Alkim, L. Ducas, T. Pöppelmann, and P. Schwabe. Post-quantum key exchange - A new hope. In T. Holz and S. Savage, editors, *25th USENIX Security Symposium, USENIX Security 16, Austin, TX, USA, August 10-12, 2016*, pages 327–343. USENIX Association, 2016. https://www.usenix.org/conference/usenixsecurity16/technical-sessions/presentation/alkim.

[15] 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.

[16] 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.

[17] U. Banerjee, T. S. Ukyab, and A. P. Chandrakasan. Sapphire: A configurable crypto-processor for post-quantum lattice-based protocols. *IACR Trans. on CHES*, 2019(4):17–61, 2019. https://dblp.org/rec/journals/tches/BanerjeeUC19.bib.

[18] P. Barrett. Implementing the rivest shamir and adleman public key encryption algorithm on a standard digital signal processor. In A. M. Odlyzko, editor, *Advances in Cryptology — CRYPTO' 86*, pages 311–323, Berlin, Heidelberg, 1987. Springer Berlin Heidelberg.

[19] K. Basu, D. Soni, M. Nabeel, and R. Karri. Nist post-quantum cryptography-a hardware evaluation study. *IACR Cryptol. ePrint Arch.*, 2019:47, 2019.

[20] L. Beckwith, D. T. Nguyen, and K. Gaj. High-performance hardware implementation of crystals-dilithium. Cryptology ePrint Archive, Report 2021/1451, 2021. `https://ia.cr/2021/1451`.

[21] 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.

[22] 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*, pages 353–367. IEEE, 2018.

[23] Z. Cao, R. Wei, and X. Lin. A fast modular reduction method. *IACR Cryptol. ePrint Arch.*, 2014:40, 2014.

[24] C. Chen, O. Danba, J. Hoffstein, A. Hülsing, J. Rijneveld, J. M. Schanck, P. Schwabe, W. Whyte, and Z. Zhang. Ntru: Algorithm specifications and supporting documentation, 2019. https://ntru.org/f/ntru-20190330.pdf.

[25] 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`.

[26] 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. on CHES*, 2021(2):159–188, 2021. `https://dblp.org/rec/journals/tches/ChungHKSSY21.bib`.

[27] 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*, 2021(2):159–188, Feb. 2021.

[28] J. W. Cooley and J. W. Tukey. An algorithm for the machine calculation of complex fourier series. *Mathematics of Computation*, 19:297–301, 1965.

[29] 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 Int. Conf. on Field-Programmable Technology*, pages 206–214, 2019.

[30] 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.*, 2020:795, 2020. `https://eprint.iacr.org/2020/795`.

[31] 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.*, page 795, 2020. `https://dblp.org/rec/journals/iacr/DangFAMNG20.bib`.

[32] V. B. Dang, K. Mohajerani, and K. Gaj. High-speed hardware architectures and fpga benchmarking of crystals-kyber, ntru, and saber. Cryptology ePrint Archive, Report 2021/1508, 2021. `https://ia.cr/2021/1508`.

[33] 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`.

[34] 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.

[35] 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`.

[36] L. Ducas, E. Kiltz, T. Lepoint, V. Lyubashevsky, P. Schwabe, G. Seiler, and D. Stehlé. Crystals-dilithium: A lattice-based digital signature scheme. *IACR Trans. on CHES*, 2018(1):238–268, 2018. `https://doi.org/10.13154/tches.v2018.i1.238-268`.

[37] V. Ennola and R. Turunen. On totally real cubic fields. *Math. Comp.*, 44(170):495–518, 1985. `https://doi.org/10.2307/2007969`.

[38] H. Fan and M. Hasan. A new approach to subquadratic space complexity parallel multipliers for extended binary fields. *Computers, IEEE Transactions on*, 56:224–233, 03 2007.

[39] P.-A. Fouque, J. Hoffstein, P. Kirchner, V. Lyubashevsky, T. Pornin, T. Prest, T. Ricosset, G. Seiler, W. Whyte, and Z. Zhang. Falcon: Fast-fourier lattice-based compact signatures over ntru, 2018.

[40] T. Fritzmann, G. Sigl, and J. Sepúlveda. RISQ-V: tightly coupled RISC-V accelerators for post-quantum cryptography. *IACR Trans. Cryptogr. Hardw. Embed. Syst.*, 2020(4):239–280, 2020. `https://doi.org/10.13154/tches.v2020.i4.239-280`.

[41] T. Fritzmann, G. Sigl, and J. Sepúlveda. Risq-v: Tightly coupled risc-v accelerators for post-quantum cryptography. *IACR Trans. on CHES*, 2020(4):239–280, Aug. 2020.

[42] 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.*, 2021:479, 2021.

[43] 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`.

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

[45] M. Imran, Z. U. Abideen, and S. Pagliarini. A systematic study of lattice-based NIST PQC algorithms: from reference implementations to hardware accelerators. *CoRR*, abs/2009.07091, 2020. `https://arxiv.org/abs/2009.07091`.

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

[47] M. J. Kannwischer, J. Rijneveld, P. Schwabe, and K. Stoffelen. pqm4: Testing and benchmarking nist pqc on arm cortex-m4. Cryptology ePrint Archive, Report 2019/844, 2019. `https://ia.cr/2019/844`.

[48] A. A. Karatsuba and Y. P. Ofman. Multiplication of many-digital numbers by automatic computers. *Doklady Akademii Nauk*, 145(2):293–294, 1962.

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

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

[51] W. Liu, S. Fan, A. Khalid, C. Rafferty, and M. O'Neill. Optimized schoolbook polynomial multiplication for compact lattice-based cryptography on fpga. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 27(10):2459–2463, 2019.

[52] 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*, pages 598–616. Springer, 2009.

[53] 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 DAC*, pages 1–6, 2020.

[54] J. Mera, A. Karmakar, and I. Verbauwhede. Time-memory trade-off in toom-cook multiplication: an application to module-lattice based cryptography. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, pages 222–244, 03 2020.

[55] D. T. Nguyen and K. Gaj. Optimized software implementations using neon-based special instructions.

[56] I. K. Paksoy and M. Cenk. Tmvp-based multiplication for polynomial quotient rings and application to saber on ARM cortex-m4. *IACR Cryptol. ePrint Arch.*, page 1302, 2020. `https://eprint.iacr.org/2020/1302`.

[57] G. B. J. D. M. Peeters and G. V. Assche. The Keccak reference. Round 3 submission to NIST SHA-3, 2011. `http://keccak.noekeon.org/Keccak-reference-3.0.pdf`.

[58] 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`.

[59] J. Proos and C. Zalka. Shor's discrete logarithm quantum algorithm for elliptic curves. *Quantum Info. Comput.*, 3(4):317–344, July 2003.

[60] O. Regev. On lattices, learning with errors, random linear codes, and cryptography. In H. N. Gabow and R. Fagin, editors, *Proceedings of the 37th Annual ACM Symposium on Theory of Computing, Baltimore, MD, USA, May 22-24, 2005*, pages 84–93. ACM, 2005. `https://doi.org/10.1145/1060590.1060603`.

[61] L. A. D. S. Ribeiro, J. P. da Silva Lima, R. J. G. B. de Queiroz, A. B. Chagas, J. P. Quintino, F. Q. B. da Silva, A. L. M. Santos, and J. R. R. Junior. Saber post-quantum key encapsulation mechanism (kem): Evaluating performance in mobile devices and suggesting some improvements. `https://csrc.nist.gov/CSRC/media/Events/third-pqc-standardization-conference/documents/accepted-papers/ribeiro-saber-pq-key-pqc2021.pdf`.

[62] 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 Int. Conf. on ARES*, New York, NY, USA, 2021. Association for Computing Machinery.

[63] 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.*, 2020(4):443–466, 2020. `https://doi.org/10.13154/tches.v2020.i4.443-466`.

[64] S. S. Roy, F. Vercauteren, N. Mentens, D. D. Chen, and I. Verbauwhede. Compact ring-lwe cryptoprocessor. In *CHES 2014*, pages 371–391. Springer Berlin Heidelberg, 2014.

[65] 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`.

[66] 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*, pages 247–258. Springer, 2017. `https://dblp.org/rec/conf/ima/Scott17.bib`.

[67] J. A. Solinas. Generalized mersenne numbers. Technical report, 1999.

[68] 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.

[69] D. Sprenkels. The kyber/dilithium ntt. `https://dsprenkels.com/ntt.html`.

[70] M. Taschwer. Modular mutliplication using special prime moduli. 2001. `http://www.isys.uni-klu.ac.at/PDF/2001-0126-MT.pdf`.

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

[72] I. Toli. Efficient arithmetic for some finite fields. 2006. `https://www.ricam.oeaw.ac.at/specsem/srs/groeb/download/Toli_poster.pdf`.

[73] A. L. Toom. The complexity of a scheme of functional elements realizing the multiplication of integers. *Soviet Mathematics Doklady*, 3(4):714–716, 1963.

[74] A. Weimerskirch and C. Paar. Generalizations of the karatsuba algorithm for efficient implementations. *IACR Cryptol. ePrint Arch.*, 2006:224, 2006.

[75] G. Xin, J. Han, T. Yin, Y. Zhou, J. Yang, X. Cheng, and X. Zeng. Vpqc: A domain-specific vector processor for post-quantum cryptography based on risc-v architecture. *IEEE Transactions on Circuits and Systems I: Regular Papers*, 67(8):2672–2684, 2020.

[76] F. Yaman, A. C. Mert, E. Öztürk, and E. Savas. A hardware accelerator for polynomial multiplication operation of CRYSTALS-KYBER PQC scheme. In *DATE 2021, Grenoble, France, Feb. 1-5, 2021*, pages 1020–1025. IEEE, 2021. `https://dblp.org/rec/conf/date/YamanMOS21.bib`.

[77] N. Zhang, B. Yang, C. Chen, S. Yin, S. Wei, and L. Liu. Highly efficient architecture of newhope-nist on fpga using low-complexity ntt/intt. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, 2020(2):49–72, Mar. 2020. `https://tches.iacr.org/index.php/TCHES/article/view/8544`.

[78] 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.*, 14(2), June 2021.

[79] 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 Trans. on CAS I: Regular Papers*, 68(3):1146–1159, 2021.