Technische Universität München
TUM School of Computation, Information and Technology

Towards Secure Coprocessors and Instruction Set Extensions for Acceleration of Post-Quantum Cryptography

Tim Fritzmann

Vollständiger Abdruck der von der TUM School of Computation, Information and Technology der Technischen Universität München zur Erlangung eines

Doktors der Ingenieurwissenschaften (Dr.-Ing.)

genehmigten Dissertation.

Vorsitz:

Prof. Dr. Andreas Herkersdorf

Prüfer der Dissertation:

1. Prof. Dr.-Ing. Georg Sigl
2. Prof. Dr.-Ing. Daniel Müller-Gritschneder

Die Dissertation wurde am 16.03.2022 bei der Technischen Universität München eingereicht und durch die TUM School of Computation, Information and Technology am 08.11.2022 angenommen.

Abstract

The development of quantum computers has made significant advances in recent years. Large-scale quantum computers can achieve extreme speedups for solving complex problems. However, such speedups lead to a severe problem for currently deployed Public-Key Cryptography (PKC) because the mathematically complex problems it relies on might become solvable using a quantum computer. PKC allows setting up a secure communication channel in an insecure network, such as the Internet. Therefore, it is an essential part of our economy and daily life. It consists of digital signature, encryption, and decryption algorithms to provide confidentiality and integrity in digital communication.

The scientist Shor developed quantum algorithms to solve the integer factorization and discrete logarithm problems in polynomial time. Typically deployed PKC cryptosystems, such as RSA and elliptic curve cryptography, rely on these two mathematical problems and are considered broken when a powerful quantum computer exists. Therefore, investigating cryptographic algorithms resistant to quantum attacks is essential to prevent a possible collapse of the global communication system. Post-quantum cryptography denotes the research area that includes such quantum-resistant cryptographic algorithms.

Numerous scientists have contributed to the development of various post-quantum cryptography algorithms. To further encourage the public research effort and prepare those algorithms for real-world applications, the National Institute of Standards and Technology (NIST) started a post-quantum standardization process in 2017. In addition to the mathematical security of the algorithm, efficient applicability is one of the most important criteria affecting the suitability for standardization. In particular, the algorithms must also be deployable in constrained devices, which are getting increasingly important due to the rapidly growing IoT market.

This work focuses on the efficient and secure implementation of post-quantum cryptography on embedded devices. In this context, hardware accelerators for mathematically complex and computationally intensive operations of post-quantum cryptography are investigated. While related works focused on standalone hardware implementations, this thesis presents hardware/software codesign solutions to combine the benefits of flexible software and efficient hardware solutions.

This work further analyzes processor coupling strategies for post-quantum hardware accelerators. The results show that loosely coupled coprocessors can efficiently accelerate modern lattice-based post-quantum cryptography. A tight coupling to the main processor can lead to further advantages. It allows sharing system resources between accelerators and the main processor, and it can avoid extensive bus communication. In order to fully exploit the potential of a tightly coupled solution, this thesis explores methods for reducing costly memory accesses and investigates tailored instruction set extensions to control the accelerators.

Abstract

The security of a system depends not only on the mathematical robustness of the chosen cryptographic algorithm but also on its implementation. This work proposes countermeasures for post-quantum cryptography accelerators in order to protect against implementation attacks, thus contributing to the development of fast and secure implementations.

In order to explore algorithmic improvements, this thesis investigates the influence of strong error-correcting codes for lattice-based cryptography, which are used to improve the security level, key/ciphertext sizes, and the intrinsic failure rate of the algorithm.

This thesis further analyzes the applicability of post-quantum cryptography for real-world applications. This work demonstrates that post-quantum cryptography can be efficiently implemented on a tiny non-commercial chip suitable for mass production. Moreover, the suitability of post-quantum cryptography for the automotive industry is analyzed. As a case study, it is shown that post-quantum cryptography can be deployed in automotive microcontrollers. Finally, an accelerator design for hybrid key encapsulations is proposed to foster a secure transition towards post-quantum cryptography.

Keywords— Post-quantum cryptography, lattice-based cryptography, embedded devices, hardware accelerators, accelerator coupling strategies, implementation attacks

Kurzfassung

Die Entwicklung von Quantencomputern hat in den letzten Jahren erhebliche Fortschritte erzielt. Große Quantencomputer können eine extreme Beschleunigung bei der Lösung von komplexen Aufgaben erreichen. Das führt allerdings zu einem erheblichen Problem für die derzeit eingesetzte Public-Key-Infrastruktur, da diese auf mathematischen Problemen beruht, welche mit einem Quantencomputer gelöst werden könnten. Public-Key-Kryptografie bildet die Grundlage für eine sichere Kommunikation über einen unsicheren Übertragungskanal, wie dem Internet, und ist somit ein essenzieller Bestandteil für unsere Wirtschaft und unser tägliches Leben. Die Public-Key-Kryptografie besteht aus digitalen Signatur-, Verschlüsselungs- und Entschlüsselungsverfahren, um Vertraulichkeit und Integrität in der digitalen Kommunikation zu erzielen.

Der Forscher Shor entwickelte Quantenalgorithmen, welche die Probleme der Integer-Faktorisierung und des diskreten Logarithmus in polynomialer Laufzeit auf einem Quantencomputer lösen können. Somit werden die üblicherweise eingesetzten Public-Key Kryptosysteme, wie RSA und Elliptische-Kurven-Kryptografie, gebrochen sein, wenn ein entsprechend leistungsstarker Quantencomputer entwickelt wurde. Um einen möglichen Kollaps des globalen Kommunikationssystems zu vermeiden, werden neuartige kryptografische Algorithmen entwickelt, bei denen davon auszugehen ist, dass sie resistent gegen Quantenalgorithmen und klassische Kryptoanalyse sind. Der Forschungsbereich, welcher sich mit diesen neuartigen Algorithmen beschäftigt, nennt sich Post-Quanten-Kryptografie.

Durch umfassende Forschungsanstrengungen wurden bereits mehrere, auf sehr unterschiedlichen Problemen beruhende, Post-Quanten-Kryptografie Algorithmen entwickelt. Das National Institute of Standards and Technology (NIST) startete im Jahr 2017 einen Standardisierungsprozess, um die Forschungsbemühungen zu fördern und um die Algorithmen praxistauglich zu machen. Neben der mathematischen Sicherheit dieser Verfahren spielt die Effizienz und Anwendbarkeit eine besonders große Rolle bei der Entscheidung, welcher Kandidat standardisiert werden sollte. Insbesondere müssen die Algorithmen auch für Geräte mit geringen Ressourcen geeignet sein, welche durch den rapide wachsenden IoT Markt immer wichtiger werden.

Diese Arbeit konzentriert sich auf die effiziente und sichere Implementierung von Post-Quanten-Kryptografie in eingebetteten Systemen. In diesem Zusammenhang werden insbesondere Hardwarebeschleuniger für die mathematisch komplexen und rechenintensiven Operationen erforscht. Während sich vorherige Arbeiten auf reine Hardwarelösungen konzentrieren, werden in dieser Arbeit Hardware/Software Codesign Lösungen präsentiert, welche die Vorteile einer flexiblen Softwarelösung und einer besonders effizienten Hardwarelösung verbinden.

Außerdem wird in dieser Arbeit analysiert, welchen Einfluss die Kopplungsart zwis-

chen Post-Quanten-Beschleuniger und dem Hauptprozessor hat. Die Ergebnisse zeigen, dass moderne gitterbasierte Post-Quanten-Kryptografie effizient mit einem lose gekoppelten Koprozessor beschleunigt werden kann. Eine enge Prozessorkopplung kann sogar zu noch besseren Ergebnissen führen, da Systemressourcen wiederverwendet und eine komplexe Buskommunikation vermieden werden können. Des Weiteren werden Befehlssatzerweiterungen zur Kontrolle der eng gekoppelten Hardwarebeschleuniger und Methoden zur Reduzierung der zeitintensiven Speicherzugriffe vorgestellt.

Die Sicherheit eines Systems hängt nicht nur von der mathematischen Robustheit des gewählten kryptografischen Algorithmus ab, sondern auch von der Implementierung. Um sich gegen Implementierungsattacken zu schützen, werden in dieser Arbeit Gegenmaßnahmen für Post-Quanten-Beschleuniger entwickelt. Somit wird ein wichtiger Beitrag zur Entwicklung von schnellen und sicheren Implementierungen dieser Algorithmen geleistet.

Auf algorithmischer Ebene wird zudem die Anwendung von starken Fehlerkorrekturverfahren für gitterbasierte Kryptografie analysiert, um das Sicherheitslevel, Schlüssel- und Geheimtextgrößen sowie die Fehlerwahrscheinlichkeit der Algorithmen zu optimieren.

Am Schluss dieser Arbeit wird die Eignung von Post-Quanten-Kryptografie für reale Anwendungsfälle analysiert. Es wird gezeigt, dass Post-Quanten-Kryptografie effizient auf einem winzigen, nicht-kommerziellen Chip implementiert werden kann, der für eine Massenproduktion geeignet ist. Darüber hinaus wird die Eignung von Post-Quanten-Kryptografie für die Automobilindustrie analysiert. Als Anwendungsfall wird gezeigt, dass Post-Quanten-Kryptografie für beliebte Mikrocontroller der Automobilbranche einsetzbar ist. Abschließend wird ein Beschleunigerdesign für hybride Schlüsselkapselungsverfahren entwickelt, um einen sicheren Übergang von herkömmlicher Kryptografie zu Post-Quanten-Kryptografie zu fördern.

Keywords— Post-Quanten-Kryptografie, gitterbasierte Kryptografie, eingebettete Geräte, Hardwarebeschleuniger, Strategien Prozessorkopplung, Implementierungsattacken

Acknowledgment

Many people supported me over the past few years. Without them, this major project—the dissertation—would probably not have been possible.

First, I would like to thank my thesis advisor Prof. Georg Sigl, who believed in my skills from the beginning and gave me the opportunity to pursue a doctorate. He has always been available for all kinds of questions and supported me directly whenever there were difficulties. I particularly had appreciated the helpful and solution-oriented advice, the realistic expectations, and the uncomplicated handling when something did not work as intended. There has always been a great working atmosphere at the chair, and I enjoyed the regular social events.

I would also like to especially thank Dr. Johanna Sepúlveda. Without her, I would probably never have discovered my interest in research and cryptography. She offered me a great master's thesis topic and greatly supported me in getting the research position at the chair. She has always supported me and guided me in the right direction. Thanks to her excellent connections and expertise, we have achieved many successes together. I learned that more things are possible than I initially thought through her encouragement.

I would also like to thank my colleagues at the chair. In particular, I would like to thank Christoph Frisch, Thomas Schamberger, Dr. Debapriya Basu Roy, and Patrick Karl for the joint research, the discussions, but also for the funny moments. It has been a pleasure working with you and with all other workmates not explicitly mentioned here. My thanks also go to Georg Maringer from the Institute for Communications Engineering for the research cooperation and the technical as well as private discussions. At this point, I would also like to thank Michiel Van Beirendonck from KU Leuven for the exciting and productive collaboration. Further, I would like to thank Dr. Thomas Pöppelmann. His expertise in the field of post-quantum cryptography is impressive. It has been a pleasure and an honor to work with him on joint research projects and articles.

I would also like to thank my students that I have supervised or co-supervised during the last years: Daniel Witsch, Fahd Kaatich, Jonas Vith, Tobias Peter, David Bongartz, Albert Brauer, Waleed Zaghloul, Paul Kohl, Abdelrahman Osman, Dinakar Raj Kariyappa Chikkamallaiah, Felix Oberhansl, Julia Mader, and Daniel Milincic.

Above all, I am grateful to my wife Sarah for always supporting me during the good but also the tough times. I would also like to thank my whole family and friends who have always encouraged me.

Contents

List of Figures

List of Tables

1 Introduction

This chapter discusses the exciting area of Post-Quantum Cryptography (PQC), related problems, and open research questions. Further, the contributions achieved in this work are summarized, and an overview of the thesis organization is provided.

1.1 Motivation

Public-Key Cryptography (PKC), or asymmetric cryptography, forms the basis for secure communication between different devices and is, therefore, a vital component of today's communication networks. The concept of PKC is based on the distinction between public and secret keys. Algorithms in the subcategory Public-Key Encryption (PKE) use the public key for the encryption of sensitive data and the secret key for the decryption of the corresponding ciphertext. Such algorithms can also be modified to establish a shared secret key on both sides of the communication channel, which can be used for further communication with fast symmetric encryption algorithms like Advanced Encryption Standard (AES). This type of PKE algorithm is also known as Key Encapsulation Mechanism (KEM). Digital signature algorithms are another subcategory of PKC. They use the secret key to create signatures and the public key for the verification. Thus, PKC is used to ensure confidentiality and integrity in digital communication.

Modern cryptography is based on the hardness of solving mathematically complex problems. Therefore, cryptosystems can only be secure if the underlying problem cannot be solved practically. In 1994, Shor developed a quantum algorithm [Sho94] that breaks—when using a sufficiently large quantum computer—the well-known and widely deployed PKC cryptosystems Rivest–Shamir–Adleman (RSA) and elliptic curve cryptography. His algorithm can compute the mathematically hard integer factorization problem (required for RSA) and the discrete logarithm problem (required for elliptic curve cryptography) in polynomial time. Grover discovered another powerful quantum algorithm, which accelerates the search in an unsorted database [Gro96]. His algorithm can be used to achieve a quadratic speedup of brute-force attacks on symmetric cryptography, e.g., on AES. In order to protect against Grover's algorithm, it is suggested to double the

key size [CJL+16]. In contrast to symmetric cryptography, PKC requires substantially different mathematical approaches because doubling key sizes is not sufficient in this case.

In 2019, Google claimed that a specific task was solved in 200 seconds on its 53-qubit quantum computer "Sycamore" while a conventional supercomputer is expected to take approximately 10 000 years for the same task [AAB+19b]. Although there have been doubts about whether a conventional computer would really take that long, this example shows that quantum computers exploiting quantum phenomena, such as superposition and entanglement, are already getting extremely powerful. Nevertheless, currently developed quantum computers are not yet powerful enough to break today's PKC. In [GE21], the authors estimated that 20 million noisy qubits might be needed to break 2 048-bit RSA. While still far away from such a powerful quantum computer, the amount of achieved physical qubits significantly increased in the last years. Moreover, ambitious goals for the following years were announced. For instance, IBM announced targeting a 1 000-qubit quantum computer by already the end of 2023 [Gam20]. It is relatively unclear how the development will further scale. However, it is expected that it could take only a few decades until a sufficiently powerful quantum computer is built to break PKC. Due to the enormous threat and long migration times, it is vital to already act now [Mos18]. In particular, products and devices with a long life cycle, such as automotive systems and airplanes, need to consider future quantum attacks. It also must be considered that encrypted data can be stored and broken later when stronger quantum computers are built. Cryptography resistant to attacks performed on quantum computers—also known as Post-Quantum Cryptography (PQC)—needs to be developed to ensure long-term secure communication channels between servers, clients, and multiple embedded devices.

1.2 Post-Quantum Cryptography Standardization

In 2015, the US National Security Agency (NSA) recognized the threat of quantum cryptanalysis and announced activities towards a transition to quantum-resistant cryptography [Nat16b]. The National Institute of Standards and Technology (NIST) has initiated a standardization process to drive the development of PQC algorithms and prepare them for real-world applications [Nat16a]. This process focuses on quantum-resistant schemes for PKE/KEM and digital signatures. In 2017, the public standardization process started with 69 potential candidates. In the second round of this standardization process, 25 candidates remained [AASA+19]. In 2020, seven algorithms were selected as finalists and eight as alternate candidates [AASA+20]. A complete list of all second-round candidates, the selected finalists, and alternate candidates is provided in Table 1.1. Finalists are considered to be standardized by NIST soon after the next selection round. Alternate candidates require more analysis or might become more interesting when new security issues at some finalists appear.

The NIST PQC submissions can be divided into five PQC classes: lattice-based cryptography, code-based cryptography, multivariate cryptography, hash-based cryptography,

and isogeny-based cryptography [CJL+16].

Lattice-based cryptography. This category contains cryptographic algorithms with security proofs based on hard lattice problems. Lattice problems are already well understood and are considered to be resistant against all known quantum computer attacks. Moreover, lattice-based cryptography is characterized by a high performance with moderate key and ciphertext sizes. These characteristics are the main reasons why this class is the largest one within the NIST standardization competition. This thesis will have, therefore, also a particular focus on lattice-based cryptography.

Code-based cryptography. This category is based on error-correcting codes and the fact that the general decoding problem is $\mathcal{NP}$-complete [BMVT78]. The most popular code-based NIST candidate is Classic McEliece. This candidate has already been well studied and has one of the smallest ciphertext sizes [AASA+20]. Nevertheless, its large public key size prevents the application of this scheme in most embedded devices. Structured code-based schemes, such as BIKE, and other new code-based approaches, such as HQC, reduced the public key size but are based on less studied problems.

Multivariate cryptography. This category is based on the hardness of solving non-linear multivariate equations of polynomials over a finite field, which is $\mathcal{NP}$-complete [DY09]. Signature schemes of this class, such as Rainbow and GeMSS, frequently suffer from large public keys and are therefore less suitable for small devices [AASA+20].

Hash-based cryptography. The security proofs of this category are based on the underlying hash function. The NIST alternate candidate SPHINCS+ is considerably slower and has larger signature sizes compared to its competitors. The scheme Picnic has a very high memory consumption and is unsuitable for small embedded devices [KRSS19]. The advantage of hash-based cryptography is the high trust in its security proofs.

Isogeny-based cryptography. This category is based on the complexity of finding isogenies between supersingular elliptic curves [JDF11]. The only NIST submission of this category, SIKE, has extremely small key and ciphertext sizes but a comparably low performance [AASA+20].

1.3 Problem Definitions

Efficient realizations and implementations of the NIST PQC candidates play a major role in increasing the confidence in the practicability of these schemes [AASA+20]. PQC introduces new mathematical elements that are computationally intensive. In particular, constrained devices with limited computing capabilities and resources usually require hardware acceleration to meet the performance and energy requirements. The design of efficient low-energy solutions gets increasingly important for smart cards, Trusted Platform Module (TPM) devices, and other smart devices related to the rising Internet

Table 1.1: Post-quantum candidates of NIST PQC competition.

Algorithm	Category	Status	Class
Kyber	PKE/KEM	finalist	lattice-based
NTRU	PKE/KEM	finalist	lattice-based
Saber	PKE/KEM	finalist	lattice-based
Classic McEliece	PKE/KEM	finalist	code-based
FrodoKEM	PKE/KEM	alternate	lattice-based
NTRU Prime	PKE/KEM	alternate	lattice-based
BIKE	PKE/KEM	alternate	code-based
HQC	PKE/KEM	alternate	code-based
SIKE	PKE/KEM	alternate	isogeny-based
LAC	PKE/KEM	2nd round	lattice-based
NewHope	PKE/KEM	2nd round	lattice-based
Round5	PKE/KEM	2nd round	lattice-based
ThreeBears	PKE/KEM	2nd round	lattice-based
LEDAcrypt	PKE/KEM	2nd round	code-based
ROLLO	PKE/KEM	2nd round	code-based
RQC	PKE/KEM	2nd round	code-based
Dilithium	signature	finalist	lattice-based
FALCON	signature	finalist	lattice-based
Rainbow	signature	finalist	multivariate-based
GeMSS	signature	alternate	multivariate-based
Picnic	signature	alternate	hash and block ciphers
SPHINCS+	signature	alternate	hash-based
qTESLA	signature	2nd round	lattice-based
LUOV	signature	2nd round	multivariate-based
MQDSS	signature	2nd round	multivariate-based

of Things (IoT) market. While hardware implementations of cryptographic algorithms can achieve a high performance and low energy consumption, they are inflexible and are difficult to update. Software implementations provide a high flexibility but have a lower performance. Achieving high flexibility, high performance, and low resource consumption is a challenging task requiring new design strategies and inevitable trade-offs.

In addition to these implementation goals, the secure realization of the cryptographic algorithm is of high relevance. Although mathematically secure, cryptographic implementations can still be broken using Side-Channel Attacks (SCA). This is a major problem of applied cryptography. At SCA, attackers measure and exploit physical characteristics (e.g., timing, power, or electromagnetic signals) of a cryptographic operation to retrieve information about secret data [Koc96]. Effective countermeasures for PQC against SCA and the corresponding cost analysis are essential points that are still largely unexplored.

Several open problems exist regarding implementation aspects but also regarding algorithmic aspects. The cryptanalysis and certain parameter settings are still research targets as most PQC algorithms are relatively new. In particular, lattice-based cryptography can use error-correcting codes to improve specific characteristics, such as the security level. However, a strong error-correcting code increases the complexity and introduces new problems. Therefore, an important step is an analysis of the benefits and drawbacks of using strong error-correcting codes.

Finally, the migration from traditional cryptography to PQC and the deployment of PQC in the applications are challenging tasks that will probably take many years.

1.4 Research Objectives, Contributions, and Thesis Outline

The following points summarize the research objectives, contributions, and publications related to this thesis:

(I) Design of efficient hardware architectures and coprocessors for PQC
(related publications: [BFM+18, FSF+19, FS19, FSM+19]);

(II) Investigation of accelerator integration strategies and instruction set extensions for PQC
(related publications: [FSS20a, FSS20b, RFS20, KFS22]);

(III) Side-channel protection mechanisms for PQC
(related publications: [FVBBR+21]);

(IV) Analysis of error-correcting codes for lattice-based cryptography
(related publications: [FPS18, FVS20, MFS20]);

(V) Integration of PQC into applications and migration using hybrid key encapsulation
(related publications: [FVS19, FVFS21, OFP+22]).

Part I. This part presents design methodologies as well as hardware and system architectures for lattice-based PQC. Several design solutions for the performance bottlenecks of PQC are investigated, including the generation of ring elements and polynomial ring arithmetic. The first hardware/software codesign solutions for lattice-based protocols are presented for different target devices to increase the flexibility and overcome the drawbacks of pure hardware designs. The developed coprocessors are coupled to the main processor through bus interfaces.

Part II. This part analyzes small and highly flexible hardware/software codesigns. In contrast to the coprocessors developed in Part I, the PQC accelerators are directly integrated into the system's processor. The control and access of the tightly coupled accelerators are realized using instruction set extensions. Design strategies to reuse existing system resources and reduce costly memory accesses are explored.

Part III. This part investigates how the presented hardware accelerators can be protected against SCA. It presents the first masked hardware accelerators with instruction set extensions for two lattice-based NIST PQC finalists and measures towards secure system design. The accelerators ensure a controlled execution and boost the performance of hardened PQC implementations. This part provides a detailed analysis of implementation costs for the hardened designs and helps to assess the advantages and disadvantages of two closely related NIST finalists.

Part IV. This part analyzes the influence of strong error-correcting codes for lattice-based cryptography. Lattice-based protocols have an intrinsic failure rate that depends on the parameter set. The analysis has shown that the security level, key/ciphertext sizes, and failure rate can be optimized with a strong error correction. This work further contributes to the failure rate analysis for lattice-based cryptography with error correction, which is not straightforward due to stochastic dependence between decryption failures.

Part V. This part shows the applicability of PQC for safety-critical applications with long-term security requirements. As a case study, the automotive industry is considered. First performance results of lattice-based PQC schemes are provided for the automotive industry. Moreover, this part presents a unified hardware accelerator that supports traditional cryptography and PQC to foster the migration to PQC using hybrid key encapsulations. Finally, a post-quantum cryptography chip is presented that was developed in the context of this thesis.

Thesis organization. The structure of this thesis is mainly ordered according to the contributions of Parts I–V. Chapter 2 introduces lattice-based cryptography, discusses its performance bottlenecks, and summarizes hardware accelerator types. Chapter 3 investigates the design and performance of loosely coupled post-quantum coprocessors.

As use cases, hardware/software codesigns of two popular lattice-based schemes are proposed. Chapter 4 investigates tightly coupled accelerators to improve flexibility, reusability, and other important design characteristics for five different post-quantum schemes. In Chapter 5, a generic arithmetic accelerator for lattice-based cryptography is presented. Moreover, side-channel countermeasures for two PQC NIST finalists are presented and evaluated. Chapter 6 investigates the integration of powerful error-correcting codes for lattice-based cryptography. The integration of PQC in real-world applications and related challenges are discussed in Chapter 7. The thesis ends with a conclusion in Chapter 8.

2 Preliminaries

This chapter provides the theoretical background for this work. At the beginning of this chapter, lattice-based cryptography and the popular NTRU and LWE cryptosystems are introduced. Further, a discussion about performance bottlenecks of lattice-based cryptography and theoretical optimizations for the polynomial ring arithmetic and polynomial sampling is provided. The chapter ends with an overview of PQC implementation strategies and hardware accelerator types. The description of NTRU is partly based on the author's publication [FSF+19]. For the description of the LWE cryptosystems, the polynomial ring arithmetic, the polynomial sampling, and the hardware accelerator types, parts of the author's publications [FSS20b, FVBBR+21, FVFS21] are used.

2.1 Mathematical Background and Notation

The following paragraphs provide the mathematical background of PQC and introduce the notation used in this thesis.

Fields. A field is a set of elements where the conventional mathematical operations (additions, subtractions, multiplications, and divisions) are defined. Further, properties like associativity, commutativity, and distributivity are valid in a field. Well-known fields are the rational numbers $\mathbb{Q}$, the real numbers $\mathbb{R}$, and the complex numbers $\mathbb{C}$ [Koc08].

Finite fields. Let $\mathbb{Z}$ be the set of integers and let $\mathbb{F}_q = \mathbb{Z}_q$ be the set of integers modular q. The set $\mathbb{F}_q$ is a typical example of a finite field. It has a finite number of in total q elements. All mathematical field operations can be applied without leaving the finite set. $\mathbb{F}_{q^k}$ with $k > 1$ is an extension field of degree k, where each element is a polynomial of degree $k-1$. Depending on the context, the finite field or the extension field is referred to as Galois field $\mathrm{GF}(q^k)$.

Polynomial rings. Rings have similar properties as fields. However, the commutativity of multiplications and the existence of inverse elements are no conditions for a ring. Nevertheless, commutativity of multiplications might be applicable, and inverse elements might exist in a ring. Let $\mathcal{R} = \mathbb{Z}_q/\langle\phi(x)\rangle$ be a ring with the cyclotomic polynomial $\phi(x)$. The ring elements of $\mathcal{R}$ are polynomials of degree $n-1$ with integer coefficients in $[0, q)$. These elements can be represented as $a = a_0 + a_1 x + a_2 x^2 + \cdots + a_{n-1}x^{n-1} = \sum_{i=0}^{n-1} a_i x^i$. The cyclotomic polynomial $\phi(x)$ is frequently set to $\phi(x) = x^n - 1$ or $\phi(x) = x^n + 1$. The coefficients are reduced modular q and the whole polynomial modular $\phi(x)$ during the ring arithmetic.

Polynomial additions. Let $a, b \in \mathcal{R}$ and let c be the sum of these two polynomials. Then, the polynomial addition is defined by

$$c = a + b = \sum_{i=0}^{n-1} (a_i + b_i \mod q) x^i \ . \tag{2.1}$$

The reduction by the cyclotomic polynomial does not affect polynomial additions or subtractions.

Polynomial multiplications. Let $a, b \in \mathcal{R}$ and let c be the product of these two polynomials. Then, the polynomial multiplication is defined by

$$c = a \cdot b \mod \phi(x) = \Big(\sum_{i=0}^{2n-2} \sum_{j=0}^{i} (a_j \cdot b_{i-j} \mod q) x^i\Big) \mod \phi(x) \ . \tag{2.2}$$

In this equation, the coefficients a_l and b_l are zero for $l \geq n$.

Polynomial inversions. The inverse polynomial a^{-1} must fulfill $a \cdot a^{-1} = \tilde{a}$ with $\tilde{a}_0 = 1$ and $\tilde{a}_i = 0$ for $\forall i \neq 0$. Such an inverse element does not necessarily exist for each element $a \in \mathcal{R}$.

Distributions. Lattice-based cryptography requires different probability distributions. Let $\mathcal{U}_q$ be a uniform distribution with outcomes in the set $[0, q)$. Let Ψ_η be a centered binomial distribution with standard deviation $\sqrt{\eta/2}$ and outcomes in the set $[-\eta, \eta]$. Usually, the outcomes of the binomial distribution are much smaller than the outcomes of the uniform distribution, i.e., $\eta \ll q$.

Sampling. The sampling process is defined by $x \xleftarrow{\$} S$, which randomly samples the value x from the set S. Each coefficient is sampled separately from the target distribution to sample a complete polynomial. The polynomial sampling from a uniform distribution is written as $a \xleftarrow{\$} \mathcal{U}_q$. Similarly, the sampling from a binomial distribution is written as $a \xleftarrow{\$} \Psi_\eta$. Some PQC schemes require the sampling of a ternary polynomial. In this thesis, the notation $a \xleftarrow{\$} \mathcal{T}_d$ is used for the sampling of a polynomial with coefficients in the set $\{-1, 0, 1\}$, where d coefficients are equal to 1 and -1, respectively. The notation $x \xleftarrow{seed} S$ is used to indicate a deterministic sampling process, i.e., repeating the sampling with the same *seed* will always return the same result.

General notation. This thesis uses the letters q or p for the modulus. The modulus is either a prime or a power-of-two integer for all considered schemes. Moreover, bold letters are mainly used for vectors and bold capital letters for matrices.

2.2 Lattice-Based Cryptography

Mathematical hard problems are the basis of modern cryptography. The hardness assumption ensures the security of the cryptographic primitives. In 1996, the pioneering work of Ajtai in [Ajt96] introduced the first cryptographic one-way function based on the hardness of lattice problems. This work has been the start of lattice-based cryptography. But the breakthrough of lattice-based cryptography came two years later when Jeffrey Hoffstein, Jill Pipher, and Joseph Silverman proposed the efficient and practical public-key encryption scheme Nth Degree Truncated Polynomial Ring Unit (NTRU) [HPS98]. In 2009, Regev proposed the Learning With Errors (LWE) problem together with the first security proofs showing that LWE is as hard as worst-case lattice problems [Reg09]. NTRU and LWE are the basis for most modern lattice-based cryptographic primitives.

2.2.1 NTRU

NTRU was standardized with modifications in the IEEE standard P1363.1 and the financial services industry standard X9.98 as an efficient alternative to RSA and elliptic curve cryptography. In 2017, the three NTRU variants NTRUEncrypt, NTRU-HRSS-KEM, and NTRUPrime were submitted to the NIST PQC standardization process. NIST recently selected the merged variant of NTRUEncrypt and NTRU-HRSS-KEM as a finalist and NTRUPrime as an alternate candidate [AASA+20].

2.2.2 Public-Key Encryption for NTRU-Based Schemes

PKE/KEM schemes are composed of the three operations key generation (KeyGen), encryption (Encrypt), and decryption (Decrypt). These operations are provided for NTRU in a generic and simplified form of the IEEE P1363.1 standard in Algorithms 1–3. The main principles are similar for all other NTRU versions. The algorithms omit details that are not important for understanding this work. A comprehensive description can be

found in the original specification [L+01]. The following paragraphs describe the main parameters of NTRU, necessary subroutines, and the NTRU protocol (Algorithms 1–3).

Parameters and rings. NTRU can be mainly defined using the integer parameters n, q, p, d_F, d_g, and d_r. The polynomial length n and the sampling parameters d_F, d_g, and d_r depend on the chosen NTRU parameter set. The primes are fixed to $q = 2048$ and $p = 3$ for the standardized parameter sets. All computations are performed in the rings $\mathcal{R}_q = \mathbb{Z}_q/\langle\phi(x)\rangle$ or $\mathcal{R}_p = \mathbb{Z}_p/\langle\phi(x)\rangle$ with $\phi(x) = x^n - 1$.

Polynomial / byte stream conversion. The subroutine `poly2byte` is used to transform a polynomial of $\mathcal{R}_q$ into its byte representation. The function `m2poly` turns the byte sequence of a message into a ternary polynomial. It splits the byte sequence into 3-bit chunks and maps them into two ternary coefficients, respectively. The reverse operation of `m2poly` is `poly2m`.

Blinding Polynomial Generation Method (BPGM). This deterministic function computes an ephemeral blinding polynomial r. It is based on a cryptographic hash function $\mathbf{G}$ (e.g., SHA256) and takes as input the byte string $seed\_bpgm$. This can be written as

$$r = \texttt{BPGM}(seed\_bpgm) \ . \tag{2.3}$$

Mask Generation Function (MGF). This function is very similar to `BPGM` and is also based on a hash function $\mathbf{G}$. However, the internal operations are slightly different. It takes as input the byte string $seed\_mgf$ and creates a full mask m_{mask} for the message. This can be written as

$$m_{mask} = \texttt{MFG}(seed\_mgf) \ . \tag{2.4}$$

NTRU.KeyGen. It generates the public key pk and the secret key sk. The algorithm first samples two random ternary polynomials $F, g \in \mathcal{R}_p$ from the ternary distributions $\mathcal{T}_{d_F}$ and $\mathcal{T}_{d_g}$. As $p = 3$, the coefficients of both polynomials can be represented as integers in the set $\{-1, 0, 1\}$. The number of coefficients equal to 1 and -1 is determined by the parameter d_F for F and d_g for g. Not all polynomials are invertible in $\mathcal{R}_q$. Therefore, it is checked if $f = 1 + p \cdot F$ and g are invertible. If they are not, the sampling is repeated. The public key $h = (f^{-1} \cdot g) \cdot p$ can be computed after the sampling. The computations involve one polynomial multiplication and one consecutive scalar multiplication with the modulus p. The secret key is defined by $f = 1 + p \cdot F$, where only a scalar multiplication and addition with 1 is required.

NTRU.Encrypt. It transforms a message m into a ciphertext ct. The algorithm first generates the byte representation of the truncated public polynomial h_{trunc} using the `poly2byte` conversion. The result is then concatenated with an object identifier (`OID`), the plaintext message m, and a random number b. This concatenation is taken to generate the blinding polynomial r using `BPGM`. The resulting polynomial r is multiplied with h.

Algorithm 1: NTRU.KeyGen

1 $F \xleftarrow{\$} \mathcal{T}_{d_F}$ // $F \in \mathcal{R}_p$
2 $f \leftarrow 1 + p \cdot F$ // $f \in \mathcal{R}_q$
3 $f^{-1} \mod q$ (goto 1 if not invertible)
4 $g \xleftarrow{\$} \mathcal{T}_{d_g}$ // $g \in \mathcal{R}_p$
5 $g^{-1} \mod q$ (goto 4 if not invertible)
6 $h \leftarrow (f^{-1} \cdot g) \cdot p$ // $h \in \mathcal{R}_q$

Result: $pk = h$, $sk = f$

Algorithm 2: NTRU.Encrypt

Input: $pk = h$, $m \in \{0, \ldots, 255\}^{len_m}$, $\texttt{OID} \in \{0, \ldots, 255\}^3$, $b \xleftarrow{\$} \mathcal{U}_{256}^{len_b}$

1 $seed_1 \leftarrow \{\texttt{OID} \,||\, m \,||\, b \,||\, \texttt{poly2byte}(h_{trunc})\}$
2 $r \leftarrow \texttt{BPGM}(seed_1)$ // $r \in \mathcal{R}_p$
3 $seed_2 \leftarrow \texttt{poly2byte}(r \cdot h)$ // $r \cdot h \in \mathcal{R}_q$
4 $m_{mask} \leftarrow \texttt{MFG}(seed_2)$ // $m_{mask} \in \mathcal{R}_p$
5 $m_{pad} \leftarrow \{b \,||\, len_m \,||\, m \,||\, 00 \ldots\}$
6 $m_{ter} \leftarrow \texttt{m2poly}(m_{pad})$ // $m_{ter} \in \mathcal{R}_p$
7 $m' \leftarrow m_{ter} + m_{mask} \mod p$ // $m' \in \mathcal{R}_p$
8 $e \leftarrow m' + r \cdot h$ // $e \in \mathcal{R}_q$

Result: $ct = e$

Algorithm 3: NTRU.Decrypt

Input: $ct = e$, $sk = f$, $pk = h$, $\texttt{OID} \in \{0, \ldots, 255\}^3$

1 $m' \leftarrow f \cdot e \mod p$ // $m' \in \mathcal{R}_p$
2 $r \cdot h \leftarrow e - m'$ // $r \cdot h \in \mathcal{R}_q$
3 $seed_1 \leftarrow \texttt{poly2byte}(r \cdot h)$
4 $m_{mask} \leftarrow \texttt{MGF}(seed_1)$ // $m_{mask} \in \mathcal{R}_p$
5 $m_{ter} \leftarrow m' - m_{mask} \mod p$ // $m_{ter} \in \mathcal{R}_p$
6 $\{b \,||\, len_m \,||\, m \,||\, 00 \ldots\} \leftarrow poly2m(m_{ter})$
7 $seed_2 \leftarrow \{\texttt{OID} \,||\, m \,||\, b \,||\, \texttt{poly2byte}(h_{trunc})\}$
8 $r_{calc} = \texttt{BPGM}(seed_2)$ // $r_{calc} \in \mathcal{R}_p$
9 **if** $r \cdot h \neq r_{calc} \cdot h$ **then**
10 $\quad m \leftarrow \texttt{error}$
11 **end**

Result: m

This product is converted into a byte stream, which is used as input for the MGF operation to generate m_{mask}. Then, a concatenation of b, the parameter len_m, and the zero padded input message is built. This concatenation is transformed into a polynomial and masked with m_{mask} to obtain m'. The masking operation corresponds to a polynomial addition with consecutive reduction by the modulus p. The ciphertext $ct = e$ is the addition of the masked message m' and $r \cdot h$.

NTRU.Decrypt. It retrieves the plaintext message m from the ciphertext $ct = e$. The algorithm uses the secret key f and the ciphertext e to compute $r \cdot h$. With the knowledge of $r \cdot h$, the mask m_{mask} can be retrieved by means of MGF. This mask can be used to unmask $m' = f \cdot e \mod p$ and obtain the predicted plaintext message m. Reapplying BPGM with m, thus, repeating parts of the encryption to obtain r_{calc}, can expose modifications of the ciphertext. This allows detecting and preventing Chosen-Ciphertext Attacks (CCA). If the recomputed product $r_{calc} \cdot h$ is equal to $r \cdot h$, the algorithm outputs the message m. Otherwise, an error message is assigned to the output.

2.2.3 Learning With Errors (LWE)

Several variants of the plain LWE problem have been published within the last years to achieve performance improvements. In the following paragraphs, the algebraically structured variants Ring Learning With Errors (RLWE), Module Learning With Errors (MLWE), Integer Module Learning With Errors (IMLWE), and Module Learning With Rounding (MLWR) are outlined.

RLWE. It was introduced in [LPR10] as an LWE variant to reduce key and ciphertext sizes and to realize efficient arithmetic.

Definition 1 (RLWE instance of [LPR10])**.** *The RLWE instance is defined by* $(a, b = a \cdot s + e)$ *with all elements in* $\mathcal{R}_q$*, the uniformly distributed public polynomial* $a \xleftarrow{\$} \mathcal{U}_q$*, the Gaussian/binomially distributed secret* $s \xleftarrow{\$} \Psi_{\eta_1}$*, and the Gaussian/binomially distributed error* $e \xleftarrow{\$} \Psi_{\eta_2}$*.*

MLWE. Instead of single large ring elements, it uses matrices and vectors of a smaller ring. This leads to higher flexibility as the arithmetic can be optimized for a single ring, and security level modifications are simply achievable with an adaptation of the matrix and vector dimensions.

Definition 2 (MLWE instance of [BGV14, LS15])**.** *The MLWE instance is defined by* $(\boldsymbol{A}, \boldsymbol{b} = \boldsymbol{A} \cdot \boldsymbol{s} + \boldsymbol{e})$ *with the uniformly distributed public matrix* $\boldsymbol{A} \in \mathcal{R}_q^{k_1 \times k_2}$*, the Gaussian/binomially distributed secret* $\boldsymbol{s} \in \mathcal{R}_q^{k_2}$*, and the Gaussian/binomially distributed error* $\boldsymbol{e} \in \mathcal{R}_q^{k_1}$*.*

MLWR. It is a combination of Learning With Rounding (LWR) proposed in [BPR12] and MLWE. LWR or MLWR replace the error term with noise generated from a deterministic rounding function. This rounding function introduces the noise required for the security proofs and automatically reduces the key and ciphertext sizes.

Definition 3 (MLWR instance of [DKRV18])**.** *The MLWR instance is defined by* $(\boldsymbol{A}, \boldsymbol{b} = \lfloor \frac{p}{q}(\boldsymbol{A} \cdot \boldsymbol{s}) \rceil)$ *with the uniformly distributed public matrix* $\boldsymbol{A} \in \mathcal{R}_q^{k_1 \times k_2}$*, the Gaussian/binomially distributed secret* $\boldsymbol{s} \in \mathcal{R}_q^{k_2}$*, and the integers* p, q *with* $q > p$*. The rounding function scales the inner product by* p/q *and rounds the result to the closest integer modular* p*.*

IMLWE. It is the integer version of MLWE. The general idea proposed in [Gu19] is to evaluate the polynomials in RLWE/MLWE schemes for a specific value of x. Thus, efficient and well-studied big-integer arithmetic libraries can be used, especially when using Mersenne primes for the evaluation.

Definition 4 (IMLWE instance of [Ham19])**.** *Let* $x = q$ *be an integer and* $\phi(x) = N$ *a prime number to avoid subrings. The integer ring is defined by* $\mathcal{R}_N = \mathbb{Z}_N$*. The IMLWE instance is similar to the MLWE instance but consists of computations in* $\mathcal{R}_N$*.*

If all elements are sampled randomly, it is conjectured to be mathematically hard to distinguish the LWE-based sample pair (a, b) or $(\boldsymbol{A}, \boldsymbol{b})$ from uniformly distributed noise (decision problem) and to recover the secret element from the instance b or $\boldsymbol{b}$ (search problem). Due to its average-case hardness assumption, LWE is well suited for creating cryptographic primitives. Prominent algorithms based on RLWE, MLWE, MLWR, and IMLWE are the NIST PQC schemes NewHope [AAB+19a], Kyber [ABD+20], Saber [BMD+20], and ThreeBears [Ham19], respectively.

2.2.4 Public-Key Encryption for LWE-Based Schemes

LWE-based PKE/KEM schemes are also composed of the three operations key generation (KeyGen), encryption (Encrypt), and decryption (Decrypt). These operations are shown in a generic and simplified form in Algorithms 4–6. The notation of these algorithms is partly based on [ABD+20]. Note that for RLWE schemes, the parameter k is set to one and that all error terms for LWR schemes are set to zero (instead, the rounding operation is used). More detailed descriptions can be found in [AAB+19a, ABD+20, BMD+20, Ham19]. The following paragraphs first describe important subroutines required for LWE-based cryptography and then explain the general protocol structure.

Compression/decompression. The function $\texttt{compress}_q(a, d)$ is used to compress the polynomial a (or vector $\boldsymbol{a}$) such that all coefficients are in the set $[0, 2^d)$. The polynomials can be compressed before transmitting them over the channel to decrease the bandwidth, i.e., the key and ciphertext sizes. The compression corresponds to a modulus switching between the prime q and 2^d, where $q > 2^d$. This is similar to the removal of the lower-order bits of the coefficients. These bits have low information content. The $\texttt{decompress}_q(\overline{a}, d)$ operation is the reverse operation required after the transmission to lift the polynomial to its original size. More formally, the two operations are defined by $\texttt{compress}_q(a, d) = \lceil (2^d/q) \cdot a \rfloor \mod 2^d$ and $\texttt{decompress}_q(\overline{a}, d) = \lceil (q/2^d) \cdot \overline{a} \rfloor \mod q$ [BDK+18].

Encoding/decoding. The functions `encode` and `decode` are used to switch between bit streams and polynomials (or vectors of polynomials). The `encode` operation transforms a bit stream, e.g., the message with a length of 256-bit, into a polynomial. Each bit of the bit stream can be mapped, e.g., to one coefficient until all bits are encoded. For instance, if a message bit is 1 the respective coefficient is mapped to $\lceil q/2 \rfloor$, otherwise it is mapped to 0. Let m be the message bit stream and let v be the resulting polynomial of the encoding. Then, the encoding is the same as $\texttt{decompress}_q(m, 1)$ and the decoding the same as $\texttt{compress}_q(v, 1)$. Some schemes also apply an error correction mechanism and map one bit to multiple coefficients. This increases the robustness of the scheme and allows a more advanced message decoding. However, when error correction is used, the decompression and compression operations cannot be applied for the message encoding and decoding.

LWE.KeyGen. It generates the public key $pk = (\overline{\boldsymbol{b}}, seed_1)$ and the secret key $sk = \boldsymbol{s}$. The algorithm first computes the uniformly distributed public element $\boldsymbol{A}$ using a random seed and a Pseudo Random Number Generator (PRNG). Typically, the hash primitives SHAKE-128/SHAKE-256 or the symmetric cipher AES are used as PRNG. In the next step, the random secret key $\boldsymbol{s}$ and the error $\boldsymbol{e}$ are sampled from the noise distribution Ψ_η^k. Finally, the LWE-based instance $\boldsymbol{b} = \boldsymbol{A}\boldsymbol{s} + \boldsymbol{e}$ is generated and optionally compressed using the $\texttt{compress}_q$ function. The secret key is set to $sk = \boldsymbol{s}$ and the public key to $pk = (\overline{\boldsymbol{b}}, seed_1)$. The public key securely hides the secret key in the LWE instance $\overline{\boldsymbol{b}}$, which can be transmitted over an insecure channel. If the parameter set is carefully chosen, it is impossible for an attacker to retrieve $\boldsymbol{s}$ from $\overline{\boldsymbol{b}}$ or to distinguish $(\boldsymbol{A}, \overline{\boldsymbol{b}})$ from a uniform sample pair. To reduce the key size, not the whole public element $\boldsymbol{A}$ is assigned to the public key but only the seed that is used to generate it.

LWE.Encrypt. It generates the ciphertext $ct = (\overline{\boldsymbol{u}}, \overline{v})$ of a given input message m. This algorithm first computes $\boldsymbol{A}$ using $seed_1$ from $pk = (\overline{\boldsymbol{b}}, seed_1)$. As a deterministic sampling operation is used, Encrypt has the same public element $\boldsymbol{A}$ as KeyGen. The algorithm further samples the secret element $\boldsymbol{s'}$ and the error elements $\boldsymbol{e'}$, e''. These terms are then used to create two LWE-based instances $\boldsymbol{u}$ and v, whereas v contains the secret input message m. At the message encoding, the byte string of m is transformed to a polynomial, and an error correction encoding can be applied. Again, optionally compression can be utilized to reduce the ciphertext size. The ciphertext is composed of $\overline{\boldsymbol{u}}$ and $\overline{v}$. These LWE instances securely hide the secret element $\boldsymbol{s'}$ and the secret input message m.

LWE.Decrypt. It computes the predicted plaintext m' from the ciphertext $ct = (\overline{\boldsymbol{u}}, \overline{v})$ using the secret key $sk = \boldsymbol{s}$. This function requires only a few decompression and arithmetic operations. The decompressed LWE instance v' contains the encoded secret input message $\texttt{encode}(m)$ but also the term $\boldsymbol{b'}^T\boldsymbol{s'} + e''$. To partly remove the latter term from v', the product $\boldsymbol{s}^T\boldsymbol{u'}$ is subtracted from v'. This removes the largest terms such that, after the message decoding, the probability is high that $m' \equiv m$. An error correction decoder can be used at the `decode` operation to increase the success rate of the decryption. See Chapter 6 for a more detailed analysis.

2.3 Polynomial Ring Arithmetic

The ring arithmetic described in the previous section is one of the major performance bottlenecks of structured lattice-based cryptography. In particular, the polynomial multiplication is frequently the optimization target for performance improvements.

2.3.1 Schoolbook Multiplication

The polynomial ring additions and subtractions of structured lattice-based cryptography can be performed coefficient-wise with a complexity of $\mathcal{O}(n)$ (as shown in Section 2.1,

Algorithm 4: LWE.KeyGen

$seed_1 \xleftarrow{\$} \mathcal{U}_2^{256}$
$\boldsymbol{A} \xleftarrow{seed_1} \mathcal{U}_q^{k \times k}$
$\boldsymbol{s}, \boldsymbol{e} \xleftarrow{\$} \Psi_\eta^k \times \Psi_\eta^k$
$\boldsymbol{b} \leftarrow \boldsymbol{A}\boldsymbol{s} + \boldsymbol{e}$
$\overline{\boldsymbol{b}} \leftarrow \texttt{compress}_q(\boldsymbol{b}, d_b)$
Result: $pk = (\overline{\boldsymbol{b}}, seed_1)$, $sk = \boldsymbol{s}$

Algorithm 5: LWE.Encrypt

Input: $pk = (\overline{\boldsymbol{b}}, seed_1)$, $m \in \{0,1\}^{256}$
$\boldsymbol{A}^T \xleftarrow{seed_1} \mathcal{U}_q^{k \times k}$
$\boldsymbol{s'}, \boldsymbol{e'}, e'' \xleftarrow{\$} \Psi_\eta^k \times \Psi_\eta^k \times \Psi_\eta$
$\boldsymbol{b'} \leftarrow \texttt{decompress}_q(\overline{\boldsymbol{b}}, d_b)$
$\boldsymbol{u} \leftarrow \boldsymbol{A}^T\boldsymbol{s'} + \boldsymbol{e'}$
$v \leftarrow \boldsymbol{b'}^T\boldsymbol{s'} + e'' + \texttt{encode}(m)$
$\overline{\boldsymbol{u}} \leftarrow \texttt{compress}_q(\boldsymbol{u}, d_u)$
$\overline{v} \leftarrow \texttt{compress}_q(v, d_v)$
Result: $ct = (\overline{\boldsymbol{u}}, \overline{v})$

Algorithm 6: LWE.Decrypt

Input: $ct = (\overline{\boldsymbol{u}}, \overline{v})$, $sk = \boldsymbol{s}$
$\boldsymbol{u'} \leftarrow \texttt{decompress}_q(\overline{\boldsymbol{u}}, d_u)$
$v' \leftarrow \texttt{decompress}_q(\overline{v}, d_v)$
$m' \leftarrow \texttt{decode}(v' - \boldsymbol{s}^T\boldsymbol{u'})$
Result: m'

Equation 2.1). Polynomial ring multiplications are significantly more expensive. The schoolbook method—the most straightforward approach—has a complexity of $\mathcal{O}(n^2)$. In lattice-based cryptography, the product of a polynomial multiplication of length $2n$ is typically reduced by the cyclotomic polynomial $\phi(x)$ (frequently $x^n - 1$ or $x^n + 1$) to a polynomial of length n as shown in Section 2.1, Equation 2.2. The reduction modular $\phi(x)$ in this equation can be simplified as illustrated in the following paragraph.

Schoolbook multiplication modular $\phi(x)$. Let $a, b \in \mathbb{Z}_q/\langle\phi(x)\rangle$ be two ring polynomials and $c \in \mathbb{Z}_q/\langle\phi(x)\rangle$ the corresponding product polynomial. Then, the polynomial ring multiplication with integrated modular reduction by $\phi(x)$ is defined by

$$c_i = (\sum_{j=0}^{i} a_j \cdot b_{(i-j)} \mod q \pm \sum_{j=i+1}^{n-1} a_j \cdot b_{(n+i-j)} \mod q) \mod q \ , \tag{2.5}$$

where the latter part is added for positive wrapped convolutions ($\phi(x) = x^n - 1$) and subtracted for negative wrapped convolutions ($\phi(x) = x^n + 1$).

Example 1 (Schoolbook multiplication)

Let $a = 12 + x + 7x^2 + 13x^3$ and $b = 4 + 2x + 15x^2 + 0x^3$ be two ring elements in $\mathbb{Z}_q/\langle\phi(x)\rangle = \mathbb{Z}_{17}/\langle x^4 + 1\rangle$. The multiplication of $c = a \cdot b \mod (x^4 + 1)$ applying the schoolbook method according to Equation 2.2 is summarized in the following steps. Note that these steps can be merged into a single one using Equation 2.5.

1. Pad input polynomial to a length of $n' = 2n$: $a = 12 + x + 7x^2 + 13x^3 + 0x^4 + 0x^5 + 0x^6 + 0x^7$ and $b = 4 + 2x + 15x^2 + 0x^3 + 0x^4 + 0x^5 + 0x^6 + 0x^7$.

2. Compute $c' = a \cdot b$, where multiplications with padded zeros are skipped:

$$\begin{aligned} c' &= (a_0 \cdot b_0 \mod q) + (a_0 \cdot b_1 + a_1 \cdot b_0 \mod q)x \\ &+ (a_0 \cdot b_2 + a_1 \cdot b_1 + a_2 \cdot b_0 \mod q)x^2 \\ &+ (a_0 \cdot b_3 + a_1 \cdot b_2 + a_2 \cdot b_1 + a_3 \cdot b_0 \mod q)x^3 \\ &+ (a_1 \cdot b_3 + a_2 \cdot b_2 + a_3 \cdot b_1 \mod q)x^4 \\ &+ (a_2 \cdot b_3 + a_3 \cdot b_2 \mod q)x^5 + (a_3 \cdot b_3 \mod q)x^6 \\ &= 14 + 11x + 6x^2 + 13x^3 + 12x^4 + 8x^5 + 0x^6 + 0x^7 \end{aligned}$$

3. Compute $c = c' \mod x^4 + 1$:

$$\begin{aligned} c = (c'_0 - c'_4 \mod q) + (c'_1 - c'_5 \mod q)x + (c'_2 - c'_6 \mod q)x^2 \\ + (c'_3 - c'_7 \mod q)x^3 = 2 + 3x + 6x^2 + 13x^3 \end{aligned}$$

2.3.2 Number Theoretic Transform (NTT)

The NTT is an efficient method to reduce the complexity of the polynomial multiplication from $\mathcal{O}(n^2)$ to $\mathcal{O}(n \log_2(n))$. It is a variant of the Fast Fourier Transform (FFT) with operations in the field $\mathbb{Z}_q$ instead of the complex numbers [RVM+14].

Let $a, b \in \mathbb{Z}_q/\langle\phi(x)\rangle$ be two ring polynomials. Then, the polynomial multiplication using the forward and inverse NTT can be computed with $c = \text{INVNTT}(\text{NTT}(a) \odot \text{NTT}(b))$, where $\odot$ represents the coefficient-wise multiplication [ADPS16b].

The ring must contain primitive roots of unity to allow an application of the NTT. Let $\omega_n \in \mathbb{Z}_q$ be the n-th root of unity with $\omega_n^n = 1 \mod q$ and $\omega_n^i \neq 1 \mod q$ for $\forall i \in [1, n)$. The forward transform of the coefficients a_i and the inverse transform of $\hat{a}_i$ are computed with

$$\hat{a}_i = \sum_{j=0}^{n-1} \gamma^j \cdot \omega_n^{ij} \cdot a_j \ , \qquad a_i = \frac{1}{n} \cdot \gamma^{-i} \sum_{j=0}^{n-1} \omega_n^{-ij} \cdot \hat{a}_j \ , \tag{2.6}$$

where $\gamma \in \mathbb{Z}_q$.

Positive and negative wrapped convolutions. The multiplications of γ^i before the NTT (preprocessing) and with γ^{-i} after the INVNTT (postprocessing) result into a reduction by $x^n - 1$ or $x^n + 1$ depending on the value of γ. Choosing $\gamma = 1$ leads to positive wrapped convolutions and choosing $\gamma = \gamma_n = \sqrt{\omega_n}$ with $\omega_n^n = 1 \mod q$, $\omega_n^{n/2} = \gamma_n^n = -1 \mod q$, and $n = 2^k$ leads to negative wrapped convolutions. The variable γ_n is also known as the $2n$-th root of unity.

Example 2 (NTT multiplication)

Let $a = 12 + x + 7x^2 + 13x^3$ and $b = 4 + 2x + 15x^2 + 0x^3$ be two ring elements in

$\mathbb{Z}_q/\langle\phi(x)\rangle = \mathbb{Z}_{17}/\langle x^4+1\rangle$. The multiplication of $c = a \cdot b \mod (x^4+1)$ using the NTT can be computed with the following steps.

1. Determine NTT parameters: $\omega_n = 4$, $\gamma_n = 2$, $\omega_n^{-1} = 13$, $\gamma_n^{-1} = 9$, $n^{-1} = 13$

2. NTT preprocessing:

$$\begin{aligned} a'_0 &= a_0 \cdot \gamma_n^0 \mod q = 12 & \qquad b'_0 &= b_0 \cdot \gamma_n^0 \mod q = 4 \\ a'_1 &= a_1 \cdot \gamma_n^1 \mod q = 2 & \qquad b'_1 &= b_1 \cdot \gamma_n^1 \mod q = 4 \\ a'_2 &= a_2 \cdot \gamma_n^2 \mod q = 11 & \qquad b'_2 &= b_2 \cdot \gamma_n^2 \mod q = 9 \\ a'_3 &= a_3 \cdot \gamma_n^3 \mod q = 2 & \qquad b'_3 &= b_3 \cdot \gamma_n^3 \mod q = 0 \end{aligned}$$

3. NTT transformation:

$$\begin{aligned} \hat{a}_0 &= (a'_0 \cdot \omega_n^{0\cdot0} + a'_1 \cdot \omega_n^{1\cdot0} + a'_2 \cdot \omega_n^{2\cdot0} + a'_3 \cdot \omega_n^{3\cdot0}) \mod q = 10 \\ \hat{a}_1 &= (a'_0 \cdot \omega_n^{0\cdot1} + a'_1 \cdot \omega_n^{1\cdot1} + a'_2 \cdot \omega_n^{2\cdot1} + a'_3 \cdot \omega_n^{3\cdot1}) \mod q = 1 \\ \hat{a}_2 &= (a'_0 \cdot \omega_n^{0\cdot2} + a'_1 \cdot \omega_n^{1\cdot2} + a'_2 \cdot \omega_n^{2\cdot2} + a'_3 \cdot \omega_n^{3\cdot2}) \mod q = 2 \\ \hat{a}_3 &= (a'_0 \cdot \omega_n^{0\cdot3} + a'_1 \cdot \omega_n^{1\cdot3} + a'_2 \cdot \omega_n^{2\cdot3} + a'_3 \cdot \omega_n^{3\cdot3}) \mod q = 1 \\ \hat{b}_0 &= (b'_0 \cdot \omega_n^{0\cdot0} + b'_1 \cdot \omega_n^{1\cdot0} + b'_2 \cdot \omega_n^{2\cdot0} + b'_3 \cdot \omega_n^{3\cdot0}) \mod q = 0 \\ \hat{b}_1 &= (b'_0 \cdot \omega_n^{0\cdot1} + b'_1 \cdot \omega_n^{1\cdot1} + b'_2 \cdot \omega_n^{2\cdot1} + b'_3 \cdot \omega_n^{3\cdot1}) \mod q = 11 \\ \hat{b}_2 &= (b'_0 \cdot \omega_n^{0\cdot2} + b'_1 \cdot \omega_n^{1\cdot2} + b'_2 \cdot \omega_n^{2\cdot2} + b'_3 \cdot \omega_n^{3\cdot2}) \mod q = 9 \\ \hat{b}_3 &= (b'_0 \cdot \omega_n^{0\cdot3} + b'_1 \cdot \omega_n^{1\cdot3} + b'_2 \cdot \omega_n^{2\cdot3} + b'_3 \cdot \omega_n^{3\cdot3}) \mod q = 13 \end{aligned}$$

4. Coefficient-wise multiplication:

$$\begin{aligned} \hat{c}_0 &= \hat{a}_0 \cdot \hat{b}_0 \mod q = 0, & \hat{c}_1 &= \hat{a}_1 \cdot \hat{b}_1 \mod q = 11 \\ \hat{c}_2 &= \hat{a}_2 \cdot \hat{b}_2 \mod q = 1, & \hat{c}_3 &= \hat{a}_3 \cdot \hat{b}_3 \mod q = 13 \end{aligned}$$

5. INVNTT transformation:

$$\begin{aligned} c'_0 &= (\hat{c}_0 \cdot \omega_n^{-0\cdot0} + \hat{c}_1 \cdot \omega_n^{-1\cdot0} + \hat{c}_2 \cdot \omega_n^{-2\cdot0} + \hat{c}_3 \cdot \omega_n^{-3\cdot0}) \mod q = 8 \\ c'_1 &= (\hat{c}_0 \cdot \omega_n^{-0\cdot1} + \hat{c}_1 \cdot \omega_n^{-1\cdot1} + \hat{c}_2 \cdot \omega_n^{-2\cdot1} + \hat{c}_3 \cdot \omega_n^{-3\cdot1}) \mod q = 7 \\ c'_2 &= (\hat{c}_0 \cdot \omega_n^{-0\cdot2} + \hat{c}_1 \cdot \omega_n^{-1\cdot2} + \hat{c}_2 \cdot \omega_n^{-2\cdot2} + \hat{c}_3 \cdot \omega_n^{-3\cdot2}) \mod q = 11 \\ c'_3 &= (\hat{c}_0 \cdot \omega_n^{-0\cdot3} + \hat{c}_1 \cdot \omega_n^{-1\cdot3} + \hat{c}_2 \cdot \omega_n^{-2\cdot3} + \hat{c}_3 \cdot \omega_n^{-3\cdot3}) \mod q = 8 \end{aligned}$$

6. NTT postprocessing:

$$\begin{aligned} c_0 &= (c'_0 \cdot n^{-1} \cdot \gamma_n^{-0} \mod q) + (c'_1 \cdot n^{-1} \cdot \gamma_n^{-1} \mod q)x \\ &\quad + (c'_2 \cdot n^{-1} \cdot \gamma_n^{-2} \mod q)x^2 + (c'_3 \cdot n^{-1} \cdot \gamma_n^{-3})x^3 \\ &= 2 + 3x + 6x^2 + 13x^3 \end{aligned}$$

NTT algorithms. When exploiting symmetry, periodicity, and scale properties of the Fourier transformation, the complexity of Equation 2.6 can be reduced with a divide-and-conquer approach from $\mathcal{O}(n^2)$ to $\mathcal{O}(n \log_2(n))$. The two most common methods for splitting a large Fourier transform into smaller pieces are the Cooley–Tukey (CT) [CT65] and the Gentleman–Sande (GS) [GS66] algorithms. The butterfly operation, which is the main operation of these algorithms, consists of arithmetic in $\mathbb{Z}_q$. The Cooley–Tukey Decimation-In-Time (DIT) approach computes $x' \leftarrow x + y \cdot \omega$ and $y' \leftarrow x - y \cdot \omega$ with $\omega, x, y \in \mathbb{Z}_q$ and ω usually a power of ω_n (also known as Twiddle factor). The Gentleman–Sande Decimation-In-Frequency (DIF) approach computes $x' \leftarrow x+y$ and $y' \leftarrow (x-y)\cdot\omega$.

Bit-reversal and algorithm variants. The bit-reversal operation is a particular permutation of a sequence of elements and is inherently part of optimized NTT algorithms. Given an array of n elements, the index of the i-th element a_i can be represented in binary notation $i = \{b_0, b_1, \ldots, b_{log_2(n)-1}\}$. The bit-reversal operation swaps the i-th element with the j-th element having the bit-reversed index $j = \{b_{log_2(n)-1}, b_{log_2(n)-2}, \ldots, b_0\}$.

Different in-place variants of the Cooley–Tukey and Gentleman–Sande algorithms exist. They can be denoted as $\text{NTT}^{CT}_{br\rightarrow no}$, $\text{NTT}^{CT}_{no\rightarrow br}$, $\text{NTT}^{GS}_{br\rightarrow no}$, and $\text{NTT}^{GS}_{no\rightarrow br}$, where, e.g., $no \rightarrow br$ indicates that the input is in normal and the output in bit-reversed order. The bit-reversal can be completely avoided when combining different variants, e.g., $\text{NTT}^{CT}_{no\rightarrow br}$ and $\text{INVNTT}^{GS}_{br\rightarrow no}$ [POG15].

2.3.3 Karatsuba / Toom–Cook

The NTT requires the existence of primitive roots of unity and is therefore not directly applicable to all lattice-based schemes. Two alternatives for performing flexible and efficient multiplications are the Karatsuba and Toom–Cook methods. These algorithms iteratively split a large polynomial multiplication into several smaller ones. Karatsuba outperforms Toom–Cook for low degree polynomials, and Toom–Cook outperforms Karatsuba for large polynomials. Therefore, it is a common approach to combine both methods to achieve an ideal performance. The exact polynomial degree where both approaches perform equally well depends on the implementation and platform. For a 32-bit microcontroller, it was shown in [BMKV20] that Karatsuba performs better than Toom–Cook at polynomial degrees ≤ 64. It must be noted that several other multiplication approaches exist. They, however, have been rarely used in lattice-based cryptography so far and are therefore not further discussed in this section.

Karatsuba multiplication. The Karatsuba algorithm [KO62] reduces the quadratic runtime of the polynomial multiplication to a complexity of $\mathcal{O}(n^{log_2(3)}) \approx \mathcal{O}(n^{1.58})$. It splits the two length-m input polynomials a and b of the multiplication into length-$m/2$ polynomials, i.e., into a lower part (a^l, b^l) and higher part (a^h, b^h). Instead of four polynomial multiplications of these length-half polynomials, Karatsuba's tweak requires only three different multiplications

$$c = as = a^l b^l + ((a^h + a^l)(b^h + b^l) - a^h b^h - a^l b^l)x^{m/2} + a^h b^h x^m \quad . \tag{2.7}$$

Saving this single multiplication costs further additions and subtractions, which are less complex for large polynomial degrees.

Toom–Cook multiplication. The Toom–Cook multiplication [Too63] is a generalization of the Karatsuba algorithm. It is typically divided into the following steps: splitting, evaluation, point-wise multiplication, interpolation, and recombination.

Splitting. The algorithm first splits two polynomials a and b into k parts of length $m = \lceil n/k \rceil$. This is written as $a(x) = \alpha_0 + \alpha_1 x^m + \cdots + \alpha_{k-1} x^{m(k-1)}$ and $b(x) = \beta_0 + \beta_1 x^m + \cdots + \beta_{k-1} x^{m(k-1)}$, where α_i and β_i are the sub-parts of the polynomials a and b. For the case $k = 2$, the k-way Toom–Cook algorithm is identical to the Karatsuba algorithm.

Evaluation. After the splitting, the equations $a(x)$ and $b(x)$ are evaluated for $2k - 1$ carefully selected values. These values are usually chosen such that this step is simple. In practice, the evaluation is a multiplication with the evaluation matrix that is constructed with the selected evaluation points $p_0, p_1, \ldots, p_{2k-2}$. The evaluation for $a(x)$ can be written as

$$\begin{bmatrix} r_0 \\ r_1 \\ \cdots \\ r_{2k-2} \end{bmatrix} = \begin{bmatrix} p_0^0 & p_0^1 & \cdots & p_0^{k-1} \\ p_1^0 & p_1^1 & \cdots & p_1^{k-1} \\ \cdots & \cdots & \cdots & \cdots \\ p_{2k-2}^0 & p_{2k-2}^1 & \cdots & p_{2k-2}^{k-1} \end{bmatrix} \cdot \begin{bmatrix} \alpha_0 \\ \alpha_1 \\ \cdots \\ \alpha_{k-1} \end{bmatrix} \tag{2.8}$$

The evaluation of $b(x)$ works equally.

Point-wise multiplication. The evaluated results are point-wise multiplied, i.e., $r' = [a(p_0) \cdot b(p_0), a(p_1) \cdot b(p_1), \ldots, a(p_{2k-2}) \cdot b(p_{2k-2})]$ is computed.

Interpolation. The interpolation step is the reverse operation of the evaluation and can be again realized with a matrix multiplication.

$$\begin{bmatrix} c'_0 \\ c'_1 \\ \cdots \\ c'_{2k-2} \end{bmatrix} = \begin{bmatrix} p_0^0 & p_0^1 & \cdots & p_0^{2k-2} \\ p_1^0 & p_1^1 & \cdots & p_1^{2k-2} \\ \cdots & \cdots & \cdots & \cdots \\ p_{2k-2}^0 & p_{2k-2}^1 & \cdots & p_{2k-2}^{2k-2} \end{bmatrix}^{-1} \cdot \begin{bmatrix} r'_0 \\ r'_1 \\ \cdots \\ r'_{2k-2} \end{bmatrix} \tag{2.9}$$

Recombination. This step leads to the final multiplication result. The product polynomial is recombined according to $c(x) = c'_0 + c'_1 x^m + \cdots + c'_{2k-2} x^{m \cdot (2k-2)}$. Toom–Cook requires in total $\mathcal{O}(n^{\frac{log_2(2k-1)}{log_2(k)}})$ primitive operations.

Example 3 (Karatsuba/Toom–Cook multiplication)

Let $a = 12 + x + 7x^2 + 13x^3$ and $b = 4 + 2x + 15x^2 + 0x^3$ be two ring elements in $\mathbb{Z}_q/\langle\phi(x)\rangle = \mathbb{Z}_{17}/\langle x^4 + 1\rangle$. The multiplication of $c = a \cdot b \mod (x^4 + 1)$ using Toom–Cook with a split of $k = 2$ can be computed with the following steps. This specific case corresponds to the Karatsuba algorithm.

1. Splitting:

$$a = \alpha_0 + \alpha_1 x^2, \text{ with } \alpha_0 = 12 + x, \ \alpha_1 = 7 + 13x$$
$$b = \beta_0 + \beta_1 x^2, \text{ with } \beta_0 = 4 + 2x, \ \beta_1 = 15 + 0x$$

2. Evaluation at $2k-1$ points: $p_0 = 0, p_1 = 1, p_2 = \infty$, where the evaluation at infinity is defined as taking the coefficient with highest degree and setting all others to zero. The multiplication with the evaluation matrix according to Equation 2.8 results in

$$\begin{bmatrix} r_0 \\ r_1 \\ r_2 \end{bmatrix} = \begin{bmatrix} 1 & 0 \\ 1 & 1 \\ 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} \alpha_0 \\ \alpha_1 \end{bmatrix} = \begin{bmatrix} 12 + x \\ 19 + 14x \\ 7 + 13x \end{bmatrix}, \quad \begin{bmatrix} s_0 \\ s_1 \\ s_2 \end{bmatrix} = \begin{bmatrix} 1 & 0 \\ 1 & 1 \\ 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} \beta_0 \\ \beta_1 \end{bmatrix} = \begin{bmatrix} 4 + 2x \\ 19 + 2x \\ 15 \end{bmatrix}$$

3. The pointwise multiplication clearly illustrates that only three different polynomial multiplications for Karatsuba are required. It leads to:

$$r'_0 = r_0 \cdot s_0 = 48 + 28x + 2x^2$$
$$r'_1 = r_1 \cdot s_1 = 361 + 304x + 28x^2$$
$$r'_2 = r_2 \cdot s_2 = 105 + 195x$$

4. Interpolation using Equation 2.9:

$$\begin{bmatrix} c'_0 \\ c'_1 \\ c'_2 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 \\ -1 & 1 & -1 \\ 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} r'_0 \\ r'_1 \\ r'_2 \end{bmatrix} = \begin{bmatrix} 48 + 28x + 2x^2 \\ 208 + 81x + 26x^2 \\ 105 + 195x \end{bmatrix}$$

5. Finally, the recombination gives

$$\begin{aligned} c = ((48 \mod q) + (28 \mod q)x + (2 + 208 \mod q)x^2 + (81 \mod q)x^3 \\ + (26 + 105 \mod q)x^4 + (195 \mod q)x^5) \mod x^n + 1 \\ = (14 + 11x + 6x^2 + 13x^3 + 12x^4 + 8x^5) \mod x^n + 1 = 2 + 3x + 6x^2 + 13x^3 \end{aligned}$$

2.4 Polynomial Sampling and Randomness Generation

Most post-quantum cryptosystems require a tremendous amount of randomness. This is particularly true for the generation (sampling) of random polynomials in lattice-based cryptography, where a large data set of uniformly distributed randomness is used to construct the coefficients according to the desired output distribution (Gaussian, binomial, or uniform). In order to produce this large amount of uniform randomness, a small seed from a physical source of randomness can be expanded using a PRNG. The primitives SHA-3, AES, and ChaCha20 are particularly suitable for this task. Among these three alternatives, SHA-3 is the most performant option in hardware because it generates the highest amount of pseudorandom bits per round [BUC19]. SHA-3 is a subset of the Keccak family standardized by NIST. The standard lists four specific instances of SHA-3 and two XOFs (SHAKE-128 and SHAKE-256). The two SHAKE variants permit extracting a variable output length, which is ideal for the pseudorandom bit generation.

The most important part of the SHA-3 and SHAKE primitives is the Keccak permutation, which calls in each of 24 rounds the `f-1600` function. The `f-1600` function is divided into five steps: Theta (θ), Rho (ρ), Pi (π), Chi (χ), and Iota (ι). These steps have an input state A (1 600-bit) and a processed output state B. The states can be represented in a three-dimensional array containing 25 words, each with a length of 64-bit. These words can be structured in a cube with x and y coordinates indexed from $0 \leq x < 5$ and $0 \leq y < 5$. Each bit of this cube can be addressed with $A[x, y, z]$. The following conventions are used to facilitate the description of the `f-1600` function: the part of the state which presents the word is also called lane, a two-dimensional part of the state with fixed z is called a slice, and all lanes with the same x-coordinate form a sheet. The following paragraphs describe the steps of the `f-1600` function according to the definition of the Keccak team [BDPVA09].

Theta Step (θ). This step first computes the parity of each sheet of the state $C[x] = A[x, 0] \oplus A[x, 1] \oplus A[x, 2] \oplus A[x, 3] \oplus A[x, 4]$ for x in $[0, 4]$. The output of the θ step can then be computed with $B[x, y] = A[x, y] \oplus (C[x - 1] \oplus \texttt{rot}(C[x + 1], 1))$ for x and y in $[0, 4]$, where `rot` denotes the rotation. Note that the bit addressing is computed modular 5.

Rho/Pi Step (ρ/π). The ρ and π steps work on the lanes of the state $A[x, y]$. Therefore, they are usually processed together. The ρ step rotates the lane by a constant offset $r[x, y]$ that depends on the x and y positions. This can be formulated as $B[x, y] = \texttt{rot}(A[x, y], r[x, y])$ for x and y in $[0, 4]$. The π step swaps the complete lanes of the state according to $B[y, 2x + 3y] = A[x, y]$ for x and y in [0,4].

Chi/Iota Step (χ/ι). The non-linear χ step can be computed according to $B[x, y] = A[x, y] \oplus (\overline{A}[x+1, y] \wedge A[x+2, y])$ for x and y in [0,4]. At the ι step, a round constant R_c is XORed to the lane $A[0, 0]$, i.e., $B[0, 0] = A[0, 0] \oplus R_c[i]$. This round constant depends on the current round i. Also, the χ and ι step can be combined.

2.5 Hardware Accelerators and Coupling Strategies

Hardware accelerators can be used to accelerate computationally intensive operations and thus reduce the load of the main processor. Especially, small and less powerful processors benefit from an application-specific hardware accelerator.

The need for hardware acceleration. Many small embedded devices could not meet performance requirements without hardware acceleration. This is particularly critical for real-time applications where reaction times and responses must be guaranteed to be within a small time frame. Failing those requirements can lead to severe safety-critical problems for some applications. Performance improvements are an obvious advantage of hardware accelerators, but several others exist. The reduction of the energy consumption and a controlled execution are two of them. The energy consumption has a high correlation with the execution time. When the execution time is reduced, the energy consumption is reduced as well if the power dissipation of the circuit remains in a similar range. Specialized operations within the accelerator can avoid costly intermediate calculations and memory accesses, leading to a positive influence on the energy consumption. The energy consumption is particularly important for battery-powered devices, which are highly present in the IoT market. The controlled execution of cryptographic algorithms is of high relevance to avoid SCA. Constant-time implementations of algorithms are easier to achieve with a hardware design as the constant-time behavior is more natural for hardware circuits. Moreover, the data movement is more defined for hardware circuits. This reduces unexpected microarchitectural leakages and facilitates certifiability.

Performance vs. flexibility. The level of flexibility is another important criterion, particularly for products with a long lifespan. New security issues might lead to the necessity of parameter set changes, algorithmic modifications, or integration of further SCA countermeasures. While software implementations achieve a high level of flexibility, hardware implementations achieve a high performance but a poor flexibility. FPGAs provide a certain degree of flexibility but updating hardware circuits is usually difficult due to their high complexity. For ASIC implementations, updating a circuit is nearly impossible and redesigns are not only time-consuming but also extremely expensive.

Hardware/software codesign. Hardware/software codesigns can be deployed to combine the advantages of flexible software and fast hardware solutions. Performance-critical operations are executed with hardware accelerators, while the general control flow of the algorithm is executed in software to keep a high level of flexibility. The actual implementation method significantly influences the execution time, energy consumption, chip area, and flexibility. These parameters are mutually dependent, and a compromise must be found.

Accelerator types and coupling strategies. Table 2.1 and Figure 2.1 summarize the different implementation types and hardware accelerator coupling strategies. The per-

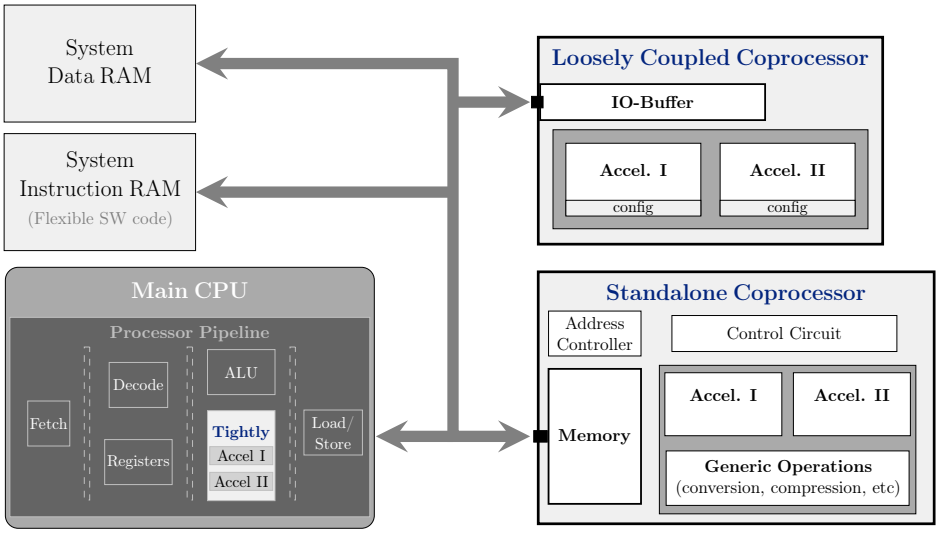


Figure 2.1: Coupling strategies.

Table 2.1: Implementation types and coupling strategies.

Type	Advantages	Disadvantages
Standalone coprocessor	Highest performance Lowest energy consumption	Lowest flexibility Highest hardware complexity
Loosely coupled coprocessor	Good performance Low energy consumption	Low/moderate flexibility Intensive bus communication
Tightly coupled accelerator	Good performance Low energy consumption High flexibility	Core modifications required Instruction set must be modified
Plain software	Highest flexibility	Lowest performance Highest energy consumption

formance, energy consumption, and flexibility behave exactly the opposite way for standalone coprocessors (monolithic hardware solutions) and plain software implementations. Many applications have a main processor that is not used when the crypto coprocessor is active. Hardware/software codesign solutions, however, use the available main processor resources. Such codesigns can be categorized into loosely and tightly coupled solutions. Loosely coupled coprocessors are connected to the main processor via a system bus, e.g., Advanced eXtensible Interface (AXI). Tightly coupled accelerators are directly integrated into the processor. Both coupling strategies, loosely and tightly, have advantages and disadvantages. A thorough analysis of these coupling strategies in the context of PQC is provided in Chapters 3–4.

FPGA-SoC platforms. FPGAs are getting increasingly powerful and already offer complete system solutions. The combination of one or multiple processors with programmable logic in one FPGA increases the flexibility and allows extremely fast design periods.

Such FPGA-SoCs provide the ideal environment for creating fast and efficient hardware/software codesigns. The strengths of FPGA-SoCs are demonstrated with a hardware/software codesign example for NTRU in Section 3.2.

RISC-V platforms. RISC-V is an open Instruction Set Architecture (ISA) based on the Reduced Instruction Set Computer (RISC) principles. The RISC-V initiative started in 2010 by the University of California, Berkley, and has meanwhile grown to a large non-profit corporation [RIS21]. RISC-V provides a free, open, flexible, and extensible ISA usable for embedded systems and high performance computers. Several open-source hardware implementations supporting the RISC-V ISA exist. Implementations that have drawn particular attention are Rocket Chip[1], VexRiscv[2], and the RISC-V cores from Parallel Ultra Low Power (PULP)[3].

Rocket Chip is based on the hardware construction language Chisel[4]. The implementation offers a dedicated interface, called Rocket Custom Coprocessor (RCC), to extend the system with hardware accelerators. However, the integration of tightly coupled accelerators is not straightforward on this platform. VexRiscv was developed using another high-level hardware description language called SpinalHDL[5]. The VexRiscv project allows modifications of the processor [AEL+20]. The PULP project features three different RISC-V cores designed using the hardware description language SystemVerilog. The CVA6 core (formerly Ariane) is a 6-stages 64-bit solution. For smaller embedded devices, the PULP team offers the in-order execution 2-stages 32-bit solution Ibex (formerly Zero-riscy) and the 4-stages 32-bit solution CV32E40P (formerly RI5CY). The RISC-V cores CV32E40P and Ibex can be integrated into the single-core microcontroller platform PULPino[6], offering a rich set of peripherals such as I2C, SPI, UART, and GPIO [TZS+16].

Selection of RISC-V core. The use of standard SystemVerilog allows selective modifications of the entire design. This makes the PULP cores ideally suitable for this work. Although the CV32E40P core is slightly inferior, it is suitable for comparisons with the ARM Cortex-M4, the most popular microcontroller for PQC benchmarks. Therefore, this RISC-V core is used for the hardware/software codesigns proposed in Chapters 3–5, 7.

RISC-V ISA extension. To enhance the basic integer instruction set (I), RISC-V defines several standard extensions: the extensions for multiplication/division (M); single, double, and quad precision floating-point operations (F, D, Q); atomic operations (A); and compressed instructions (C). The CV32E40P core fully supports the I, M, F, and C instruction sets. In addition, it provides the PULP-specific extension Xpulp, which

[1] https://github.com/chipsalliance/rocket-chip (Last accessed 1st Nov. 2021).
[2] https://github.com/SpinalHDL/VexRiscv (Last accessed 1st Nov. 2021).
[3] https://github.com/pulp-platform (Last accessed 1st Nov. 2021).
[4] https://www.chisel-lang.org (Last accessed 1st Nov. 2021).
[5] https://github.com/SpinalHDL (Last accessed 1st Nov. 2021).
[6] https://github.com/pulp-platform/pulpino (Last accessed 1st Nov. 2021).

includes hardware loops, SIMD extensions, bit manipulations, and post-increment instructions. This work further develops the PQC extension (Chapters 4, 5, 7).

RISC-V defines four different base instruction format types: R-type, I-type, S-type, and U-type. Depending on the instruction type, the instruction structure consists of an *opcode*, function fields, immediate values, the source registers $rs1$ and $rs2$, and the destination register rd. Most PQC instructions developed in this work use the R-type. An R-type 32-bit instruction follows the format $\{func7, rs2, rs1, func3, rd, opcode\}$, where 7-bit are reserved for *opcode*, 3×5-bit for $rs1$, $rs2$, and rd, 3-bit for $func3$, and 7-bit for $func7$. The use of function fields ($func3/func7$) allows designing multiple operations with only a single opcode. This thesis uses the available opcode 0x77 for the PQC instructions. More details about the instruction formats can be found in the RISC-V instruction set manual [WAE19].

3 Loosely Coupled Coprocessors for PQC

In this chapter, hardware circuits and coprocessors to accelerate computationally expensive operations of PQC are presented. The two popular post-quantum algorithms NTRU and NewHope are chosen as a case study. They represent the two main branches of lattice-based cryptography: NTRU-based and LWE-based systems. With an FPGA-SoC and RISC-V SoC, two different platforms were used for the coprocessor evaluations. The NTRU system design was published in [BFM+18, FSF+19]. For the description of the NTRU coprocessor, small fractions of the author's publication [FSS20a] are also used. An earlier version of the NewHope system design was published in [FS19, FSM+19]. Moreover, a small part of the author's publication [FSS20b] is used for the description of the NewHope coprocessor.

3.1 Introduction Loosely Coupled PQC Coprocessors

A coprocessor is an additional hardware block that supports the main processor at computationally intensive tasks. Such coprocessors are usually connected by a bus to the main processor, as mentioned in Section 2.5 and illustrated in Figure 3.1. In this chapter, it is analyzed if loosely coupled coprocessors are suitable for lattice-based PQC. In contrast to previous works, the focus of this work is not on standalone hardware solutions for a whole algorithm but on hardware/software codesign solutions. While standalone

solutions achieve a high performance, they usually suffer from a low flexibility. Hardware/software codesign techniques can be used to meet performance, energy, and flexibility requirements. Such approaches can also significantly improve development cycles as hardware designs must be only designed for specific bottlenecks. The overall control flow and algorithm execution remain in software. As a case study, this chapter analyzes hardware/software codesigns for the lattice-based algorithms NTRU and NewHope.

Related works. The development of hardware circuits for lattice-based cryptography started soon after the invention of NTRU. In 2001, the authors of [BCE+01] proposed the probably first NTRU encryption hardware circuit. Subsequent works focused on acceleration techniques for the costly polynomial multiplications of NTRU [ABF+08, KY09, LW15]. The work in [ABF+08] targeted a small multiplier design for low-cost devices, and [KY09] exploited the special structure of NTRU polynomials (several zero-coefficients) to reduce the overall amount of multiplications. The authors in [LW15] proposed to model the polynomial multiplication using a Linear-Feedback Shift Register (LFSR) structure. This approach was later improved in [LW16], where a circuit was developed that skips the multiplication whenever two consecutive zero-coefficients are detected. As shown by the author of this thesis and other coauthors of [BFM+18, FSF+19], this approach leaks timing side-channel information and thus reduces the security level of NTRU. In [BFM+18], the author of this thesis and other coauthors of the publication presented a full hardware implementation of the IEEE 1363.1 version of NTRU. Interested readers might have a deeper look directly at the publications [BFM+18, FSF+19] as the full NTRU hardware implementation and the timing side-channel leakage of the NTRU implementation in [LW16] are not further discussed in this dissertation. A hardware design of the NTRU Prime streamline variant was proposed in [Mar20].

Over the years, the focus shifted due to patent issues from NTRU-based to LWE-based schemes. The main optimization target for LWE-based cryptography has also been the polynomial multiplication—particularly multiplications with NTT. First hardware implementations of the NTT multiplication for PQC were proposed in [GFS+12, PG12]. The authors in [APS13] and [RVM+14] improved the NTT memory access strategy and presented approaches to compute the Twiddle factor (required for the NTT) on the fly. Complete standalone hardware solutions were later designed for NewHope variants [OG17, KLC+17, ZYC+20], Kyber [HHLW20], Saber [SRB20], and Frodo [HOKG18].

Sapphire [BUC19] and VPQC [XHY+20] are two coprocessor solutions where the main processor is mostly used for configuration purposes. Sapphire supports multiple algorithms: Frodo, NewHope, qTESLA, Kyber, and Dilithium. The coprocessor VPQC is able to support NewHope, Kyber, and LAC. Still, these processors have a limited flexibility as almost all the computations are executed in hardware. One of the first real hardware/software codesigns with a lattice-based cryptography coprocessor was proposed by the thesis author in [FSF+19]. The content, an NTRU system design, is summarized in Section 3.2. In contrast to the strong hardware-oriented solutions, the presented approach executes the complete algorithmic flow in software and performs only critical operations in hardware. A few months later, the authors of [FND+19] proposed a similar hard-

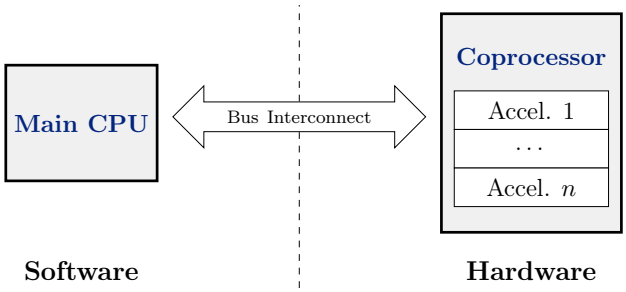


Figure 3.1: Loosely coupled coprocessor.

ware/software codesign for NTRU, which was further extended to more NTRU variants in [FDNG19]. The authors in [AHH+18] reused an existing RSA coprocessor to develop a hardware/software codesign for the LWE-based scheme Kyber. The most probably first complete hardware/software codesign with a coprocessor specially developed for LWE-based cryptography was presented by the thesis author in [FSM+19]. The summarized content, a codesign for NewHope, is provided in Section 3.3. Another hardware/software codesign for the LWE-based signature scheme qTESLA was presented in [WTJ+20].

Contributions. This chapter analyzes the suitability of lattice-based cryptography for flexible hardware/software codesign solutions. The two schemes NTRU and NewHope (LWE-based) are investigated to represent the two main directions. As target platforms, an FPGA-SoC is selected for NTRU and a RISC-V SoC for NewHope.

The contributions of this chapter can be summarized as follows:

- Design of hardware accelerators for NTRU. This includes an efficient modular p reduction circuit and a flexible ternary multiplier. The ternary multiplier is enhanced to support the non-ternary multiplications in NTRU;
- Integration, hardware/software codesign, and evaluation of NTRU on an FPGA-SoC;
- Design of hardware accelerator blocks for NewHope. This includes an NTT multiplier and a generic sampling module;
- Analysis of hardware optimizations for the NTT multiplier, including the integration of the NTT post-processing into the main algorithm, integration of efficient reduction routines, and power optimizations;
- Integration, hardware/software codesign, and evaluation of NewHope on a RISC-V SoC.

3.2 Use Case: NTRU on FPGA-SoC Platform

This section introduces the algorithmic components of the IEEE 1363.1 version of NTRU, describes the hardware accelerators designed for this NTRU variant, and finally shows the coprocessor integration for an FPGA-SoC platform. The methods presented in this section are, in general, also applicable to other NTRU variants, i.e., to the NTRU NIST submissions.

3.2.1 Algorithmic Operations in NTRU

The IEEE version of NTRU is based on the Shortest Vector Encryption Scheme (SVES), which uses the principles of the NTRU Asymmetric Encryption Padding (NAEP) transform [HGSSW03] to provide security against CCA. As discussed in Section 2.2.2, the encryption and decryption operations of NTRU require additions, subtractions, and multiplications of ring elements. The polynomial ring additions and subtractions can be efficiently computed. The polynomial multiplications $\{r \cdot h\}$ at the encryption and $\{f \cdot e, r \cdot h, r_{calc} \cdot h\}$ at the decryption are much more expensive and therefore the main target for optimizations. Previous works also concluded that the multiplications are the main bottleneck of NTRU [GPM+17]. Other important building blocks are the functions `BPGM` and `MGF`. They mainly consist of computationally intensive hash calculations. Optimizations of these two functions have been left as future work.

Modular reduction. The ring arithmetic in NTRU requires modular reductions by q and p. The modulus q is often chosen as a power of two, turning reductions into a simple bit masking. The parameter p is set for the IEEE 1363.1 version of NTRU to 3. For the Mersenne prime number $p = 3$, a fast reduction routine, shown in Algorithm 7, is applicable. It is based on [Jon01, GPM+17]. The algorithm makes use of the fact that $2^8 \mod 3 \equiv 1$, $2^4 \mod 3 \equiv 1$, and $2^2 \mod 3 \equiv 1$. This allows the reduction of the input to a smaller value congruent to modular 3, e.g., $x \mod 3 = (x/2^8 + (x \mod 2^8)) \mod 3$. The algorithm iteratively reduces the input range of x until it is small enough to use a LUT for the final reduction. Note that divisions correspond to logical shifts and modular reductions to AND operations with a bitmask for a power-of-two value.

Algorithm 7: Modular reduction $p = 3$

Input: Integer $x \in [0, 2^{16})$
Result: Integer $z = x \mod 3$

1 $x = (x \gg 8) + (x \wedge 0\text{xFF})$ // $x \mod 3 = (x/2^8 + (x \mod 2^8)) \mod 3$
2 $x = (x \gg 4) + (x \wedge 0\text{xF})$ // $x \mod 3 = (x/2^4 + (x \mod 2^4)) \mod 3$
3 $x = (x \gg 2) + (x \wedge 0\text{x3})$ // $x \mod 3 = (x/2^2 + (x \mod 2^2)) \mod 3$
4 $x = (x \gg 2) + (x \wedge 0\text{x3})$ // $x \mod 3 = (x/2^2 + (x \mod 2^2)) \mod 3$
// Now $x < 6$
5 $\text{LUT} = \{0, 1, 2, 0, 1, 2\}$
6 $z = \text{LUT}[x]$ // Corresponds to final conditional subtraction

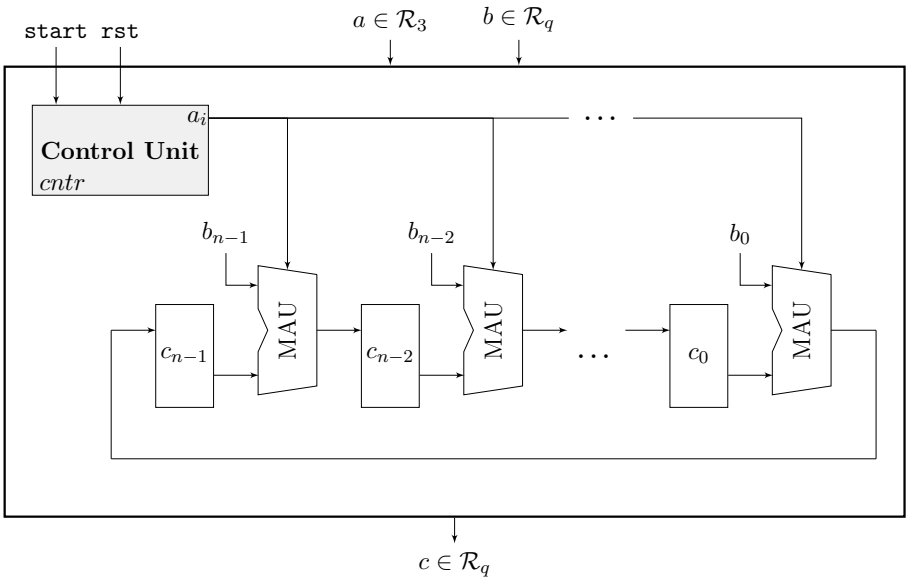


Figure 3.2: Ternary multiplication accelerator.

3.2.2 Ternary Polynomial Multiplication Accelerator

NTRU requires the multiplications $r \cdot h$ and $f \cdot e$, where $h, f, e \in \mathcal{R}_q$ with $q = 2048$ and $r \in \mathcal{R}_p$ with $p = 3$. The hardware architecture of this work is based on the efficient LFSR convolution method of [LW15]. Their architecture is able to multiply a ternary polynomial in $\mathcal{R}_p$ with a polynomial in $\mathcal{R}_q$. The circuit is enhanced by a control unit to increase the flexibility of their architecture and to support the non-ternary multiplication of $f \cdot e$. Figure 3.2 illustrates the architecture of this optimized ternary multiplication module.

Architecture of the ternary multiplication module. This module multiplies $a \in \mathcal{R}_p$ (coefficients in $\{-1, 0, 1\}$) with $b \in \mathcal{R}_q$ (coefficients in $[0, q)$) to compute $c = a \cdot b \mod x^n - 1$ in n clock cycles. The circularity of the convolution for polynomial multiplications is achieved due to the LFSR structure, which shifts the coefficients of c in each cycle to the right. The register width is set to 11-bit to support computations in $\mathcal{R}_{2048}$. The feedback loop realizes the positive wraparound, i.e., the reduction by $x^n - 1$. The polynomial a is forwarded to the *Control Unit*, which serializes the ternary coefficients a_i, starting from a_0, at the first clock cycle, until a_{n-1}. The coefficients of b are always assigned to the first input of the Modular Arithmetic Unit (MAU), which performs the actual arithmetic operations. The MAUs have three operation modes (addition, subtraction, and forwarding), which are activated depending on the value of the coefficient a_i: i) when 1, $c_i + b_i \mod q$ is calculated; ii) when -1, $c_i - b_i \mod q$ is calculated; and iii) when 0, c_i is forwarded. This way, multiplications are completely avoided. The width of the MAU is set to 11-bit. Keeping only the least significant 11 bits during the computations realizes the modular reduction by $q = 2048$. The multiplication $r \cdot h$

can be directly supported with this ternary multiplier within n cycles.

Support for non-ternary multiplication. In this work, the fact $f = 1+pF$ with $F \in \mathcal{R}_p$ is exploited to use the ternary multiplier for the multiplication of $f \cdot e$. The multiplication of the secret key with the ciphertext can be reformulated $f \cdot e = (1 + pF) \cdot e = e + pF \cdot e$. In order to compute $pF \cdot e$, the convolution of $F \cdot e$ is repeated two more times ($p = 3$) without resetting the registers in between. After the first n cycles, the result registers contain the value $F \cdot e$, after $2n$ cycles $F \cdot e + F \cdot e$, and after $3n$ cycles $3F \cdot e$. In order to add e on the result of $pF \cdot e$, the control unit sets the first input polynomial to $a = 1$ and the second polynomial will remain e. After further n cycles, the desired result $e + pF \cdot e$ is computed. It is also possible to skip this round if the registers are preloaded with e at the beginning of the decryption process.

3.2.3 NTRU System Design for an FPGA-SoC

This section presents the developed hardware/software codesign of NTRU. The general NTRU algorithm is executed in software and the performance-critical multiplications are outsourced to the hardware coprocessor. The focus of this section is on the encryption and decryption routines as the NTRU key generation must be executed only once.

FPGA-SoC interfaces. Modern FPGA-SoCs provide multiple ports to transfer data between the processing system (e.g., CPU or memory) and programmable logic (e.g., coprocessor). The interfaces can be frequently divided into General-Purpose Port (GP), High Performance Port (HP), and Accelerator Coherency Port (ACP). The GP interface is mainly used for small data transfers, i.e., for configurations and status requests. HP and ACP interfaces allow high-speed transfers and direct system memory access (e.g., to DDR-RAM). The main difference between HP and ACP is that ACP ensures cache coherency.

AXI4 is frequently used in embedded devices to realize the on-chip communication. The AXI-Lite specification is suitable for simple memory-mapped data transfers and thus for the GP interface. For the high-speed interfaces, AXI-Full and AXI-Stream are preferred. The advantage of AXI-Full is that burst transfers are supported. AXI-Stream is suitable for unidirectional data streams between two points without requiring any address signals.

NTRU system design. Figure 3.3 shows the proposed design of the NTRU FPGA-SoC architecture. The processor of the processing system runs the NTRU algorithm and configures the coprocessor via the GP interface. The coprocessor accelerates the computationally intensive polynomial multiplications ($r \cdot h$ and $f \cdot e$) with optional consecutive modular reductions by $p = 3$. The consecutive reduction by p is required for the computation of $f \cdot e \mod p$ during the decryption (see Section 2.2.2, Algorithm 3).

The Direct Memory Access (DMA) module allows an efficient data transfer between coprocessor and the system's DDR-RAM. The main processor controls the configuration of the DMA module via the GP interface. The configuration includes the memory transfer

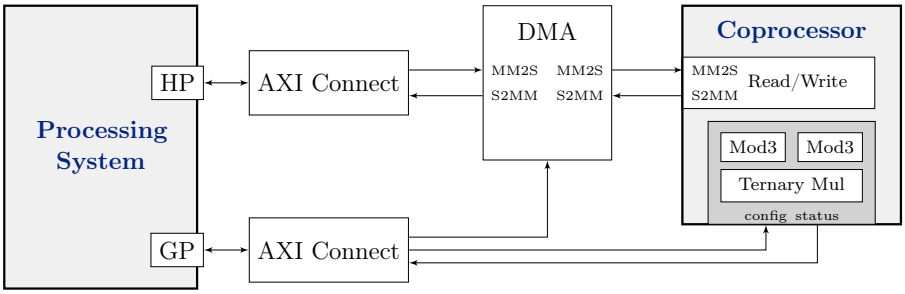


Figure 3.3: NTRU hardware/software codesign for an FPGA-SoC.

start and end addresses, the direction, and the trigger for the data transfer. With the two AXI-Stream interfaces, the coprocessor achieves simple and fast data transfers. The abbreviation MM2S indicates memory-mapped to stream ports and S2MM stream to memory-mapped ports. One data packet contains two coefficients in $\mathbb{Z}_p$ (2×2-bit) and two coefficients in $\mathbb{Z}_q$ (2×11-bit) to exploit the 32-bit bus width. Remaining bit positions are packed with zeros, i.e., the packet has the structure $(000 \,||\, a_{i+1} \,||\, b_{i+1} \,||\, 000 \,||\, a_i \,||\, b_i)$ with $a \in \mathcal{R}_p$ and $b \in R_q$.

The coprocessor has the states *idle*, *read*, *write*, *mul_ter*, and *mul_gen*. The state switches from *idle* to *read* when the DMA sends a valid input packet and remains in this state until the last packet signal is identified. Depending on the `config` signal, the state machine switches to *mul_ter* for usual ternary multiplications or to *mul_gen* for generic multiplications in the form $f \cdot e$ according to the description in Section 3.2.2 (paragraph: "Support for non-ternary multiplication"). During the write state, the coefficients of the polynomial product c are transferred back to the system's memory. Optionally, a modular p reduction according to Algorithm 7 is performed at this step. Each data packet of the transfer of c contains two coefficients. Hence, two reduction accelerators are instantiated to optionally reduce the coefficients modular p before they are transmitted.

3.2.4 Experimental Results of NTRU System Design

The evaluation platform of this section is the Xilinx Zynq UltraScale+ MPSoC ZCU102, which is equipped with a quad-core ARM Cortex-A53. For the baseline software implementation, the library of the NTRU open-source project [Why17] is used. It is compliant with the IEEE-1363.1 standard and was designed by the NTRU authors. The software polynomial multiplication exploits the sparsity of the ternary polynomial and skips computations with coefficients equal to zero.

Parameter sets. The IEEE-1363.1 standard defines 12 different parameter sets for NTRU. They are summarized in Table 3.1. The parameter sets are categorized into the security levels 112-bit, 128-bit, 192-bit, and 256-bit. Three parameter sets for different design goals are defined for each security level. The targeted design goals are: i) size, ii) cost, defined as $(\text{operation time})^2 \times \text{size}$, and iii) speed. The parameter sets mainly differ in the polynomial length n and the sampling parameters (e.g., d_F, d_g). The

Table 3.1: NTRU parameter sets as defined in [L+01].

	Size	Cost	Speed
112 bit	ees401ep1	ees541ep1	ees659ep1
128 bit	ees449ep1	ees613ep1	ees761ep1
192 bit	ees677ep1	ees887ep1	ees1087ep1
256 bit	ees1087ep2	ees1171ep1	ees1499ep1

Table 3.2: Resource utilization of NTRU hardware/software codesign.

Parameter set	Total LUT/FF	Coprocessor LUT/FF	Ternary Mul. LUT/FF	DMA LUT/FF
ees401ep1	19 582/16 104	15 572/10 710	11 378/5 342	1 263/1 759
ees541ep1	25 108/19 729	21 090/14 335	15 167/7 074	1 273/1 759
ees659ep1	29 181/22 787	25 169/17 393	18 454/8 614	1 270/1 759
ees449ep1	21 463/17 375	17 440/11 981	12 763/5 980	1 272/1 759
ees613ep1	27 547/21 583	23 533/16 189	17 169/8 013	1 265/1 759
ees761ep1	32 615/25 449	28 602/20 055	21 303/9 944	1 267/1 759
ees677ep1	29 891/23 249	25 880/17 855	18 956/8 848	1 267/1 759
ees887ep1	38 670/28 752	34 659/23 358	24 832/11 590	1 268/1 759
ees1087ep1	46 788/33 966	42 773/28 572	30 423/14 199	1 271/1 759
ees1087ep2	46 788/33 966	42 773/28 572	30 423/14 199	1 271/1 759
ees1171ep1	50 402/36 183	46 384/30 789	32 757/15 296	1 272/1 759
ees1499ep1	63 221/44 766	59 197/39 372	41 912/19 574	1 276/1 759

naming convention of the parameter set is `eesXXXXepY`, where `XXXX` is the polynomial length. As the design goal classification is made for a software reference implementation, this classification does not need to hold for the hardware/software codesign of this thesis.

Resource utilization. Table 3.2 summarizes the resource utilization of the programmable logic for the proposed design with different NTRU parameter sets. The evaluation shows that the required LUTs and FFs (registers) highly depend on the polynomial length. In particular, the coprocessor, containing the ternary multiplier, increases with larger polynomial degrees. The size of the DMA module does not depend on the parameter set. It is, however, relatively large as it requires, in addition to the listed LUTs and registers, two Block RAM (BRAM) instances.

Performance. The ARM Cortex-A53 is set to a target frequency of 1 200 MHz (real 1 199.88 MHz) and the hardware accelerator to a target frequency of 200 MHz (real 187.48 MHz). The runtime was measured using the cycle count register provided by the Performance Monitor Unit (PMU) of the ARM Cortex-A53 and was verified with a hardware counter. Table 3.3 presents the clock cycle counts for the NTRU baseline software implementation with and without hardware accelerator, including all communication overhead (optimization flag `-O3`). The measured clock cycles are related to the clock of the processing system. One clock cycle in hardware corresponds to roughly six cycles in the processing system.

Table 3.3: Cycle count in kilo cycles of NTRU hardware/software codesign and comparison to reference implementation.

Parameter set	Mult. ternary ref./opt.	Mult. generic ref./opt.	Encryption ref./opt.	Decryption ref./opt.
ees401ep1	141.6/13.0(×10.9)	153.3/24.3(×6.3)	257.3/131.5(×2.0)	421.5/164.0(×2.6)
ees541ep1	81.1/14.0(×5.8)	97.0/29.2(×3.3)	180.1/114.8(×1.6)	289.1/154.3(×1.9)
ees659ep1	73.4/16.0(×4.6)	92.8/34.7(×2.7)	181.2/124.8(×1.5)	273.6/156.0(×1.8)
ees449ep1	183.7/14.6(×12.6)	196.3/27.2(×7.2)	316.5/149.9(×2.1)	524.5/186.5(×2.8)
ees613ep1	100.5/15.3(×6.6)	118.1/32.8(×3.6)	211.8/128.4(×1.6)	343.2/171.8(×2.0)
ees761ep1	90.6/17.6(×5.1)	113.7/39.3(×2.9)	215.3/144.8(×1.5)	329.6/182.7(×1.8)
ees677ep1	307.1/19.6(×15.7)	327.7/38.5(×8.5)	482.0/194.7(×2.5)	824.5/247.9(×3.3)
ees887ep1	201.7/21.2(×9.5)	229.1/47.0(×4.9)	365.9/187.5(×2.0)	613.1/249.8(×2.5)
ees1087ep1	189.5/24.5(×7.7)	221.4/55.2(×4.0)	374.8/212.1(×1.8)	593.9/263.9(×2.3)
ees1087ep2	357.1/25.6(×13.9)	389.4/56.8(×6.9)	567.8/239.5(×2.4)	980.8/318.5(×3.1)
ees1171ep1	335.1/26.5(×12.6)	369.3/60.2(×6.1)	553.0/247.8(×2.2)	947.1/330.6(×2.9)
ees1499ep1	314.8/31.9(×9.9)	358.8/75.2(×4.8)	559.2/280.7(×2.0)	919.7/353.8(×2.6)

The original grouping of the parameter sets in size, cost, and speed does not apply to the presented hardware/software codesign. For example, the parameter set `ees1499ep1` (grouped to speed) is slower than the parameter set `ees1087ep2` (grouped to size).

The design achieves speedup factors of up to 2.5 for the encryption and up to 3.3 for the decryption. This is a considerable improvement because the ARM Cortex-A53 is already a powerful processor running at a higher clock frequency than the hardware accelerator. The ternary multiplication itself achieves a speedup factor of up to 15.7. It could be considered to develop a hardware accelerator for the computing-intensive hash operations at the polynomial sampling, `BPGM` operation, and `MGF` operation to further improve the performance.

3.3 Use Case: NewHope on RISC-V SoC Platform

This section analyzes a fast and power-optimized mapping of the NTT algorithm to a hardware circuit. It further proposes an efficient hardware/software codesign for the PQC scheme NewHope on a RISC-V platform. The methods used in this section are also applicable to other LWE-based schemes, e.g., Kyber or Dilithium.

3.3.1 Algorithmic Optimizations of NTT

As discussed in Section 2.3.2, the Cooley–Tukey method is an efficient NTT variant to realize a complexity of $\mathcal{O}(n \log_2(n))$ for polynomial multiplications. Algorithm 8 shows the standard $\text{NTT}^{CT}_{br \rightarrow no}$ algorithm. The algorithm has three nested loops. The DIT butterfly operation, which consists of a modular multiplication, addition, and subtraction, is performed in the inner loop. The following paragraphs discuss algorithmic optimizations to efficiently map this NTT algorithm to a hardware circuit. First, existing optimizations of previous works are discussed. Then, new approaches are investigated, such as an efficient

Algorithm 8: $\text{NTT}^{CT}_{br \to no}$ transform (notation adapted from [RVM+14])

Input: Coefficients a_i with $i \in [0, n)$
Result: Coefficients $\hat{a}_i$ with $i \in [0, n)$

```
1  a ← BitReversal(a)
2  for m = 2 to n by m = 2m do
3      ω_m ← ω_n^{n/m}
4      ω ← 1
5      for j = 0 to m/2 − 1 by 1 do
6          for k = 0 to n − 1 by m do
7              z_1 ← a_{k+j+m/2} · ω  mod q
8              a_{k+j+m/2} ← a_{k+j} − z_1  mod q
9              a_{k+j} ← a_{k+j} + z_1  mod q
10         end
11         ω ← ω · ω_m  mod q
12     end
13 end
```

integration of modular reduction techniques, integration of the NTT post-processing, and power optimizations.

Calculation of Twiddle factors (powers of ω_n). Most NTT software implementations precompute the Twiddle factors and store them in memory. As the generated tables for these Twiddle factors are very large for high-degree polynomials, previous works developed an approach to compute the Twiddle factors in hardware on the fly [RVM+14]. Due to the high memory access latency of large tables, an on-the-fly generation can be even faster than loading the Twiddle factors from memory.

Memory access. One of the main performance bottlenecks of the NTT is the transfer of the coefficients between the main memory and processing element. The authors in [RVM+14] observed that most LWE-based schemes have coefficients smaller than 16-bit and that two coefficients can be stored in one word of a 32-bit system to decrease the required load and store operations. After performing the butterfly operation, the intermediate results of the two coefficients must be partly swapped to prepare the computations of the next NTT layer.

Optimized Cooley–Tukey algorithm. Algorithm 9 shows the $\text{NTT}^{CT}_{br \leftarrow no}$ algorithm with the discussed optimizations: on-the-fly Twiddle factor computation, storing two coefficients in one memory word, and the swapping operation. The algorithm can be divided into three steps: input preparation, calculation of the first NTT layers, and calculation of the last NTT layer.

In the first step (Line 1), the coefficients of polynomial a are stored in a bit-reversed order in the main memory. In this step, it is considered that two coefficients are stored in a single word.

Algorithm 9: $\mathrm{NTT}^{CT}_{br\rightarrow no}$ transform with optimized memory access (notation adapted from [RVM+14])

Input: Coefficients a_i with $i \in [0, n)$, precalculated values of $\omega_n^{n/m}$ ($\omega_n^{-n/m}$ for INVNTT)
Result: Coefficients $\hat{a}_i$ with $i \in [0, n)$

$a \leftarrow \textsc{BitReversal}(a)$
for $m = 2$ **to** $n/2$ **by** $m = 2m$ **do**
  $\omega_m \leftarrow \omega_n^{n/m}$ or $\omega_n^{-n/m}$ for INVNTT
  $\omega \leftarrow \omega_n^{n/(2m)}$ or 1 for INVNTT
  for $j = 0$ **to** $m/2 - 1$ **by** 1 **do**
    for $k = 0$ **to** $n/2 - 1$ **by** m **do**
      $a_{k+j+m/2}, a_{k+j} \leftarrow \mathrm{MEM}_{k+j}$ // Load first coefficient pair
      $H_1, L_1 \leftarrow a_{k+j+m/2}, a_{k+j}$
      $a_{k+j+3m/2}, a_{k+m+j} \leftarrow \mathrm{MEM}_{k+j+m/2}$ // Load second coefficient pair
      $H_2, L_2 \leftarrow a_{k+j+3m/2}, a_{k+m+j}$
      $H_1, H_2 \leftarrow (H_1 \cdot \omega) \mod q, \quad (H_2 \cdot \omega) \mod q$
      $a_{k+j+m/2}, a_{k+j} \leftarrow (L_1 - H_1) \mod q, \quad (L_1 + H_1) \mod q$
      $a_{k+j+3m/2}, a_{k+m+j} \leftarrow (L_2 - H_2) \mod q, \quad (L_2 + H_2) \mod q$
      $\mathrm{MEM}_{k+j} \leftarrow a_{k+j+m}, a_{k+j}$ // Store and swap coefficients
      $\mathrm{MEM}_{k+j+m/2} \leftarrow a_{k+j+3m/2}, a_{k+j+m/2}$ // Store and swap coefficients
    end
    $\omega \leftarrow (\omega \cdot \omega_m) \mod q$
  end
end
$m \leftarrow n$ // Prepare last NTT layer
$\omega_m \leftarrow \omega_n^{n/m}$ or $\omega_n^{-n/m}$ for INVNTT
$\omega \leftarrow \omega_n^{n/(2m)}$ or 1 for INVNTT
for $j = 0$ **to** $m/2 - 1$ **by** 1 **do**
  $a_{j+m/2}, a_j \leftarrow \mathrm{MEM}_j$ // Load coefficient pair
  $H_1, L_1 \leftarrow a_{j+m/2}, a_j$
  $H_1 \leftarrow (H_1 \cdot \omega) \mod q$
  $a_{j+m/2}, a_j \leftarrow (L_1 - H_1) \mod q, \quad (L_1 + H_1) \mod q$
  $\mathrm{MEM}_j \leftarrow a_{j+m/2}, a_j$ // Store without swap
  $\omega \leftarrow (\omega \cdot \omega_m) \mod q$
end

Algorithm 10: Barrett reduction

Input: a, k, $m = \lfloor 2^k/q \rfloor$
Result: $a \mod q$
1 $u = (a \cdot m) \gg k$
2 $u = u \cdot q$
3 $a = a - u$
4 **if** $a \geq q$ **then**
5 $\quad a = a - q$
6 **end**

Algorithm 11: Montgomery reduction

Input: a, $R = 2^n$, $q' = -q^{-1} \mod R$
Result: $a \cdot R^{-1} \mod q$
1 $u = ((a \mod R) \cdot q') \mod R$
2 $a = a + (u \cdot q)$
3 $a = a/R$
4 **if** $a \geq q$ **then**
5 $\quad a = a - q$
6 **end**

In the second step (Lines 2–19), the first $log_2(n) - 1$ NTT layers are calculated. The Twiddle factor ω is initialized in Line 4 and always updated during runtime by a modular multiplication with ω_m in Line 17. The value of ω_m depends on the current NTT layer indicated by the variable m. In practice, the values for ω_m still need to be precomputed. However, only one entry for each layer is required, resulting in $log_2(n)$ precomputations. The variable ω is initialized with the square root of ω_m in Line 4 to implicitly realize the multiplications by powers of $\gamma = \gamma_n$ for the preprocessing discussed in Section 2.3.2. The same precomputations as for ω_m can be used because $\omega_m^{1/2}$ and ω_m from the previous layer have the same value. Only for the first layer ($m = 2$), the value $\omega_m^{1/2}$ must be computed separately.

The most important part of the algorithm is the butterfly operation, described in the inner loop (Lines 7–15). In the beginning, two coefficient pairs are loaded from the main memory locations MEM_{k+j} and $\text{MEM}_{k+j+m/2}$ and are assigned to two temporary variables, each consisting of a lower halfword L_1/L_2 and a higher halfword H_1/H_2 (Lines 7–10). Then, two butterfly operations are performed (Lines 11–13). Finally, the result is swapped and stored in the correct memory location (Lines 14–15).

The third and last step is the computation of the last NTT layer (Lines 20–30). The operations of the last layer are similar to the operations of the other layers. The main difference is that no swapping operation is required.

Modular reduction. Modular reductions are complex operations that usually require hardware dividers. But the use of a constant modulus in the NTT algorithm allows the integration of efficient reduction techniques. The Barrett reduction [Bar86] (Algorithm 10) and Montgomery reduction [Mon85] (Algorithm 11) are common methods to perform the reductions of finite field arithmetic.

For the Barrett reduction, the modulus q and the input range of a determine the selection of the parameters k and m. Due to approximations within the algorithm, an error $e = 1/q - m/2^k$ (a floating-point number) that depends on these parameters exists. The algorithm works correctly as long as the input is smaller than $\lfloor 1/e \rfloor$. The Barrett algorithm performs exceptionally well for relatively small input ranges, i.e., for modular

additions at LWE-based PKE/KEM schemes.

The Montgomery reduction is more suitable than the Barrett reduction for larger input ranges [BGV93]. Therefore, it is frequently used for the prime modular multiplications of lattice-based schemes. At least one of the input operands (or the product itself) should be in Montgomery representation, i.e., the input of the Montgomery algorithm should be multiplied with R. The output of the algorithm is scaled with R^{-1}. The constant R is chosen such that the modular reduction by this constant is simple, e.g., a power of two. Further, the condition $\text{GCD}(q, R) = 1$ must hold. The valid input range of the Montgomery algorithm depends on R and is $[0, R \cdot q)$.

To illustrate the suitability of these algorithms for PQC, let us have a look at an example. Let the modulus be $q = 12289$ (e.g., as in NewHope, Falcon). For modular multiplications, the minimum input range that must be supported is $[0, (q-1)\cdot(q-1)]$. In this case, the smallest parameters for the Barrett algorithm are $k = 27$ and $m = 10921$, and the smallest suitable parameter for the Montgomery algorithm is $R = 2^{14}$. Now, the Barrett algorithm requires one 28×14-bit multiplication and one 14×14-bit multiplication in Lines 1–2 (Algorithm 10). The Montgomery algorithm requires only two 14×14-bit multiplications in Lines 1–2 (Algorithm 11). The Barrett algorithm is more suitable for smaller input ranges, e.g., for modular additions. For instance, when choosing $m = 5$ and $k = 16$, inputs of up to $\approx 16 \cdot q$ can be handled at the cost of only 18×3-bit and 4×14-bit multiplications.

The preferred method for modular multiplications is the Montgomery algorithm. For modular additions/subtractions, conditional subtractions are the most optimal solution. This method subtracts q from the input (e.g., the sum) if it is larger or equal to q. Otherwise, the input is left unchanged. In this thesis, all NTT hardware architectures use the cheaper conditional subtraction instead of the Barrett algorithm for modular additions and subtractions. It must be noted that many software implementations use the *lazy reduction* method at the sampling and ring arithmetic to improve the performance [ADPS16a]. At this method, the operands are only reduced by the modulus when the input range gets too large or when all subsequent computations are completed. Therefore, designs must be carefully implemented to avoid precision errors when changing the reduction method.

Efficient modular reduction for the Cooley–Tukey algorithm. The modular additions and subtractions of Lines 12, 13, and 27 in Algorithm 9 can be directly performed with a conditional subtraction (`csub`). The prime q can be added before the reduction is applied to avoid negative numbers at the modular subtraction. In order to allow the usage of the Montgomery reduction, the precomputed values of ω_m are stored in the Montgomery representation (Lines 3 and 21), i.e., $\omega_m = \omega_n^{n/m} \cdot R \mod q$ for the forward transform and $\omega_m = \omega_n^{-n/m} \cdot R \mod q$ for the inverse transform. The variable ω is initialized with values that are in the Montgomery domain, i.e., $\omega = \omega_n^{n/(2m)} \cdot R \mod q$ for the forward transform or $\omega = R$ for the inverse transform, to ensure that the update of the Twiddle factor in Lines 17 and 29 remains in the Montgomery domain. Thus, all modular multiplications, including the multiplications of Twiddle factors and coefficients, can be

conducted with Montgomery reductions.

Hiding NTT post-processing costs. The multiplications by the powers of $\gamma = \gamma_n$ (pre-processing) for the forward NTT can be realized by the discussed initialization of ω (see paragraph: "Optimized Cooley–Tukey algorithm"). Realizing the multiplications by n^{-1} and the inverse powers of γ_n (post-processing) is more complicated. This work shows a methodology to integrate the post-processing directly into the Cooley–Tukey algorithm without requiring large precomputations for the inverse powers of γ_n. The optimized method, which affects the last NTT layer, is illustrated in Algorithm 12. In the last round of the NTT, the coefficients a_i and $a_{i+n/2}$ are stored in the same word (loop counter is equal to the indexing variable $j = i$ in the last round). Four coefficients are always loaded (memory locations MEM_j and MEM_{j+1}) to use the same pipeline structure as for the first NTT layers. The scaling factor α_1 is used to multiply $n^{-1}\gamma_n^{-i}$ with coefficient a_i; α_2 to multiply $n^{-1}\gamma_n^{-i-1}$ with a_{i+1}; α_3 to multiply $n^{-1}\gamma_n^{-i-n/2}$ with $a_{i+n/2}$; and α_4 to multiply $n^{-1}\gamma_n^{-i-n/2-1}$ with $a_{i+n/2+1}$. The scaling factors α_1, α_2, α_3, and α_4 (Lines 4–7) are therefore initialized for the first coefficients with the precomputed values $n^{-1}R \mod q$, $n^{-1}\gamma_n^{-1}R \mod q$, $n^{-1}\gamma_n^{-n/2}R \mod q$, and $n^{-1}\gamma_n^{-n/2-1}R \mod q$, respectively. Note that the Montgomery representation is used during the initialization. The butterfly operation is performed in Lines 13–17. The post-processing at the INVNTT corresponds to multiplications of the coefficients in MEM_j and MEM_{j+1} with α_1, α_2, α_3, and α_4. In Lines 21–22, the values of α_1, α_2, α_3, and α_4 are updated. The discussed approach does not require extra clock cycles for the post-processing step as the proposed algorithm allows integrating this step into the pipeline structure.

3.3.2 NTT Hardware Accelerator

This section presents the developed NTT hardware architecture. It requires several design parameters that depend on the specific scheme. These parameters include: (i) the polynomial length n; (ii) the parameters q, R, and q' for the reduction; (iii) $\log_2(n) + 1$ values of ω_m (γ_m) or ω_m^{-1} (γ_m^{-1}) in Montgomery domain; and (iv) the precomputed scaling parameters $\beta, \gamma_1, \gamma_2, \gamma_3$, and γ_4 for the INVNTT. Further, the control signals `ntt_start` and `inverse` (for selecting NTT/INVNTT) are required.

Functionality. Figure 3.4 shows the proposed NTT architecture. It is mainly composed of a single port RAM, *Address Unit*, ω/α *Update Unit*, and arithmetic components. For the NTT/INVNTT operation, the RAM is first initialized with the coefficients of the polynomial that should be transformed. In this step, the lower and higher halfword of a 32-bit memory line are initialized with the even and odd coefficients, respectively. The *Address Unit* controls the load and store operations of the operands according to Algorithms 9 and 12. It is responsible for passing the coefficients to the arithmetic circuit and selects the current index for the precomputed ω_m/ω_m^{-1} selection.

The two coefficients from the RAM are sequentially stored in the registers L_1 (lower halfword) and H_1 (higher halfword). First, the coefficients of memory location MEM_{k+j}

Algorithm 12: Optimized last NTT layer

$m \leftarrow n$
$\omega_m \leftarrow \omega_n^{n/m} \cdot R$ or $\omega_n^{-n/m} \cdot R$ for INVNTT
$\omega \leftarrow \omega_n^{n/(2m)} \cdot R$ or R for INVNTT
$\alpha_1 \leftarrow \gamma_1$ // forward: $\gamma_1 = 0$, inverse: $\gamma_1 = n^{-1}R$
$\alpha_2 \leftarrow \gamma_2$ // forward: $\gamma_2 = 0$, inverse: $\gamma_2 = n^{-1}\gamma_n^{-1}R$
$\alpha_3 \leftarrow \gamma_3$ // forward: $\gamma_3 = 0$, inverse: $\gamma_3 = \gamma_1\gamma_n^{-n/2}$
$\alpha_4 \leftarrow \gamma_4$ // forward: $\gamma_4 = 0$, inverse: $\gamma_4 = \gamma_2\gamma_n^{-n/2}$
for $j = 0$ **to** $m/2 - 1$ **by** *2* **do**
    $a_{j+m/2}, a_j \leftarrow \text{MEM}_j$ // Load first coefficient pair
    $H_1, L_1 \leftarrow a_{j+m/2}, a_j$
    $a_{j+m/2+1}, a_{j+1} \leftarrow \text{MEM}_{j+1}$ // Load second coefficient pair
    $H_2, L_2 \leftarrow a_{j+m/2+1}, a_{j+1}$
    $H_1 \leftarrow \texttt{montgomery}(H_1 \cdot \omega)$
    $\omega \leftarrow \texttt{montgomery}(\omega \cdot \omega_m)$
    $H_2 \leftarrow \texttt{montgomery}(H_2 \cdot \omega)$
    $a_{j+m/2}, a_j = \texttt{csub}(L_1 - H_1 + q), \texttt{csub}(L_1 + H_1)$
    $a_{j+m/2+1}, a_{j+1} = \texttt{csub}(L_2 - H_2 + q), \texttt{csub}(L_2 + H_2)$
    if INVNTT **then**
        $a_{j+m/2}, a_j \leftarrow \texttt{montgomery}(\alpha_3 \cdot a_{j+m/2}), \texttt{montgomery}(\alpha_1 \cdot a_j)$
        $a_{j+m/2+1}, a_{j+1} \leftarrow \texttt{montgomery}(\alpha_4 \cdot a_{j+m/2+1}), \texttt{montgomery}(\alpha_2 \cdot a_{j+1})$
        $\alpha_1, \alpha_2 \leftarrow \texttt{montgomery}(\alpha_1 \cdot \beta), \texttt{montgomery}(\alpha_2 \cdot \beta)$ // $\beta = \omega_n^{-1}R$
        $\alpha_3, \alpha_4 \leftarrow \texttt{montgomery}(\alpha_3 \cdot \beta), \texttt{montgomery}(\alpha_4 \cdot \beta)$ // $\beta = \omega_n^{-1}R$
    end
    $\text{MEM}_j \leftarrow a_{j+m/2}, a_j$ // Store without swap
    $\text{MEM}_{j+1} \leftarrow a_{j+m/2+1}, a_{j+1}$ // Store without swap
    $\omega \leftarrow \texttt{montgomery}(\omega \cdot \omega_m)$
end

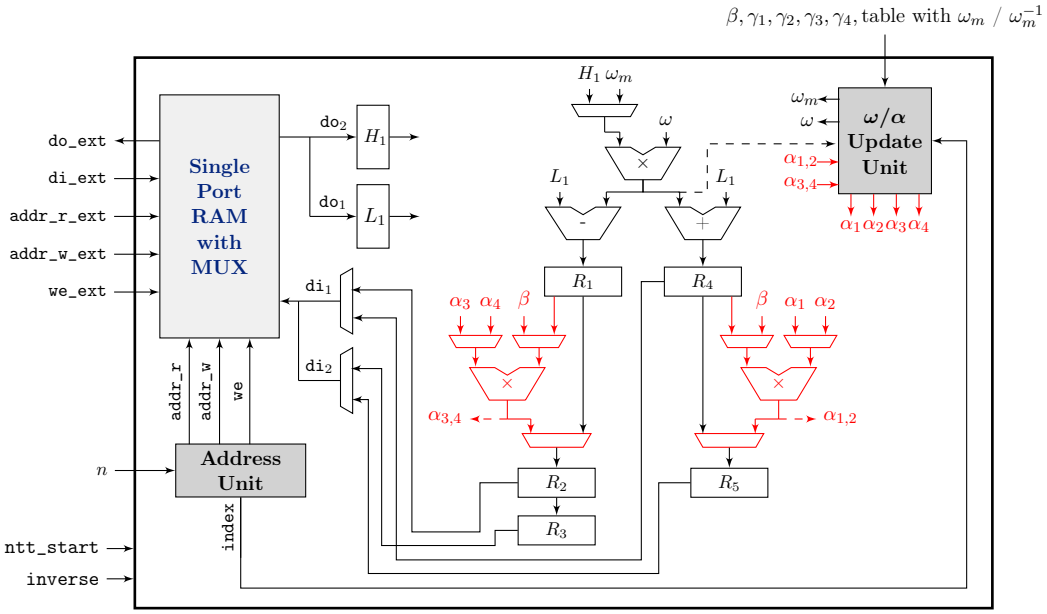


Figure 3.4: NTT architecture (parts for post-processing in red).

are loaded and processed (modular multiplication, addition, and subtraction operations are performed). One clock cycle later, the memory location $\text{MEM}_{k+j+m/2}$ is loaded and processed. The first $\log_2(n)-1$ rounds require a swap of the coefficients (see Lines 14–15, Algorithm 9). The registers R_1–R_5 are used to store intermediate results to allow the swapping operation. The data of R_4 and R_5 is written back to the memory location MEM_{k+j} during the first write cycle and the data of R_2 and R_3 to $\text{MEM}_{k+j+m/2}$ during the second one. For the last round, no swap is required, and the results of R_2 and R_5 are written in the first and second write cycle into MEM_j and MEM_{j+1}, respectively.

Additional components that are required for the INVNTT are highlighted in red in Figure 3.4. This includes multipliers for realizing Lines 19–22 of Algorithm 12.

The ω/α *Update Unit* ensures that the variables ω, α_1, α_2, α_3, and α_4 are initialized and updated. For the update $\omega = \omega \cdot \omega_m$, the multiplier of the butterfly operation upper multiplier) is reused. In the last NTT layer, each loop iteration requires two multiplications of the coefficients with ω and another two multiplications for the updates of ω. These four multiplications fit exactly into the four cycles that are required for the memory access of one loop iteration. Also, the lower two multipliers are reused for updating purposes. In each loop iteration, four multiplications with the coefficients and four multiplications for the update of the scaling factors α_1, α_2, α_3, and α_4 are realized. The select signals of the multiplexers ensure that the right operands are fed into the multipliers.

3.3.3 NTT Power Optimizations

The large number of arithmetic operations required for the NTT leads to a large power dissipation. To reduce the dynamic power consumption of the NTT, in this thesis, the application of two methods is proposed: clock gating and operand isolation.

Clock gating. Due to the high fan-out, the clock signal usually contributes to a large extent to the overall dynamic power consumption. Clock gating is an efficient method to reduce this consumption. The aim of clock gating is to turn on the clock signal only for functional units that must be active. All inactive registers have then only a static power consumption due to leakage currents. During the clock gating optimization process, registers with a common enable signal are grouped together. The original enable logic is then replaced by a common gated clock cell. Special clock gating cells are used for this purpose to avoid glitches on the gated clock signal.

Operand isolation. The second technique used in this thesis to reduce the switching activity of the circuit is operand isolation [MBC+08]. It blocks the signal propagation when it is not required for the functionality. This avoids unnecessary toggles and leads to a lower dynamic power consumption. The signal propagation is usually blocked with latches, AND gates, or multiplexers.

Operand isolation is particularly important to reduce switching activities in the arithmetic components of the proposed design (modular adder, subtractor, multiplier). The results of the first multiplier ($H_1 \cdot \omega$), subtractor, and adder in Figure 3.4 are required in only two out of four clock cycles. As a result, the input of the registers H_1 and L_1 can be frozen during all write cycles. At the write operation, the memory usually either outputs the value that is currently written or zero. Therefore, the old values of H_1 and L_1 must be preserved with a clock gating cell to avoid unnecessary toggles, as illustrated in Figure 3.5. The upper multiplier is reused to update the value of ω during the idle cycles. But the result of the multiplier for this update is not required for the subsequent adder or subtractor circuits. In order to block the signal propagation, if not required, a latch is added to the circuit, as shown in the right part of Figure 3.5.

The multipliers used for the inverse transform are always updated with new values from R_1 and R_4 when no update of the scaling parameters is performed. As the output of these multipliers is only required at the last layer of the inverse transform, the input can be set to a constant value in all other rounds, as shown in Figure 3.6. Note that β is a constant value, while R_1 and R_4 change their value. Thus, the default input should be β. To reduce the switching activity, the select signals of the multiplexers should only change if required. Putting additional conditions on the select signals requires further logic but significantly reduces the switching activity.

Evaluation of power savings. The design was synthesized with the UMC 65 nm ASIC technology to evaluate the power savings. A low leakage library with a high threshold voltage was chosen to achieve a low-power design. For the measurements, a moderate frequency of 25 MHz, a nominal supply voltage of 1.2 V, and a temperature of 25°C were

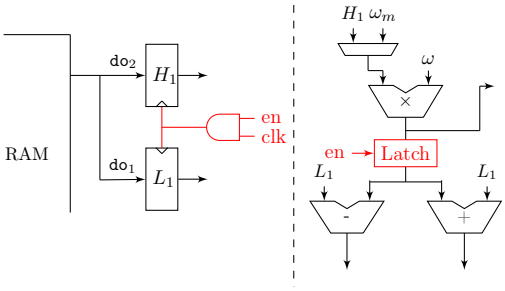


Figure 3.5: Operand isolation at input registers and after first multiplier.

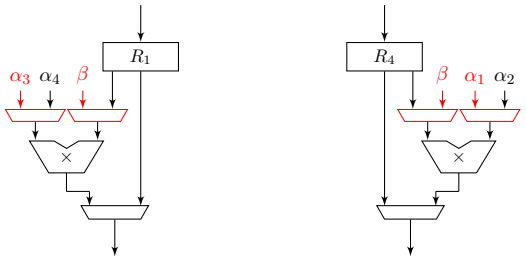


Figure 3.6: Operand isolation at multipliers for INVNTT. Set red input as default and avoid unnecessary switching.

Table 3.4: NTT power results (ASIC 65 nm).

Variant	n	Mode	P_{static} (nW)	P_{dyn} (nW)	P_{total} (nW)
Non-Optimized	256	NTT	5 634	2 701 784	2 707 417
		INVNTT	5 633	2 702 856	2 708 489
	512	NTT	5 634	2 612 861	2 618 495
		INVNTT	5 634	2 631 534	2 637 168
	1024	NTT	5 631	2 663 700	2 669 331
		INVNTT	5 631	2 684 750	2 690 381
Clock Gating and Operand Isolation	256	NTT	5 589	1 289 621	1 295 210
		INVNTT	5 589	1 452 673	1 458 262
	512	NTT	5 589	1 365 274	1 370 863
		INVNTT	5 589	1 430 824	1 436 412
	1024	NTT	5 588	1 360 870	1 366 458
		INVNTT	5 588	1 423 302	1 428 890

considered. The dynamic post-synthesis power consumption using gate-level switching activity files was determined to obtain a reasonable accuracy.

Table 3.4 summarizes the power consumption and Table 3.5 the NTT area for commonly used lattice-based cryptography parameter sets. The first test set has the parameters $(n = 256, q = 7681, \omega_n = 3844)$, the second $(n = 512, q = 12289, \omega_n = 3)$, and the third $(n = 1024, q = 12289, \omega_n = 49)$.

The static power consumption does not depend on the NTT parameters as the same architecture is used for all test sets. The dynamic power consumption also does not significantly change for different parameters. Only the energy consumption and clock cycle count highly depend on the parameter selection—particularly on n.

The design contains in total 976 FFs, where 700 FFs are suitable for clock gating. Clock gating and operand isolation increase the cell count by 79. The overall circuit size, however, does not increase. When taking both optimization methods into consideration, the total power consumption of the NTT can be decreased with the first, second, and third parameter set by 52.16 %, 47.65 %, and 48.81 %, respectively. This reduction is

Table 3.5: NTT area results (ASIC 65 nm).

Variant	#Cells	Cell-Area (μm^2)	Net-Area (μm^2)	Tot.-Area (μm^2)
Non-Optimized	7 474	102 942	17 712	120 654
Clock Gating and Operand Isolation	7 553	101 508	17 317	118 825

achieved at nearly no cost. The power consumption of the inverse transform is larger than the one of the forward transform as the multipliers are always active during the last round of the inverse computation.

Cycle count and storage results. The implementation requires $n \cdot \lceil \log_2(q_{max}) \rceil$ bits for storing the input/output coefficients and only $(2 \cdot \log_2(n)+1) \cdot \lceil \log_2(q_{max}) \rceil$ bits for storing the powers of ω_m, ω_m^{-1}, and γ_m, where $\lceil \log_2(q_{max}) \rceil$ was set to 16 in the experiments.

When a single port RAM and one clock cycle for each memory access are considered, the amount of required clock cycles for the NTT and INVNTT is equal to $n \cdot \log_2(n)$ (plus 8 cycles latency). In contrast to previous hardware accelerators of the Cooley–Tukey $\text{NTT}^{CT}_{br \rightarrow no}$, no additional clock cycles for the post-processing are required as it is integrated into the main algorithm. This saves at least $2n$ clock cycles. A dual-port RAM is more costly and has a higher area consumption but might be preferable for certain applications focusing on high performance.

3.3.4 NewHope System Design for RISC-V

This section presents the loosely coupled coprocessors and hardware/software codesign of NewHope.

RISC-V platform. The CV32E40P core is integrated into the PULPino microcontroller platform discussed in Section 2.5. The overall architecture of the hardware/software codesign is shown in Figure 3.7. The RISC-V core includes the following components: prefetch buffer, instruction decoder, General-Purpose Register (GPR), optional Floating-Point Register (FPR), Control and Status Register (CSR), Arithmetic Logic Unit (ALU), multiplication unit, optional Floating-Point Unit (FPU), and Load-Store Unit (LSU). An AXI interface is used to connect the RISC-V core with the developed hash and NTT coprocessors. The AXI to APB bridge provides an interface for the peripherals (UART, SPI, I2C, and GPIO). The instruction and data memories are accessible from the RISC-V core and from the AXI interface. All peripherals and coprocessors are memory-mapped. The corresponding address spaces are listed in Table 3.6. The address mapping is used for the configuration of the address decoder and for the development of software drivers.

NTT coprocessor. Figure 3.8 illustrates the loosely coupled NTT coprocessor. Its main component is a wrapper containing the NTT design discussed before. The memory-

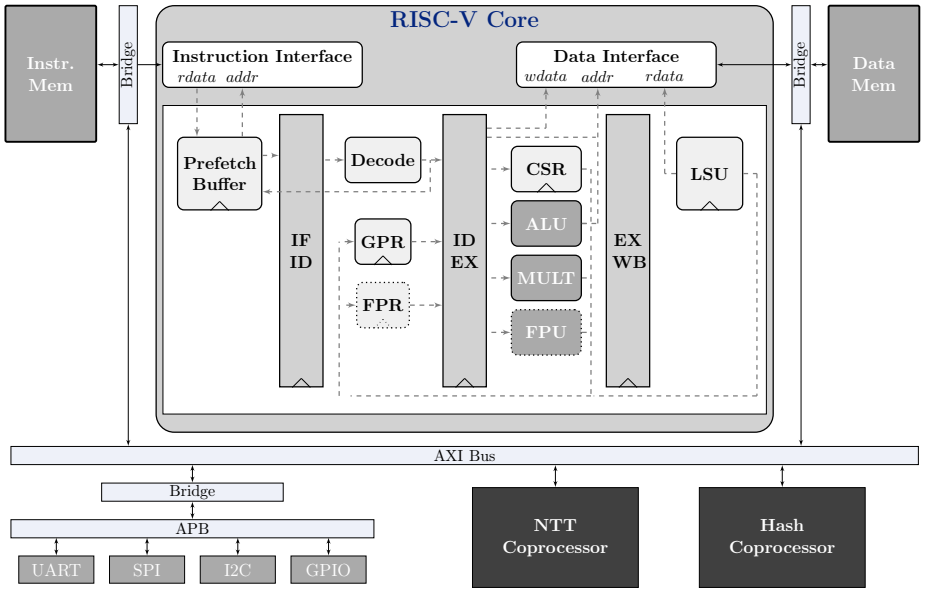


Figure 3.7: NewHope hardware/software codesign (RISC-V SoC architecture).

Table 3.6: RISC-V memory mapping of NewHope hardware/software codesign.

Address Range	Peripheral
0x00000000 – 0x0000FFFF	Instruction memory
0x00100000 – 0x0010FFFF	Data memory
0x1A100000 – 0x1A10FFFF	UART, SPI, I2C, GPIO
0x1B100000 – 0x1B10FFFF	NTT coprocessor
0x1C100000 – 0x1C10FFFF	Hash coprocessor

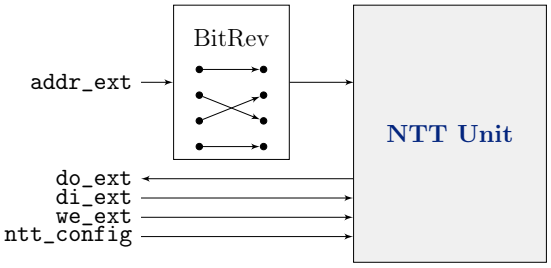


Figure 3.8: Loosely coupled NTT coprocessor.

mapped signal `ntt_config` is used for the configuration and control of the coprocessor. The NTT coprocessor has four main configuration modes: idle, write data to NTT memory, read data from NTT memory, and NTT/INVNTT operation. As discussed in Section 2.3.2, the bit-reversal operation of the NTT is a permutation of the sequence of the coefficients. For the deployed $\text{NTT}^{CT}_{br \rightarrow no}$ algorithm, this operation is required before the NTT/INVNTT starts. Hence, the coefficients are written in bit-reversed order into the NTT memory. While software implementations require large LUTs or expensive operations for the bit-reversal step, in hardware, the address signal can be simply rewired during the write operation. This ensures that the coefficients are stored in bit-reversed order in the memory.

Hash coprocessor. The hash coprocessor, illustrated in Figure 3.9, is used to generate the uniformly distributed public polynomial of NewHope (with SHAKE-128) and the binomially distributed secret/error polynomials (with SHAKE-256). The `keccak_config` signal triggers the two main submodules of Keccak: the absorption module and permutation module. It further configures the read/write access to handle the data transfer between the main processor and Keccak coprocessor. The Keccak absorb module takes the input of the hash function (e.g., a random seed) and the memory-mapped Keccak parameters `rate` and `input_length`. During the absorption phase, the input is transformed into a state with 25×64 bits. The Keccak permutation module, which is based on the implementation in [Hsi12], transforms the state according to the Keccak `f-1600` function (see Section 2.4). After the permutation phase, the output is ready and stored in the state register. The output hash length of a single run depends on the rate and is either 1344-bit for SHAKE-128 or 1088-bit for SHAKE-256. If more randomness is required, the state can be permuted again using the `f-1600` function.

3.3.5 Experimental Results of NewHope System Design

To evaluate the performance of the proposed NewHope design, the FPGA Xilinx Zynq-7000 (Zedboard) is used. The software is compiled with the RISC-V PULP toolchain[1] (Version 7.1.1) with compiler flag `-O3` (optimization for speed). The reference imple-

[1] https://github.com/pulp-platform/pulp-riscv-gnu-toolchain (Last accessed 1st Nov. 2021).

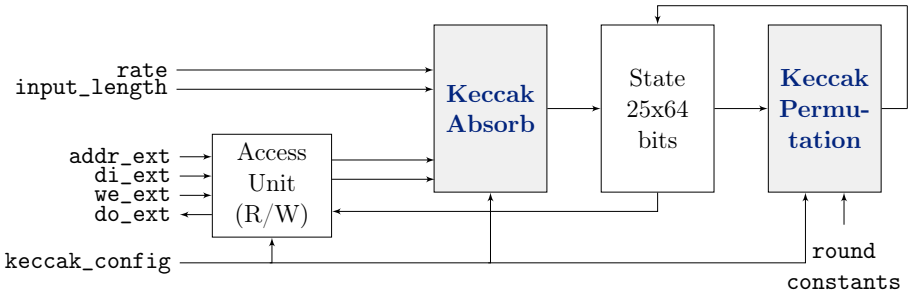


Figure 3.9: Loosely coupled hash coprocessor.

Table 3.7: Cycle count of NewHope-1024 hardware/software codesign.

	GenA	Sample	NTT	INVNTT	KeyGen	Encaps	Decaps
SW CPA Cortex-M4 [AJS16] [a)]	263 089	111 794	86 769	97 340	781 518	1 140 594	174 798
SW CCA Cortex-M4 [KRSS19]	–	–	–	–	1 219 908	1 903 231	1 927 505
SW CCA Cortex-M4 [ABCG20]	–	–	68 131	51 231	1 157 222	1 674 899	1 587 107
SW CPA (this work)	544 299	559 897	238 233	252 535	2 312 751	3 291 016	417 543
SW CCA (this work)					2 767 270	4 282 504	4 239 534
HW/SW CPA (this work)	48 730	72 840	24 119	24 119	361 927	591 779	168 505
HW/SW CCA (this work)					784 734	1 534 879	1 229 142

[a)] ChaCha20 instead of SHAKE primitives are used to achieve a higher speedup.

Table 3.8: Resource utilization of NewHope-1024 hardware/software codesign.

Complete Design				
	LUT	**FF**	**DSP**	**BRAM**
Extended PULPino	27 801	15 189	15	33

Single Components and Accelerators				
	LUT	**FF**	**DSP**	**BRAM**
RISC-V core	6 062	2 210	6	0
Peripherals+Memory	5 373	5 193	0	32
NTT coprocessor	647	416	9	1
Keccak coprocessor	11 049	4 097	0	0

mentation is based on the C code of the NewHope NIST submission [AAB+19a]. The largest parameter set with the highest security level (NewHope-1024) is selected for the evaluation.

The clock cycle measurements in Table 3.7 show a significant cycle count reduction when the loosely coupled coprocessors are used. This validates the effectiveness of the chosen approach. The comparison to related works is not straightforward as RISC-V software implementations are still lacking. Although the commercial ARM Cortex-M4 has a more advanced instruction set, it is due to its high relevance for embedded applications a good baseline for comparisons. The plain software implementation of this work is significantly slower than the assembly-optimized ARM Cortex-M4 implementations in [AJS16, KRSS19, ABCG20]. However, the achieved cycle count with the coprocessors is clearly better than the cycle count of these implementations. Even the use of ChaCha20 in [AJS16] to accelerate the randomness generation does not lead to a lower cycle count compared to the approach of this work. The cycle counts for the CCA versions of NewHope are always larger than those of the CPA versions. In particular, the decapsulation is slower as it requires a re-encryption step to prevent CCA attacks.

Writing the coefficients of a polynomial for the NTT from the RISC-V core to the coprocessor's memory and reading the transformed coefficients requires in total 13 858 clock cycles. The NTT accelerator is called two times in KeyGen (27 716 cycles for the transfer), three times in Encaps (41 574 cycles for the transfer), and one time in the CPA version of Decaps (13 858 cycles for the transfer). Also, at the sampling of the uniform polynomial (GenA) and the sampling of the secret and error polynomials (Sample), the bus communication highly affects the cycle count costs. Pure hardware implementations, e.g., the implementation in [ZYC+20], can avoid such costs and are therefore faster than the proposed hardware/software codesign. But the design in this work outperforms existing microcontroller implementations and at the same time offers high flexibility.

The achieved performance comes at the cost of an increased area. Table 3.8 summarizes the resource utilization of the complete system. The NTT coprocessor has a relatively low amount of LUTs and FFs. The DSP slices are instantiated in the constant-time modular multipliers. In contrast to the software implementations, this work avoids large LUTs for the bit-reversal, Twiddle factors, and preprocessing/postprocessing.

3.4 Summary

In this chapter, two different hardware/software codesign solutions for PQC were proposed. The results prove that a significant speedup can be achieved for both NTRU and NewHope when loosely coupled coprocessors are integrated.

For NTRU, a flexible ternary hardware multiplier was designed, which achieved speedup factors of up to 15.7 for the polynomial multiplication. To further accelerate the overall performance, a hash accelerator could be integrated into the design.

For NewHope, optimized NTT and hash coprocessors were developed. It has been demonstrated that constant-time reduction algorithms and the NTT postprocessing can

be efficiently integrated into the Cooley–Tukey NTT algorithm with on-the-fly Twiddle factor computation. Further, it was shown that clock gating and operand isolation significantly decrease the power consumption of the NTT computation.

The presented hardware/software codesigns achieve good performance results and a high flexibility. Nevertheless, a non-negligible part of the runtime is consumed by the bus communication. DMA modules can be used to improve the communication overhead between the processing system and coprocessor but also increase the design complexity and area overhead. In the next chapter, further methods are analyzed to avoid complex bus communication, decrease the area footprint, and further increase the flexibility.

4 Tightly Coupled Accelerators and Instruction Set Extensions for PQC

In this chapter, an investigation of tightly coupled hardware accelerators for PQC is presented. In contrast to the previous chapter, the accelerators developed in this chapter are not loosely coupled but deeply integrated into the processor. In order to investigate the advantages of this approach, this chapter proposes tightly coupled accelerators and system evaluations for LAC (published in [FSS20a]), NewHope/Kyber/Saber (published in [FSS20b]), and SIKE (published in [RFS20]).

4.1 Introduction of Tightly Coupled Accelerators for PQC

Integrating application-specific hardware accelerators directly into the processor core can improve important design parameters. It avoids complex bus communication and can lead to higher flexibility compared to standalone hardware or loosely coupled coprocessor solutions. For instance, copying the large polynomials of structured lattice-based cryptography from the system memory to the accelerator is very costly. DMA modules can be helpful but are also not the optimal solution due to an increase in the design complexity and the necessity of DMA setup routines. Therefore, this chapter explores how accelerators for PQC can be directly integrated into the core region. Thus, system resources can be reused, complex bus communication can be avoided, and accelerators have direct access to the processor's register files.

Related works. The rapid advances in open-source processor designs and the RISC-V ISA foster research in the areas of tightly coupled cryptographic accelerators and instruction set extensions. Different RISC-V security working groups have elaborated cryptographic extensions—particularly for symmetric ciphers [OM20]. Further, recent works proposed instruction set extensions for AES [MNP+20] and ChaCha20 [MPP21].

As discussed in Chapter 3, previous hardware designs for PQC rather focused on standalone hardware solutions and partly on loosely coupled coprocessors. Only a few works have investigated tightly coupled PQC accelerators. The first tightly coupled accelerators for the lattice-based scheme LAC were developed by the thesis author in [FSS20a]. This publication is summarized in Section 4.2. In [AEL+20], a small tightly coupled finite field accelerator for NewHope and Kyber was developed. Concurrently, more powerful tightly coupled accelerators for NewHope, Kyber, and Saber were developed by the thesis author in [FSS20b]. The corresponding work is summarized in Section 4.3. The first tightly coupled accelerators for isogeny-based cryptography were proposed by the thesis author in [RFS20]. The respective design and analysis are recapitulated in Section 4.4. RISC-V vector extensions for the code-based scheme Classic McEliece running on a relatively powerful RISC-V core were investigated in [PGZMG21].

Contribution. This chapter analyzes RISC-V ISA extensions and tailored tightly coupled hardware accelerators for PQC. In contrast to [AEL+20], all bottlenecks are considered, and more powerful approaches are evaluated.

The contributions of this chapter can be summarized as follows:

- System design of LAC on a RISC-V platform with tightly coupled accelerators and ISA extensions;
- Enhancement of the ternary polynomial multiplier of Section 3.2.2 to support different convolution modes and investigation of accelerators for the error correction used in LAC;
- System design of NewHope, Kyber, and Saber on a RISC-V platform with tightly coupled accelerators and ISA extensions;

- Development of tightly coupled accelerators for NTT multiplications, Karatsuba/ Toom–Cook multiplications, and polynomial sampling with optimized memory access strategy and reuse of system resources;

- System design of SIKE on a RISC-V platform with tightly coupled accelerators for large finite field arithmetic.

4.2 Instruction Set Extensions for the PQC Scheme LAC

LAC is based on the RLWE problem and has high similarities with NewHope. The particularity of LAC is the integration of a powerful error correction into the protocol. It uses Bose–Chaudhuri–Hocquenghem (BCH) codes to decrease the protocol's intrinsic failure rate in order to allow a reduction of the coefficient sizes and, consequently, key/ciphertext sizes. LAC is the only LWE-based scheme that has ternary secret and error polynomials and requires, just like NTRU, ternary polynomial multiplications. But in contrast to NTRU, it uses the cyclotomic polynomial $\phi = x^n + 1$ instead of $x^n - 1$. The next sections discuss how to increase the flexibility of ternary polynomial multipliers and how to accelerate the error correction of LAC.

4.2.1 Extension of Ternary Polynomial Multiplication Accelerator

In this section, the ternary hardware multiplier presented in Section 3.2.2 is extended to support both positive and negative wrapped convolutions. The hardware architecture of this multiplier is shown in Figure 4.1. The components highlighted in red are added to support the convolution for $\phi = x^n + 1$. The multiplexers forward a_i or the negation of it. The select signal $\texttt{sel}_i = 0$ for positive wrapped convolutions ($\texttt{conv\_n} = 0$). For negative wrapped convolutions ($\texttt{conv\_n} = 1$), $\texttt{sel}_i = 1$ if $i > n - 1 - cntr$ and $\texttt{sel}_i = 0$ otherwise, where the value of $cntr$ is equal to the current clock cycle (from 0 to $n - 1$). After n clock cycles, the coefficients of the result are ready to be forwarded from the registers $c_0, \ldots, c_{n-1}$ to the output c.

Example 4 (Extended Ternary Multilier)

Let $a = a_0 + a_1x + a_2x^2$ and $b = b_0 + b_1x + b_2x^2$ be two ring elements in $\mathbb{Z}_q/\langle\phi(x)\rangle$ with $n = 3$. Let $cntr$ be equal to the current clock cycle and let $k = n - 1 - cntr$ be an auxiliary variable used for the select signals of the multiplexers. Table 4.1 illustrates the register content of the ternary multiplier, which contains after three steps (clock cycles) the result $c = c_0 + c_1x + c_2x^2 = ab \mod x^3 - 1$. In this case, the multiplexers MUX0–MUX2 always select the positive serialized coefficient a_i. Table 4.2 shows the register content for $\phi(x) = x^3 + 1$. Now, it holds $\texttt{sel}_i = 1$ if $i > k$. When $\texttt{sel}_i = 1$, the i-th MUX forwards the negative coefficient $-a_i$ instead of the positive one. This realizes the negative wrapping.

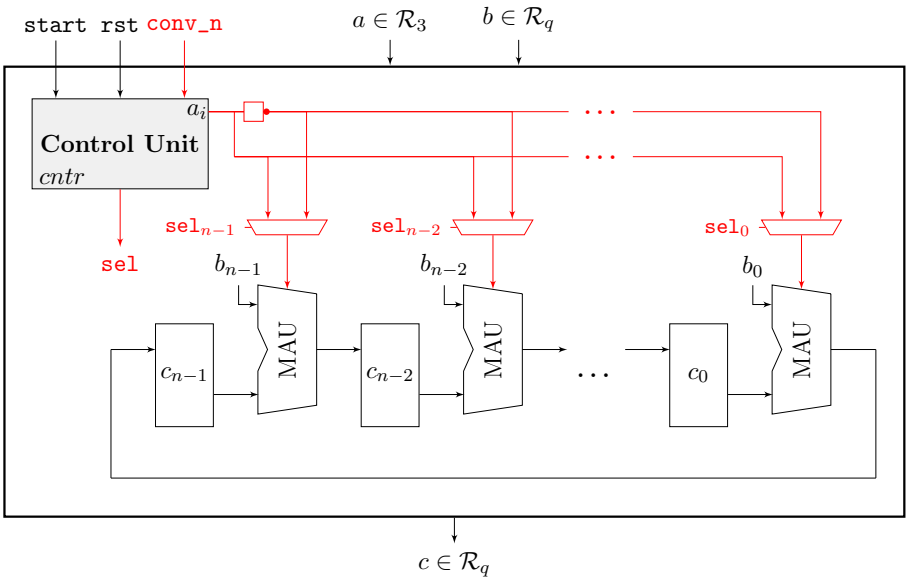


Figure 4.1: Extended ternary multiplication accelerator.

Table 4.1: Example ternary multiplier positive wrapped convolution (`conv_n` = 0).

cntr	*k*	MUX2|MUX1|MUX0	c_2	c_1	c_0
0	2	$+a_0 \quad +a_0 \quad +a_0$	a_0b_0	a_0b_2	a_0b_1
1	1	$+a_1 \quad +a_1 \quad +a_1$	$a_0b_1 + a_1b_0$	$a_0b_0 + a_1b_2$	$a_0b_2 + a_1b_1$
2	0	$+a_2 \quad +a_2 \quad +a_2$	$a_0b_2 + a_1b_1 + a_2b_0$	$a_0b_1 + a_1b_0 + a_2b_2$	$a_0b_0 + a_1b_2 + a_2b_1$

Table 4.2: Example ternary multiplier negative wrapped convolution (`conv_n` = 1).

cntr	*k*	MUX2|MUX1|MUX0	c_2	c_1	c_0
0	2	$+a_0 \quad +a_0 \quad +a_0$	a_0b_0	a_0b_2	a_0b_1
1	1	$-a_1 \quad +a_1 \quad +a_1$	$a_0b_1 + a_1b_0$	$a_0b_0 - a_1b_2$	$a_0b_2 + a_1b_1$
2	0	$-a_2 \quad -a_2 \quad +a_2$	$a_0b_2 + a_1b_1 + a_2b_0$	$a_0b_1 + a_1b_0 - a_2b_2$	$a_0b_0 - a_1b_2 - a_2b_1$

Software-based polynomial splitting. The proposed multiplication architecture scales directly with the polynomial length n, where $n = 512$ for LAC-128 and $n = 1024$ for LAC-192/LAC-256. In order to use the same hardware architecture for all three security levels, this work proposes to perform a software-based split when $n = 1024$ is set.

The software split is described in the following paragraphs. The length-m polynomials a and b can be split into length-$m/2$ polynomials: (i) lower part (a^l, b^l); and (ii) higher part (a^h, b^h). The multiplication with these smaller polynomials can be written as

$$c = ab = a^l b^l + (a^l b^h + a^h b^l)x^{m/2} + a^h b^h x^m \quad . \tag{4.1}$$

Note that Karatsuba's algorithm reduces the four polynomial multiplications in Equation 4.1 to three. However, Karatsuba's approach requires the multiplication of $(a^h + a^l) \cdot (b^h + b^l)$ (see Section 2.3.3, Equation 2.7). The addition of $a^h + a^l$ performed in $\mathbb{Z}_q/\langle\phi(x)\rangle$ leads to a polynomial that is not ternary anymore (although a^h, a^l are initially ternary). Therefore, Karatsuba's algorithm requires general multiplications $\mathcal{R}_q \times \mathcal{R}_q$, preventing the applicability of the ternary multiplier. Enhancing the ternary multiplier to support general multiplications implies an increase in the design complexity. Therefore, Karatsuba's tweak is not applied in this case.

A length-512 ternary multiplier presents a good trade-off between performance and area. It directly supports LAC-128 and requires a split for LAC-192/LAC-256. Let MulTernary512 be the ternary hardware multiplication operation supporting length-512 multiplications. In order to use MulTernary512 for multiplications of length-1024 polynomials, a two-level polynomial split must be performed: first a split into length-512 and then into length-256 polynomials. The reason for the second split is that MulTernary512 only supports reductions by $x^{512} \pm 1$ and not by $x^{1024} \pm 1$. Algorithm 13 shows the two-level splitting technique. It first splits the polynomials a and b into length-512 polynomials. In Lines 1–2, the shares of these polynomials are forwarded to four instances of Algorithm 14, which splits the polynomials further into length-256 polynomials in order to use MulTernary512. The four partial results of Equation 4.1 are then recombined in Lines 3–11 (Algorithm 14). When the four instances of Algorithm 14 completed the recombination, Algorithm 13 starts the recombination phase in Lines 3–12. While in Algorithm 14 the recombination is done as in Equation 4.1, Algorithm 13 already integrates the modular reduction by $x^{1024} + 1$. The polynomial reduction by $x^{1024} + 1$ is performed by wrapping coefficients larger than 1023 negatively around as done in Lines 5 and 11. The wrapping requires simple subtractions modular $q = 251$.

Algorithm 13: SplitMulHigh(a, b, c)

Input: $a, b \in \mathcal{R}_{n=1024}$

1 SplitMulLow(a^l, b^l, c^{ll}), SplitMulLow(a^h, b^h, c^{hh})
2 SplitMulLow(a^l, b^h, c^{lh}), SplitMulLow(a^h, b^l, c^{hl})
3 **for** $i \leftarrow 0$ **to** $1024 - 1$ **do**
4 $\quad c_i \leftarrow c_i^{ll}$
5 $\quad c_i \leftarrow (c_i - c_i^{hh}) \mod q$ `// wrap around`
6 **end**
7 **for** $i \leftarrow 0$ **to** $512 - 1$ **do**
8 $\quad c_{i+512} \leftarrow (c_{i+512} + c_i^{lh} + c_i^{hl}) \mod q$
9 **end**
10 **for** $i \leftarrow 512$ **to** $1024 - 1$ **do**
11 $\quad c_{i-512} \leftarrow (c_{i-512} - c_i^{lh} - c_i^{hl}) \mod q$ `// wrap around`
12 **end**

Result: $c \in \mathcal{R}_{n=1024}$

Algorithm 14: SplitMulLow(a, b, c)

Input: $a, b \in \mathcal{R}_{n=1024}$ with condition $\forall i \geq 512,\ a_i = 0, b_i = 0$ (not $\mathcal{R}_{n=512}$ as $\phi = x^{1024} + 1$ remains after splitting)

1 MulTernary512(a^l, b^l, c^{ll}), MulTernary512(a^h, b^h, c^{hh})
2 MulTernary512(a^l, b^h, c^{lh}), MulTernary512(a^h, b^l, c^{hl})
3 **for** $i \leftarrow 0$ **to** $512 - 1$ **do**
4 $\quad c_i \leftarrow c_i^{ll}$
5 **end**
6 **for** $i \leftarrow 0$ **to** $512 - 1$ **do**
7 $\quad c_{i+256} \leftarrow (c_{i+256} + c_i^{lh} + c_i^{hl}) \mod q$
8 **end**
9 **for** $i \leftarrow 0$ **to** $512 - 1$ **do**
10 $\quad c_{i+512} \leftarrow (c_{i+512} + c_i^{hh}) \mod q$
11 **end**

Result: $c \in \mathcal{R}_{n=1024}$

4.2.2 Error Correction Accelerator

LAC uses a BCH(n=511,k=367,t=16) code for LAC-128/LAC-256 and a BCH$(511, 439, 8)$ code for LAC-192, where n is the codelength, k the maximum message length, and t the number of correctable bits. The BCH decoder can be divided into three steps: calculation of the syndromes, calculation of the error locator polynomial Λ (e.g., with Berlekamp–Massey algorithm [Ber66, Mas69]), and calculation of the roots of the error locator polynomial (e.g., with Chien search [Chi64]). In this work, the Chien search is accelerated. It is the costliest operation for the selected BCH code parameters (see Section 4.2.4).

Hardware architecture Galois field multiplier. The BCH codes used in LAC require arithmetic operations in the Galois field GF(2^m) with $m = 9$. Let α be a primitive element in GF(2^m) and the generator for the closed finite field $F_q^* = \{0, 1, \alpha, \alpha^2, \ldots, \alpha^{2^m-2}\}$. The multiplicative group F_q^* is also called power representation. To obtain the vector representation of the finite field, first x of the primitive polynomial $p(x) = 1 + x^4 + x^9$ (for $m = 9$) is substituted with α and the equation is then set to zero. This results into $\alpha^9 = 1 + \alpha^4$ (with binary additions). Using this knowledge, the vector representation can be constructed. For example, $\alpha^9 = 1 + \alpha^4 \mathrel{\widehat{=}} (100010000)$, $\alpha^{10} = \alpha^9\alpha = (1 + \alpha^4)\alpha = \alpha + \alpha^5 \mathrel{\widehat{=}} (010001000)$, and $\alpha^{11} = \alpha^2 + \alpha^6 \mathrel{\widehat{=}} (001000100)$.

The vector representation is especially suitable for finite field additions as simple XOR operations are sufficient. GF-multiplications in the vector representation are more complicated than additions. The developed GF-multiplication module of this work, which is presented in Figure 4.2, is based on the architecture in [LC04]. It has a shift-and-add structure with interleaved reduction by the primitive polynomial. This reduction is achieved by the feedback loop. For $m = 9$, the result of register c_8 is fed back to the inputs of c_0 and c_4. The AND and XOR gates are used to perform the required binary multiplications and additions, respectively. The bits a_i of the input a are directly assigned to the first input of the AND gates. The *Control Unit* selects and forwards

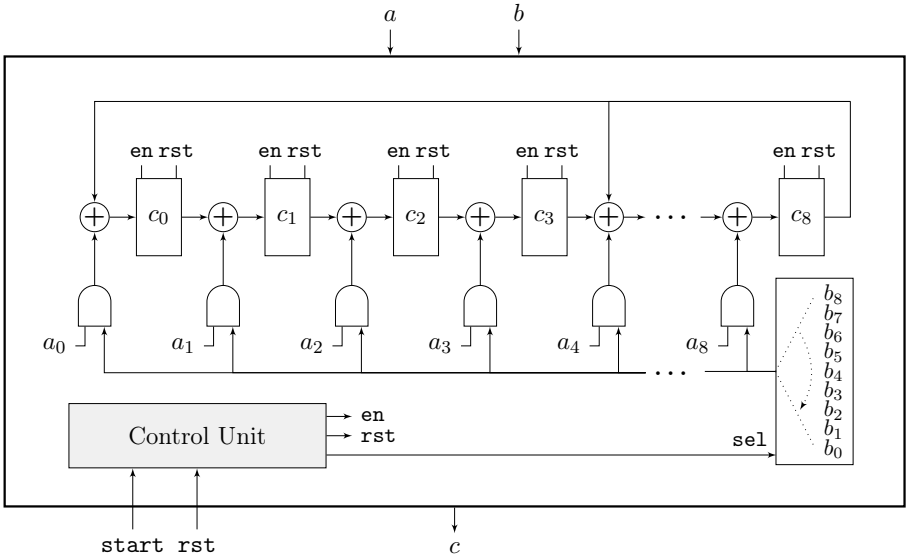


Figure 4.2: Galois field multiplier.

sequentially the bits b_i to the second input of the AND gates, beginning in the first clock cycle with the last element b_8. It further triggers the computation when `start` $= 1$ and stops the rotation after m cycles. The result of the registers is then forwarded to the output c.

Hardware architecture Chien search. The last step of the decoding process and the most time-consuming one is to find the roots of the error locator polynomial $\Lambda(x) = \lambda_0 + \lambda_1 x + \cdots + \lambda_t x^t$, where t denotes the maximum number of correctable errors, and λ_i is a finite field element. The root-finding problem can be solved using the Chien search algorithm [Chi64]. It successively substitutes x of $\Lambda(x)$ with $1, \alpha, \alpha^2, \ldots, \alpha^{n-1}$, where n is the code-length:

$$\Lambda(\alpha^i) = \lambda_0 + \lambda_1 \alpha^i + \cdots + \lambda_t \alpha^{it} \ . \tag{4.2}$$

If α^l turns out to be a root, there is an error at the location number $n - l$. As the codeword in LAC is systematic, and the message length is only 256-bit, not all powers of α must be checked, i.e., only $\Lambda(\alpha^{112})$ to $\Lambda(\alpha^{368})$ for LAC-128/LAC-256, and $\Lambda(\alpha^{184})$ to $\Lambda(\alpha^{440})$ for LAC-192.

Special circuits for multiplying a field element with the constant α^i exist [LC04]. However, for each constant, a different circuit would be required. The general Galois field multiplier is used (MUL-GF) to achieve high flexibility while keeping the area overhead low. In order to speed up the arithmetic operations of Equation 4.2, four field multiplications and accumulations are computed in parallel, as shown in Figure 4.3. Thus, the equation is split into two parts for $t = 8$ and into four parts for $t = 16$:

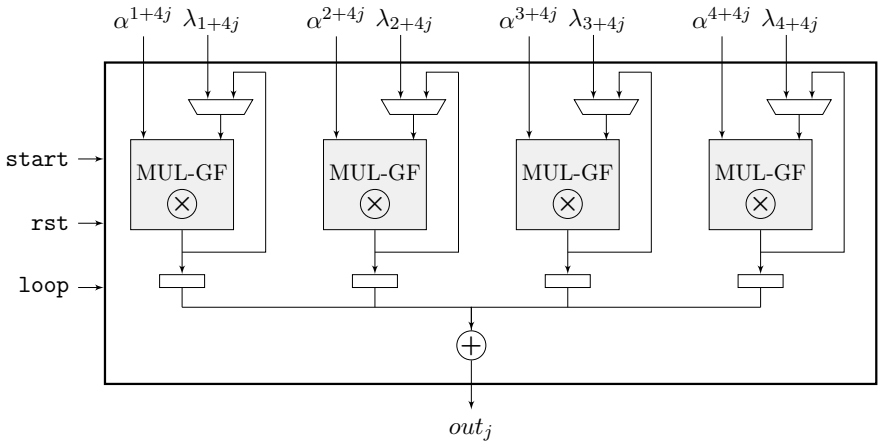


Figure 4.3: Chien search multiplier.

$$\begin{aligned}\Lambda(\alpha^i) = \lambda_0 + \sum_{j=0}^{t/4-1} \lambda_{1+4j}\alpha^{i(1+4j)} + \lambda_{2+4j}\alpha^{i(2+4j)} \\ +\lambda_{3+4j}\alpha^{i(3+4j)} + \lambda_{4+4j}\alpha^{i(4+4j)} = \lambda_0 + \sum_{j=0}^{t/4-1} out_j \ .\end{aligned} \tag{4.3}$$

A feedback loop from the output to the input is set in order to avoid an update of the multiplier input values in each i-th check ($\Lambda(\alpha^i)$). The values λ_{1+4j} to λ_{4+4j} are only loaded to the second input of the GF multipliers in the first round. In all other rounds, the result is fed back (`loop` enabled), thus increasing the performance. The first input is set to α^{1+4j} for the first multiplier until α^{4+4j} for the fourth multiplier.

4.2.3 Core Integration of LAC Accelerators

For the experiments of this chapter, the same processor and platform as in Section 3.3 are used.

The original RISC-V core is extended by the Post-Quantum ALU (PQ-ALU), as shown in Figure 4.4. The PQ-ALU is directly placed in the execution stage of the processor and includes the ternary multiplier (MUL-TER), the Chien search multiplier (MUL-CHIEN), a SHA256 accelerator, and a modular reduction accelerator (MODq). The SHA256 accelerator is based on a previous work of the author of this thesis [BFM+18] and is not further discussed here. The MODq module is based on the Barrett reduction (Algorithm 10, Section 3.3). Note that conditional subtractions could be used for modular reductions when lazy reductions are avoided. However, the Barrett multiplier is very small compared to the remaining design (see next section). Therefore, the more flexible Barrett reduction is integrated for this specific case.

Controlling the accelerators. To access and control the custom accelerators, the RISC-V ISA is extended by four PQC instructions: *pq.mul_ter*, *pq.mul_chien*, *pq.sha*256,

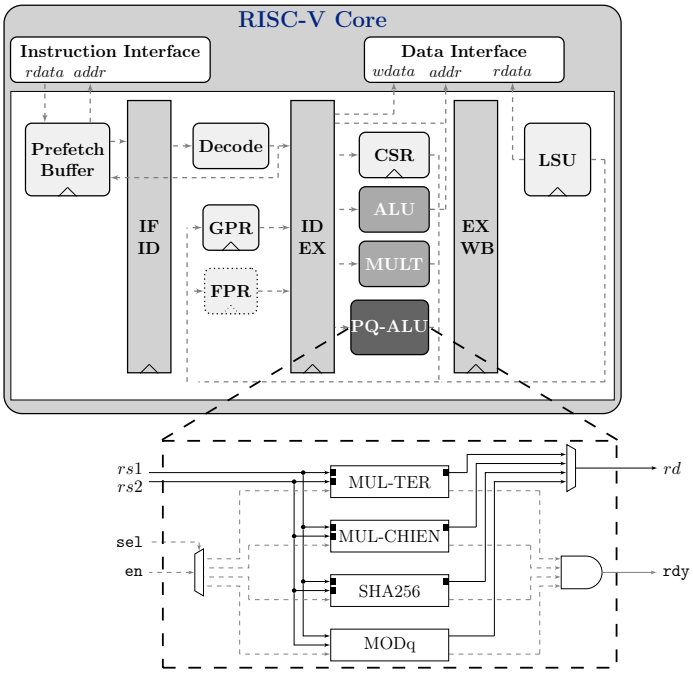


Figure 4.4: LAC hardware/software codesign.

and *pq.modq*. The developed instructions have the R-type instruction format (see Section 2.5). The instruction decoder sets the `sel` signal according to the function fields of the instruction. This activates the desired accelerator. The MUL-TER, MUL-CHIEN, and SHA256 accelerators use input/output buffers as their operands are too large to fit into two input registers $rs1/rs2$ and one output register rd.

The MUL-TER unit has three operation modes: read input coefficients, calculate the multiplication, and write output coefficients. During the read input operation, five general coefficients (8-bit each) and five ternary coefficients (2-bit each) are repetitively packed into the source registers $rs1$ and $rs2$ until all $n = 512$ coefficients are available at the accelerator. The calculate multiplication operation computes the ternary multiplication in n cycles. During the write output operation, rd is repetitively packed with four coefficients (8-bit each). The remaining bits not required for these operations are used for the control of the accelerator (e.g., the `conv_n` signal and read/write address).

The MUL-CHIEN unit has three operation modes: read four field elements for the left two multipliers, read four elements for the right two multipliers, and calculate/return the result. One field element requires 9-bit. Therefore, four elements are packed into the two source registers. The remaining bits are used for the control (e.g., the `loop` signal).

At the SHA256 unit, $rs1$ is used for the input values (8-bit each) and $rs2$ for the read/write address as well as the configuration signals (generate the hash and reset the internal state).

Table 4.3: Cycle count of LAC with tightly coupled accelerators CCA versions.

Scheme	Device	KeyGen	Encaps	Decaps	GenA	Sample	PolyMul	BCH-Dec
LAC-128 ref. [KRSS19]	ARM M4	2 266 368	3 979 851	6 303 717	–	–	–	–
LAC-192 ref. [KRSS19]	ARM M4	7 532 180	9 986 506	17 452 435	–	–	–	–
LAC-256 ref. [KRSS19]	ARM M4	7 665 769	13 533 851	21 125 257	–	–	–	–
LAC-128 ref.	RISC-V	2 981 055	4 969 238	7 897 403	159 192	190 256	2 381 843	514 280
LAC-192 ref.	RISC-V	10 162 502	13 388 952	23 126 138	287 736	165 185	9 482 261	220 181
LAC-256 ref.	RISC-V	10 515 588	18 165 040	28 220 945	287 609	344 436	9 482 263	513 687
LAC-128 opt.	RISC-V	542 814	640 237	839 132	154 746	159 134	6 390	160 295
LAC-192 opt.	RISC-V	816 635	1 086 148	1 324 014	282 264	156 320	151 354	52 142
LAC-256 opt.	RISC-V	1 086 252	1 388 366	1 759 756	282 264	291 007	151 355	160 296
NewHope-1024 opt.	RISC-V	784 734	1 534 879	1 229 142	48 730	72 840	120 510	–

4.2.4 Experimental Results of LAC System Design

This section evaluates the performance improvement for LAC when using the proposed accelerators. Three different security levels were analyzed: LAC-128 (NIST Level I), LAC-192 (NIST Level III), and LAC-256 (NIST Level V). The RISC-V system with the extended processor was synthesized and implemented using the programmable logic of the Xilinx Zynq UltraScale+ ZCU102 platform. The software code for the RISC-V core was compiled using the official RISC-V compiler from Berkeley (Version 8.2.0) and was loaded via SPI to the instruction and data memory.

Constant runtime. Employing the powerful BCH error correction improves certain characteristics of lattice-based cryptography but increases the complexity of the design. It further introduces a potential point of attack. The authors in [DTVV19] successfully mounted a timing attack on LAC. Timing attacks are particularly critical and must be prevented as they are applicable in many use cases and usually do not require expensive equipment. A constant-time version of the BCH code was proposed in [WR20] to protect against timing attacks. As the second round implementation of LAC had initially not a constant runtime, the constant-time BCH software implementation of [WR20] is used as a baseline for this work. The BCH$(511, 367, 16)$ code requires for the simulation with 16 failures 89 335 cycles (syndrome computation), 33 867 cycles (Berlekamp–Massey), and 380 748 cycles (Chien search) on the RISC-V platform. These results show that the Chien search is the most expensive step of the decoding.

Cycle count comparison. Table 4.3 summarizes the cycle count results of four different implementations: (i) the LAC reference implementation on ARM Cortex-M4 [KRSS19]; (ii) the LAC reference implementation on RISC-V (with constant-time BCH); (iii) the optimized LAC implementation on RISC-V with instruction set extensions; and (v) the optimized NewHope hardware/software codesign of Section 3.3.

Compared to the LAC reference implementation on ARM Cortex-M4, the reference implementation on RISC-V is clearly slower. When comparing the optimized design with the reference, speedup factors of 7.84 (LAC-128), 14.47 (LAC-192), and 13.44 (LAC-

Table 4.4: Resource utilization of LAC hardware/software codesign with tightly coupled accelerators.

	LUT	FF	DSP	BRAM
RISC-V core total	53 819	13 928	10	0
– Ternary multiplier	31 465	9 305	0	0
– Chien accelerator	86	158	0	0
– SHA256 accelerator	1 031	1 556	0	0
– Modular arithmetic accelerator	35	0	2	0
NTT accelerator (Sect. 3.3)	647	416	9	1
Keccak accelerator (Sect. 3.3)	11 049	4 097	0	0

256) are achieved. LAC-256 and NewHope-1024 can be categorized to the highest NIST security level. Compared to the loosely coupled NewHope implementation, the optimized LAC implementation is around 19 % slower for the whole algorithm execution. The overhead can be mainly explained by the slower SHA256 (compared to Keccak) and the additional error-correcting code.

The functions GenA and Sample are used to generate the public polynomial and secret/error polynomials, respectively. They use repetitive calls of SHA256 (LAC) and Keccak/SHAKE (NewHope). The SHA256 hardware module has a lower output bit rate compared to the Keccak implementation, but Table 4.4 shows that the required resources are also considerably lower. This is mainly due to the size of the internal state: 256-bit (SHA256) vs. 1600-bit (Keccak). Using the Keccak accelerator for LAC has been left as future work. A particularly high speed improvement was achieved for the polynomial multiplication. When the software split is used (LAC-192 and LAC-256), however, the NTT-based approach in NewHope is faster. A multiplication at NewHope requires two forward and one inverse NTT operation as well as further cycles for the coefficient-wise multiplications. While the software split decreases the performance, the amount of LUTs and registers is significantly reduced as these resources directly scale with the polynomial length. Compared to the NTT accelerator, the ternary multiplier still requires a high amount of LUTs and registers but no DSPs or BRAMs. In comparison to the reference implementation on RISC-V, the total BCH decoding time was improved by a factor of 3.21 and 4.22 for LAC-128/256 and LAC-192, respectively. Although LAC is slower as NewHope in this scenario, it should be considered that LAC has significantly lower key and ciphertext sizes, e.g., for the highest NIST security level LAC/NewHope requires $\|pk\| = 1\,056/1\,824$, $\|sk\| = 1\,024/1\,792$, and $\|ct\| = 1\,424/2\,176$ bytes.

4.3 Instruction Set Extensions for the PQC Schemes NewHope, Kyber, and Saber

The ternary multiplier for LAC is, despite its tight coupling and the polynomial splitting, still relatively large. In order to exploit the full potential of a tightly coupled approach, this section provides an exploration of smaller and more flexible solutions that do not

require large input/output buffers and that systematically make use of existing system resources. As use cases, the PQC schemes NewHope, Kyber, and Saber are analyzed. While NewHope and Kyber included the NTT as part of their specification, Saber does not directly support NTT-based ring arithmetic.

4.3.1 Optimizing the NTT for a Tight Coupling

In this section, the design rationale and optimization techniques for the development of a tightly coupled NTT accelerator are presented. The performance and size of such an NTT accelerator are highly influenced by the calculation of Twiddle factors, bit-reversal computation, and memory access strategy.

NTT algorithm selection. The polynomial length is $n = 512$ (NewHope-512), $n = 1024$ (NewHope-1024), and $n = 256$ (all Kyber instances). As the polynomial length highly affects the NTT costs, different design decisions were made for NewHope and Kyber. For NewHope, the basic $\text{NTT}^{CT}_{br \rightarrow no}$ algorithm with on-the-fly Twiddle factor computation is used to avoid large precomputed tables. For Kyber, the precomputation of the Twiddle factors is due to the smaller polynomial length less memory consuming. Therefore, precomputed tables together with the combination of $\text{NTT}^{CT}_{no \rightarrow br}$ and $\text{INVNTT}^{GS}_{br \rightarrow no}$ are used to avoid extra computational costs for the bit-reversal and post-processing steps. On-the-fly computations of the Twiddle factors with the combination of these two NTT algorithms are not straightforward. Hence, this combination is not applied for NewHope in this thesis.

Memory access strategy. The NTT can be divided into $log_2(n)$ layers, with n being the polynomial length. A naive approach requires loading and storing all n coefficients in each layer, resulting in total in $n \cdot log_2(n)$ load and store operations, respectively. In order to decrease the number of memory accesses, multiple coefficients can be stored in a single memory word, as discussed in Section 3.3. As the polynomial coefficients of NewHope and Kyber can be expressed with at most 16-bit, two coefficients are stored in a single word (32-bit). After performing the butterfly operation, the intermediate results of the coefficients are swapped to prepare them for the next layer. Note that precalculated Twiddle factors can be stored in any desired order. Such particularity enhances the traverse possibilities through the NTT structure and allows avoiding the swapping operation. Merging the NTT layers is another technique used in [GOPS13, AJS16, BKS19, ABCG20]. The goal is to keep a certain amount of coefficients as long as possible within the register file. Let l denote the total number of coefficients that can be stored within the register file. After computing the butterfly operations for the first l coefficients within the first layer, the results are not written back to the main memory. Instead of completing the first layer, the next layer is already processed. This method saves the memory accesses for the next layer. When l coefficients can be loaded into the registers, up to $log_2(l)$ layers can be merged. The exact number of mergeable layers depends on the selected NTT instance and must be determined on a case-by-case basis. Previous works loaded up to 16 coefficients into the registers to process up to four NTT

levels without reloading coefficients between the layers. In [ABCG20], for NewHope-512 $3+3+3$ layers of in total 9 layers were merged (Layer 1–3, 3–6, and 6–9, respectively). For Kyber, the authors in [ABCG20] merged $3+3+1$ layers of in total 7 layers. In this work, the complete FPR of the RISC-V core (32×32 bit) is used to store $l = 2 \times 32 = 64$ coefficients for NewHope. An address controller loads the coefficients directly from the FPR into two butterfly units such that the calculation of multiple NTT levels can be performed. Kyber requires different algorithms for the forward and inverse transforms, and thus two address controllers would be required. As this increases the complexity and area of the design, for Kyber no dedicated address controllers are developed. An analysis of the performance improvement of Kyber with address controller has been left as future work. This means that in this thesis, only the GPR was used for Kyber to store $l = 16$ coefficients in order to merge $3+3+1$ layers. Note that Kyber uses the concept of an incomplete NTT, which leads to an early abort of the NTT transformation. For this reason, the last layer of Kyber cannot be merged with the other ones [ABCG20]. For further details about the incomplete NTT, please refer to Section 5.4.1.

Figure 4.5 illustrates an example of the applied NTT layer merging and swapping techniques for the $\text{NTT}^{CT}_{br \leftarrow no}$ variant that is used for NewHope. In the small example for $n = 16$, eight coefficients can be loaded from the main memory into the registers ($l = 8$). Two butterfly operations can be processed in parallel within the units *BF0* and *BF1*. The red rectangles show the order in which the coefficient pairs are processed by the two butterfly units. In order to simplify the following description, the term distance is defined as the difference between the positions of two coefficients in an array. At the beginning, the first eight coefficients $a_0, a_8, a_4, a_{12}, a_2, a_{10}, a_6, a_{14}$ are loaded pairwise into the register file. In the first layer, the distance between the coefficients that are processed together is equal to one, i.e., they are in the same register. The coefficient pairs (a_0, a_8), (a_4, a_{12}) are processed first using the two butterfly units, and (a_2, a_{10}), (a_6, a_{14}) are processed next. Instead of storing the result back to the main memory, the second layer is already processed. In the second layer, the distance between the coefficients that are processed together is equal to two. It requires the results of the first layer for the coefficient pairs (a_0, a_4), (a_2, a_6) and (a_8,a_{12}), (a_{10},a_{14}). A swap is performed by the butterfly units, indicated in Figure 4.5 by the blue arrows, as these coefficient pairs are not next to each other, i.e., they are not within the same register. This swapping operation, which is an exchange of 16-bit values in two registers, can be performed in hardware almost for free. In the third layer, the distance between the coefficients that are processed together is four, which means that some required coefficients would still be within the register set. However, not all coefficients required for the swapping operation to prepare the content of the registers for the next layer are in the register set. Therefore, the number of layers that can be merged is $log_2(l) - 1$. In the example shown in Figure 4.5, it is two. However, the swapping technique allows always having the right coefficients in the same word. Table 4.5 illustrates the register content and the input values for the butterfly units *BF0* and *BF1* for the small example in Figure 4.5. An unoptimized version of the NTT only loads two coefficients at the same time, has only one butterfly unit, and processes layer after layer. While in this small example with $n = 16$ coefficients the unoptimized version requires $n \cdot log_2(n) = 16 \cdot 4 = 64$ memory accesses, the optimized version shown

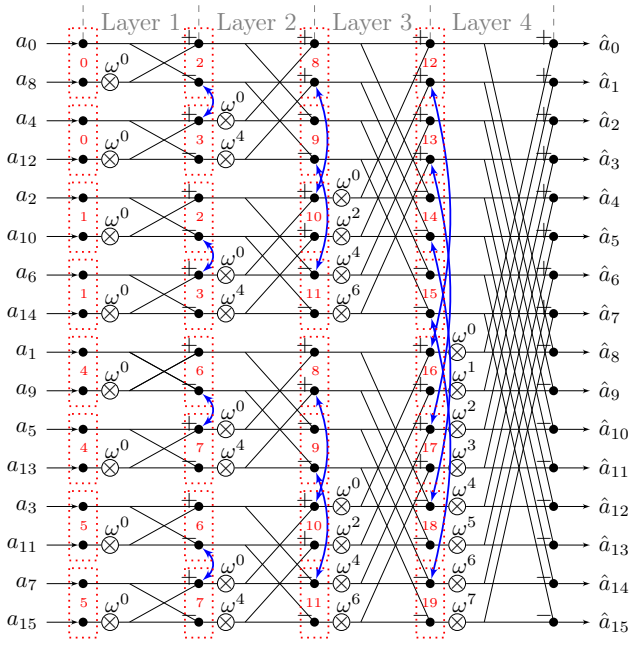


Figure 4.5: Optimized $\text{NTT}^{CT}_{br\leftarrow no}$ example with $n = 16$, $l = 8$, and two parallel butterfly units. The red boxes indicate which coefficients are stored together and in which order they are processed by the two butterfly units. The blue arrows indicate the swapping.

in Figure 4.5 ($n = 16$, $l = 8$) requires $n/2 \cdot (log_2(n) + 1 - (log_2(l) - 1)) = 24$ memory accesses.

The algorithmic modifications required for the merging technique include the manipulation of the start and end values of the outer and inner loop of Algorithm 9, Lines 2 and 6 (Section 3.3.1). For the hardware architecture, 32 registers are considered to be available from the FPR for storing $l = 64$ coefficients to merge five NTT layers. The nested loops can be split into $n/64$ parts. The outer loop, which indicates the current layer, starts for each part from 2^1 and terminates at the value 2^5, i.e., $m = \{2, 4, 8, 16, 32\}$. The inner loop iterates for the i-th part from $k = 32i$ to $k = 32i + 31$, where $i \in [0, n/64)$. Instead of loading and storing the coefficients from the main memory, the coefficients are kept within the register set and are only refreshed between the $n/64$ parts.

Table 4.5: Register content and input for the two butterfly units BF0 and BF1 for the example $n = 16$, $l = 8$.

Register content	Load coeffs.	Step 0		Step 1		Step 2		Step 3		Store coeffs./ Load coeffs.	...
		BF0	BF1	BF0	BF1	BF0	BF1	BF0	BF1		
R0	a_0, a_8	a_0, a_8				a_0, a_4				a_1, a_9	
R1	a_4, a_{12}		a_4, a_{12}					a_8, a_{12}		a_5, a_{13}	
R2	a_2, a_{10}			a_2, a_{10}			a_2, a_6			a_3, a_{11}	
R3	a_6, a_{14}				a_6, a_{14}				a_{10}, a_{14}	a_7, a_{15}	

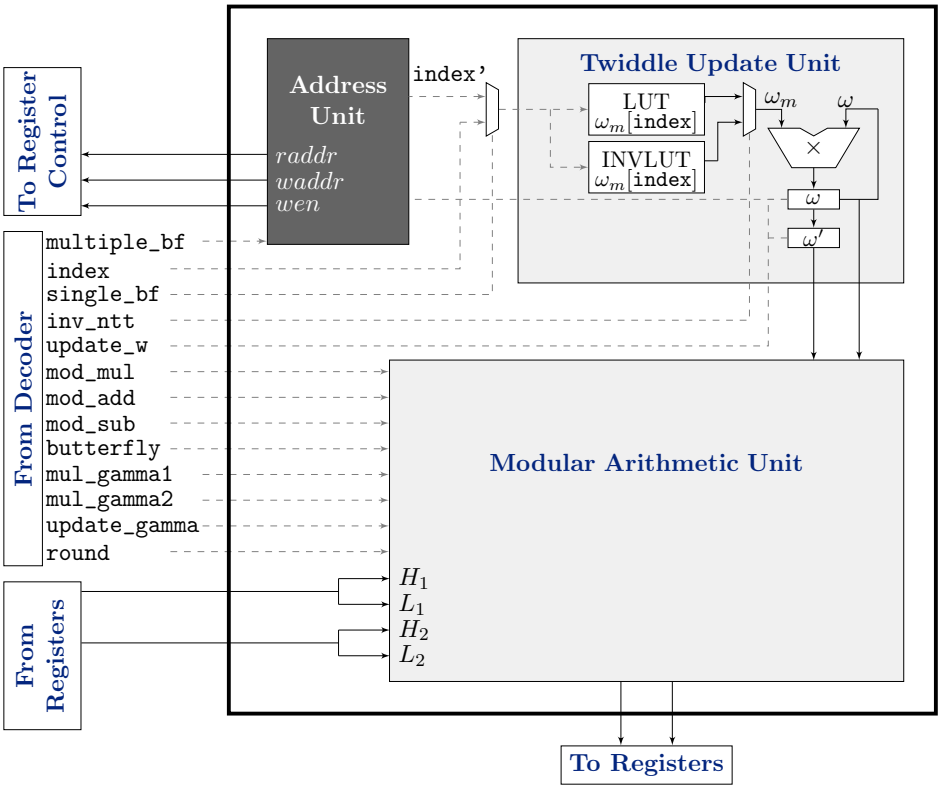


Figure 4.6: NTT and Modular Arithmetic Unit.

4.3.2 Tightly Coupled NTT Accelerator

Figure 4.6 illustrates the hardware architecture of the proposed *NTT and Modular Arithmetic Unit*. The architecture is composed of three main modules: *Address Unit*, *Twiddle Update Unit*, and *Modular Arithmetic Unit*. The design is optimized for the $\text{NTT}^{CT}_{br\leftarrow no}$ algorithm, which is used in NewHope, but also supports different NTT algorithms. The input of the *NTT and Modular Arithmetic Unit* is the content of two registers from the processor's register bank (with lower halfwords L_1/L_2 and higher halfwords H_1/H_2) and the control signals from the instruction decoder. The output consists of the processed input values and the control signals for the register bank.

Address Unit. This module controls the automatic NTT computation and the merge of five NTT layers by setting the two read and write addresses for the register bank according to Algorithm 9, Section 3.3 (with modified loop start and end values). In this work, the automatic address calculation is only supported for the $\text{NTT}^{CT}_{br\leftarrow no}$ variant and therefore only for NewHope. Supporting different NTT variants would be possible but has been left as future work. The address calculation is triggered by the `multiple_bf` signal. At each clock cycle, the read/write addresses and write enable signal are updated. The *Address Unit* is also responsible for selecting the correct index for the small LUT

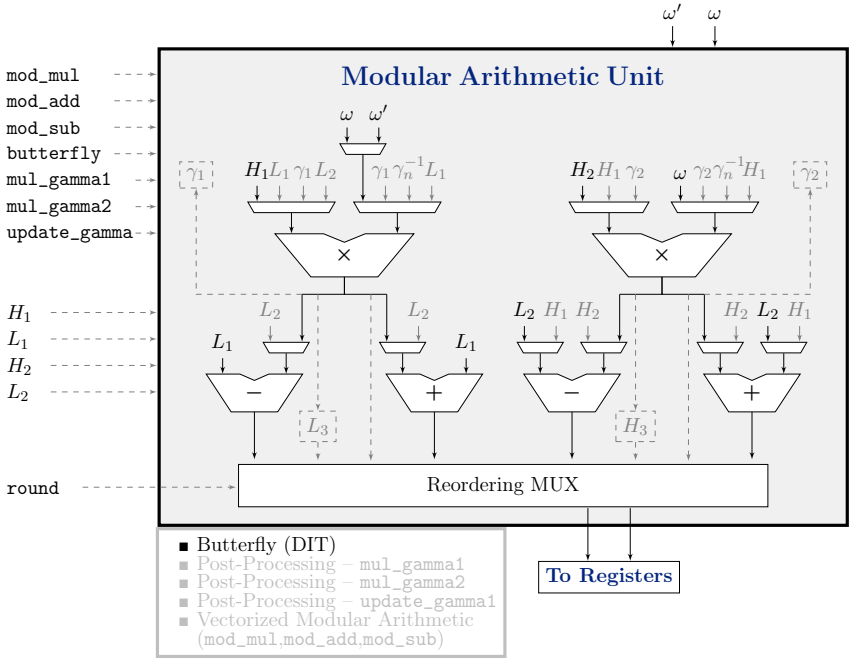


Figure 4.7: Modular Arithmetic Unit – butterfly operation decimation-in-time.

of the precalculated values of ω_m and for triggering the Twiddle factor update. When the *NTT and Modular Arithmetic Unit* is in single operation mode, the *Address Unit* remains in an idle state.

Twiddle Update Unit. This module calculates the Twiddle factors ω on-the-fly. The update of ω is always triggered by the `update_w` signal, which is either set by the *Address Unit* or by the corresponding signal from the instruction decoder. The Twiddle factor update $\omega = \omega \cdot \omega_m \mod q$ is performed as described in Algorithm 9, Lines 17 and 29 (Section 3.3). The value for ω_m is determined through the current NTT layer. All $log_2(n)$ possible values for ω_m are precalculated and stored in a LUT within the hardware accelerator. The `index` signal is used to select the correct value from the LUT. Depending on the value of the `inv_ntt` signal, either the LUT for the forward or inverse NTT is selected.

Modular Arithmetic Unit. This module can perform the following operations: butterfly operation (decimation-in-time and decimation-in-frequency), post-processing (`mul_gamma1`, `mul_gamma2`, `update_gamma`), and vectorized modular arithmetic (`mod_mul`, `mod_add`, `mod_sub`). The detailed microarchitecture of the *Modular Arithmetic Unit* in the butterfly operation mode is shown in Figure 4.7. A set of multiplexers is used to set the desired operation.

The butterfly operation mode (triggered by `butterfly`) calculates two butterfly operations $\{L_1 - H_1 \cdot \omega,\ L_1 + H_1 \cdot \omega\}$ and $\{L_2 - H_2 \cdot \omega,\ L_2 + H_2 \cdot \omega\}$ in parallel (modular

reduction omitted in the description). The previous Twiddle factor ω' is also forwarded to the *Modular Arithmetic Unit* to allow the computation of two butterfly operations in parallel within the last NTT layer. As a result, the utilization of the *Modular Arithmetic Unit* can be increased. The *Reordering MUX* is responsible for preparing the coefficients for the next round (NTT layer) by swapping the coefficients after the butterfly operation for all rounds except for the last one. The architecture is extended to support the calculation of the decimation-in-frequency butterfly operation, i.e., the computation of $\{(L_1 - H_1) \cdot \omega,\ L_1 + H_1\}$ and $\{(L_2 - H_2) \cdot \omega,\ L_2 + H_2\}$. For reasons of better visibility, this extension is not shown in Figure 4.7. In order to avoid any combinatorial loop in the circuit, either two additional modular multipliers or modular subtractors are required. As such subtractors are less complex than multipliers, they were chosen and placed on top of the existing multipliers. The input values of the additional subtractors are (L_1, H_1) and (L_2, H_2). The output is connected to the multiplexers of the left input of the existing multipliers.

The post-processing operations (`mul_gamma1`, `mul_gamma2`, `update_gamma`) are used to calculate the multiplications with n^{-1} and γ_n^{-i} at the inverse NTT operation. These multiplications can be merged with the last layer of the inverse NTT. The control signal `mul_gamma1` ensures that the coefficients a_i and $a_{i+n/2}$ of the first input (L_1, H_1) are multiplied with $\gamma_1 = n^{-1}\gamma_n^{-i}$ and $\gamma_2 = n^{-1}\gamma_n^{-i-n/2}$, respectively. Before the multiplication with the first coefficients a_0 and $a_{n/2}$ starts, γ_1 is initialized with $n^{-1}\gamma_n^0 \mod q$ and γ_2 with $n^{-1}\gamma_n^{-n/2} \mod q$. The `mul_gamma2` control signal is used to perform the multiplications with the next coefficient pair a_{i+1} and $a_{i+1+n/2}$. The *Reordering MUX* brings the results of the `mul_gamma1` and `mul_gamma2` operation in the desired order $\{a_{i+1}, a_i\}$ and $\{a_{i+1+n/2}, a_{i+n/2}\}$ and assigns the result to the output. The `update_gamma` signal is used to update $\gamma_1 = \gamma_1\gamma_n^{-1}$ and $\gamma_2 = \gamma_2\gamma_n^{-1}$ after each `mul_gamma1` and `mul_gamma2` operation.

The vectorized modular arithmetic (`mod_mul`, `mod_add`, `mod_sub`) calculates vectorized modular multiplications ($L_1 \cdot L_2 \mod q$, $H_1 \cdot H_2 \mod q$), additions ($L_1 + L_2 \mod q$, $H_1 + H_2 \mod q$), and subtractions ($L_1 - L_2 \mod q$, $H_1 - H_2 \mod q$). Like the butterfly operation, the vectorized modular arithmetic belongs to the category of packed arithmetic and follows the Single Instruction Multiple Data (SIMD) principle.

Bit-Reversal. Reversing the bits is an expensive software task. A straightforward approach is looping through all $m = log_2(n)$ bits of an integer. The fastest solution is to use a LUT with n entries of m bits each. For large arrays, i.e., for large polynomial lengths, this approach leads to a high memory footprint. To achieve a better trade-off between memory footprint and performance, the RISC-V ISA is extended, and special instructions for the bit-reversal operation are developed. These instructions are derived from the store word/halfword operations *sw, sh*. In addition to the value to be stored and the destination address, the new instructions also take an offset. This offset is added in bit-reversed order to the destination address. The functionality of the new instructions can be expressed as $\text{MEM}_{rs1+bitrev(rs2)} \leftarrow rd$, where $rs1$ contains the destination address, $rs2$ the offset, and rd the value that is stored. Reversing the offset in hardware

can be efficiently solved through rewiring. The bit-reversal of a whole polynomial can be performed by loading in a loop each coefficient and storing the coefficient with the new instruction. In this case, the offset simply corresponds to the loop counter.

4.3.3 Experimental Results of Tightly Coupled NTT Accelerator

In the remainder of this chapter, the RISC-V PULP toolchain (Version 7.1.1) with compiler flag `-O3` (optimization for speed) is used to compile the software programs. This toolchain supports customized instructions for the PULP cores that are not part of the RISC-V standard. Table 4.6 summarizes the clock cycle count required for the NTT, INVNTT, and bit-reversal of NewHope and Kyber. The following implementations are compared: (i) the RISC-V baseline implementation derived from [AAB+19a] and [ABD+19]; (ii) the RISC-V implementations with finite field multiplication accelerator in [AEL+20]; (iii) the loosely coupled NTT accelerator implementation of Chapter 3.3; and (iv) the tightly coupled NTT accelerator implementation of this chapter. As discussed in Section 4.3.1, the bit-reversal (BitRev) is eliminated for Kyber. For the loosely coupled accelerator, the bit-reversal costs are hidden through a rewiring when copying the input coefficients to the NTT memory.

The results show that the tightly coupled approach of this work achieves the lowest cycle count. In comparison to the baseline implementation of NewHope, it achieves a speedup factor of 13.18/12.40 (NTT/INVNTT) for $n = 512$ and 13.01/11.95 (NTT/INVNTT) for $n = 1024$. Further, the proposed bit-reversal instruction allows eliminating any LUTs for this step (1024 bytes for NewHope-512 and 2048 bytes for NewHope-1024). Moreover, it improves the performance. Although the architecture of the *NTT and Modular Arithmetic Unit* is not optimized for the $\text{NTT}^{CT}_{no\rightarrow br}$ and $\text{INVNTT}^{GS}_{br\rightarrow no}$ variants, Kyber achieves a considerable speedup factor of 17.93/27.79 (NTT/INVNTT) when compared to the baseline. Kyber strongly benefits from the parallel decimation-in-frequency butterfly and vectorized modular multiplication operations.

Another advantage of the proposed architecture is the reduction of precomputations. The NewHope reference implementation in [AAB+19a] requires $7n$ precomputed bytes (n denotes the polynomial length) for the bit-reversal step, Twiddle factors, and pre-/post processing. The proposed implementation only requires $4 \cdot log_2(n) + 4$ bytes. In concrete numbers, the LUTs were reduced from 7 168 to 44 bytes for NewHope-1024 and from 3 584 to 40 bytes for NewHope-512. Moreover, in contrast to the loosely coupled approach, the tightly coupled architecture does not require extra memory blocks or input/output buffers.

4.3.4 Tightly Coupled Accelerator for Karatsuba/Toom–Cook Multiplications

Recent works proposed a combination of the Karatsuba and Toom–Cook methods to perform the polynomial multiplication in Saber [DKRV18, DKRV20, BMKV20]. Similar to [DKRV18], in this chapter, the polynomials of Saber are split using four-way Toom–Cook and two consecutive levels of Karatsuba. The polynomial length is then small

Table 4.6: Cycle count of the NTT operation.

	Device	NTT	INVNTT	BitRev
NewHope-512 [ABCG20]	ARM Cortex-M4	31 217	23 439	–
NewHope-512 [AEL+20]	RISC-V (VexRiscv)	14 787	14 893	0
NewHope-512 baseline	RISC-V (PULPino)	107 666	107 668	6 623
NewHope-512 tightly	**RISC-V (PULPino)**	**8 169**	**8 684**	**2 056**
NewHope-1024 [ABCG20]	ARM Cortex-M4	68 131	51 231	–
NewHope-1024 [AEL+20]	RISC-V (VexRiscv)	31 295	31 735	0
NewHope-1024 baseline	RISC-V (PULPino)	241 121	241 123	13 279
NewHope-1024 loosely	RISC-V (PULPino)	24 119	24 119	0
NewHope-1024 tightly	**RISC-V (PULPino)**	**18 537**	**20 171**	**4 105**
Kyber [ABCG20]	ARM Cortex-M4	6 855	6 983	0
Kyber [AEL+20]	RISC-V (VexRiscv)	6 868	6 367	0
Kyber baseline	RISC-V (PULPino)	34 703	53 636	0
Kyber tightly	**RISC-V (PULPino)**	**1 935**	**1 930**	**0**

enough (16 coefficients) to efficiently apply the schoolbook multiplication. At a certain point, further splitting the polynomials does not bring any performance advantage since the savings derived from the multiplication do not outweigh the increasing number of additions.

Since the coefficients in Saber are, similar to NewHope and Kyber, smaller than 16-bit, they are suitable for packed (vectorized) arithmetic. Although the ISA extension for packed arithmetic of the RISC-V specification[1] is still in draft mode, the RISC-V core used in this work already supports some packed operations [TGS]. In the following, useful instructions for the polynomial multiplication in Saber are listed:

$$
\begin{aligned}
pv.add.h: &\quad rd[15{:}0] = (rs1[15{:}0] + rs2[15{:}0]) \mod 2^{16} \\
&\quad rd[31{:}16] = (rs1[31{:}16] + rs2[31{:}16]) \mod 2^{16} \\
p.mulu: &\quad rd[31{:}0] = rs1[15{:}0] \cdot rs2[15{:}0] \\
p.mulhhu: &\quad rd[31{:}0] = rs1[31{:}15] \cdot rs2[31{:}15] \\
p.macuN: &\quad rd[31{:}0] = (rs1[15{:}0] \cdot rs2[15{:}0] + rd) >> Is3 \\
p.machhuN: &\quad rd[31{:}0] = (rs1[31{:}16] \cdot rs2[31{:}16] + rd) >> Is3
\end{aligned}
$$

Saber and other PQC candidates, e.g., some NTRU variants, use a power-of-two modulus with $q \leq 2^{16}$. This allows processing two multiplications in parallel. Besides, the schoolbook and Karatsuba multiplication in Equation 2.7 (Section 2) benefit from the Multiply Accumulate (MAC) principle. As a result, a vectorized modular multiply-accumulate function is proposed. It is defined by:

$$
\begin{aligned}
pq.mac: &\quad rd[15{:}0] = (rs1[15{:}0] \cdot rs2[15{:}0] + rd[15{:}0]) \mod q' \\
&\quad rd[31{:}16] = (rs1[31{:}16] \cdot rs2[31{:}16] + rd[31{:}16]) \mod q'
\end{aligned}
$$

In this work, the parameter q' is set to 2^{16} in order to increase the flexibility and to

[1] https://github.com/riscv/riscv-isa-manual/releases/download/Ratified-IMAFDQC/riscv-spec-20191213.pdf Version 20191213 (Last accessed 1st Nov. 2021).

support multiple schemes. After performing the polynomial multiplication, the result can be reduced with the original modulus q because $(a \mod q') \mod q \equiv a \mod q$ if both moduli are a power of two and $q' \geq q$. When using the *pq.mac* operation, the number of clock cycles for the polynomial multiplication in Saber was reduced from 104 074 to 71 349.

4.3.5 Tightly Coupled Hash Accelerator

In Section 3.3, a loosely coupled Keccak implementation optimized for high performance is presented. The aim of this section is to find an alternative design approach to decrease the resource consumption while maintaining a good performance.

The resource consumption and performance mainly depend on how the Keccak state is stored and how it is accessible for the Keccak operations θ, ρ, π, χ, and ι. The Keccak state is a three-dimensional bit array that must be stored in a two-dimensional memory location. Each of the Keccak operations transforms the state into a new state, as illustrated in Figure 4.8.

In [BDH+20], the Keccak team presented three different Keccak hardware designs[2]: a high-speed core, a low-area core, and a mid-range core (trade-off between speed and area). The high-speed core is based on similar principles as the standalone loosely coupled coprocessor presented in Section 3.3. It requires a buffer for the input message and output hash. The Keccak state is stored in internal registers. This allows accessing all state bits in parallel. The low-area core reuses the system memory as storage for the Keccak state. Only some intermediate results are internally stored in this design. This method saves a lot of resources compared to the high-speed core. On the other side, memories usually have only one or two read/write ports. Therefore, loading and storing the Keccak state is time-consuming. The mid-range core splits the state into smaller chunks to reduce the area overhead. Further, the order of the permutation function is modified such that the slice and lane-oriented steps can be processed together, respectively. As the ρ, π and ι steps work on lanes, and the χ and θ steps on slices, the state must be loaded at least two times for each `f-1600` permutation.

The proposed accelerator of this section does not split the state into multiple parts to keep the amount of memory accesses small. Instead, the complete state is stored in the FPR with 32×32-bit registers and in a part of the GPR with 18×32-bit (in the temporary registers `t0`–`t6` and saved registers `s1`–`s11`). Still enough registers remain untouched in the GPR to guarantee a normal operation of the RISC-V core. The saved registers must be stored on the stack before the Keccak operation starts. In order to keep a high degree of flexibility, one round of the Keccak `f-1600` permutation function is accelerated in hardware. The advantages of this design approach are that available system resources are reused and that the full state can be accessed in parallel to achieve a high performance. In contrast to the loosely coupled Keccak design, no input/output buffers, control circuit, and absorb module are required.

Figure 4.8 illustrates the proposed tightly coupled Keccak accelerator. The accelerator

[2] https://keccak.team/files/Keccak-implementation-3.2.pdf (Last accessed 1st Nov. 2021).

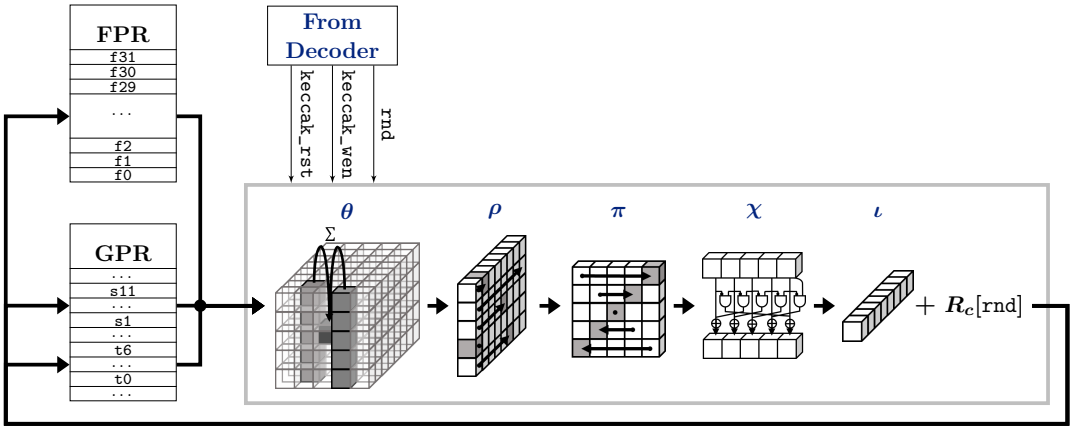


Figure 4.8: Tightly coupled Keccak accelerator.

takes as input the mentioned registers from FPR/GPR, the round signal `rnd`, the start signal `keccak_wen`, and the reset signal `keccak_rst`. Triggered by the start signal, the accelerator performs one round of the `f-1600` function, where the round signal selects the corresponding round constant. The result of this operation is written back to the registers.

The polynomial sampling can be performed with the following steps. The state is first set to zero using the `keccak_rst` signal, which resets all related registers in one cycle. After this initialization, the input message or a message block is written together with a constant into a subset of the state. This step is also known as the Keccak absorption. The state permutation will then transform this initial state. Depending on the rate, a certain number of bits is squeezed out while a part of the state, the capacity, remains untouched. The squeezed output is then processed to model the desired distribution of the polynomial coefficients. Thereby, the state registers must remain untouched when the state is not written back to the memory. In order to obtain fresh randomness, the state is permuted and squeezed again. Keeping the state for the whole polynomial sampling process within the registers leads to a significant performance improvement as slow main memory accesses to store the state are avoided. Only in case of an interrupt, the respective state registers must be stored in memory.

4.3.6 Tightly Coupled Binomial Sampling Accelerator

Many LWE-based schemes, such as NewHope, Kyber, and Saber, replaced the discrete Gaussian error distribution with a centered binomial distribution. This increases the efficiency and avoids complex arithmetic or table lockups. Let Ψ_η be a centered binomial distribution with standard deviation $\sigma = \sqrt{\eta/2}$. Let x and x' be two uniformly distributed random η-bit integers and let x_i denote the i-th bit of the integer x. Given

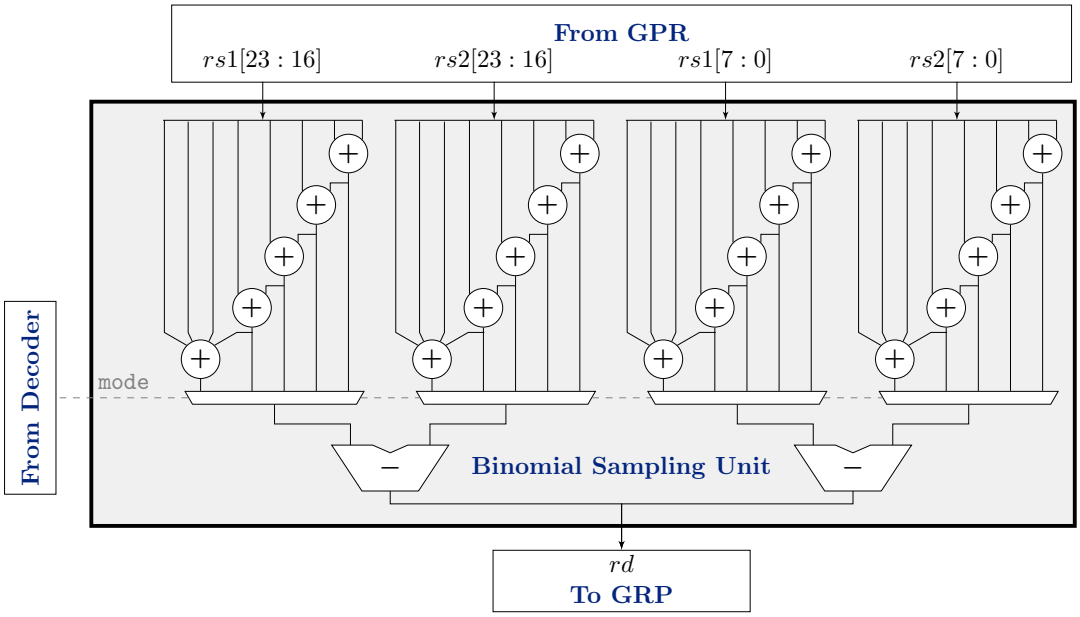


Figure 4.9: Binomial Sampling Unit.

x and x', a binomially distributed sample can be retrieved by

$$\Psi_\eta = \sum_{i=0}^{\eta-1}(x_i - x'_i) \mod q \quad , \tag{4.4}$$

for any modulus $q \in \mathbb{Z}$.

Figure 4.9 shows the proposed tightly coupled *Binomial Sampling Unit*. It reads the uniformly distributed samples of registers $rs1$ and $rs2$ and transforms them into binomially distributed coefficients. Two output samples are computed in parallel as a single coefficient is at most 16-bit for the considered schemes. The *Binomial Sampling Unit* supports $\eta = \{2, 3, 4, 5, 8\}$ to cover all instances of NewHope, Kyber, and Saber. The least significant byte of the first input register ($rs1[7:0]$) contains the η-bit integer x, and the least significant byte of the second input register ($rs2[7:0]$) contains the η-bit integer x'. The two adder trees on the right side of Figure 4.9 compute $s(\eta) = \sum_{i=0}^{\eta-1} x_i$ and $s'(\eta) = \sum_{i=0}^{\eta-1} x'_i$ for all $\eta = \{2, 3, 4, 5, 8\}$. The `mode` signal is responsible for the configuration of the multiplexers. It ensures that the corresponding sums $s(\eta_{sel})$ and $s'(\eta_{sel})$ are forwarded to the modular subtractor. The modular subtractor computes $s(\eta_{sel}) - s'(\eta_{sel}) \mod q$ as in Equation 4.4. The left side of Figure 4.9 transforms the next sample pair in parallel. The corresponding uniform input samples are located in the upper halfword of $rs1$ and $rs2$. The results of the two modular subtractions are combined in register rd.

Table 4.7: Cycle count of SHAKE-256 (32-byte input/output length), GenA, and Sample. Kyber and Saber have for all parameter sets the same polynomial length. The results for the sampling are given for the generation of one polynomial.

	SHAKE-256	GenA	Sample
NewHope-512 baseline	31 907	272 615	280 079
NewHope-512 tightly	**308**	**10 136**	**7 847**
NewHope-1024 baseline	31 907	548 019	560 058
NewHope-1024 loosely	–	48 730	72 840
NewHope-1024 tightly	**308**	**20 205**	**15 643**
Kyber-512 baseline	31 907	501 300	33 902
Kyber-512 tightly	**308**	**22 414**	**2 375**
Lightsaber baseline	–	306 387	130 896
Lightsaber tightly	–	**9 652**	**3 260**

4.3.7 Experimental Results of Keccak and Polynomial Sampling

Table 4.7 summarizes the clock cycle count for the accelerated SHAKE-256, uniform sampling of polynomial a (GenA), and binomial sampling from the error distribution (Sample). The proposed tightly coupled approach accelerates SHAKE-256 by a factor of 103.59 compared to the software baseline implementation on RISC-V. For GenA, a speedup factor of up to 27.12 for NewHope, 22.37 for Kyber, and 31.74 for Saber is achieved. For Sample, the measured speedup factors are between 14.27 and 40.15. The speedup for Kyber is smaller compared to NewHope or Lightsaber due to the smaller variance of the error distribution.

The tightly coupled approach is significantly faster than the speed-optimized loosely coupled coprocessor of Section 3.3. The main reason for the remarkable performance is that the Keccak state is held within the register sets for the complete sampling process, and complex bus communication is completely avoided.

4.3.8 Core Integration of Modular Arithmetic and Sampling Accelerators

Figure 4.10 illustrates the architecture of the CV32E40P RISC-V core with integrated modular arithmetic and sampling accelerators. It consists of two new components PQI-ALU and PQII-ALU. All accelerators within these components are added as modules and can be selected using dedicated define directives.

The PQI-ALU contains the *NTT and Modular Arithmetic Unit* and the Keccak accelerator. It is placed within the decode stage to achieve a particular tight coupling to the register sets. This avoids routing a large number of register signals to the execution stage. The PQII-ALU contains the *Binomial Sampling Unit*. This accelerator requires access to two input and one output register. In order to reuse existing hardware resources, the *pq.mac* operation is directly integrated into the *MULT Unit*. The hardware resources for performing the multiplications are already available in the *MULT Unit*. An extension for the *pq.mac* support comes with a negligible overhead of multiplexers and two adders.

The developed post-quantum instructions to control the accelerators can be divided

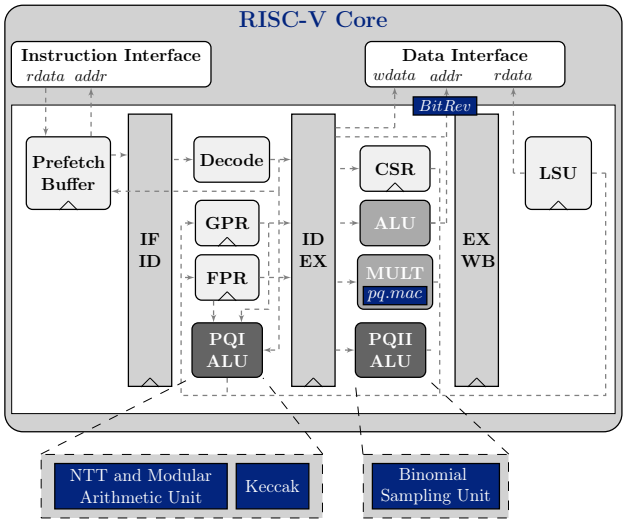


Figure 4.10: RISC-V core integration of tightly coupled accelerators.

into seven main classes: NTT configuration, NTT operation, modular arithmetic, bit-reversal, PQ-MAC, Hash, and binomial sampling. A complete list of all instructions is provided in Table 4.8.

NTT configuration class. This class contains instructions for the following functionalities: (i) setting the scheme NewHope-512, NewHope-1024, or Kyber (all security categories); (ii) setting the forward or inverse NTT; and (iii) setting either the first rounds or the last round of the NTT. Selecting one of the NewHope variants sets the modulus to $q = 12289$ and the Montgomery parameter for the modular multipliers to $-q^{-1} \mod R = 12287$ with $R = 2^{18}$. When Kyber is selected, $q = 3329$ and $-q^{-1} \mod R = 199935$ is set. Setting the forward or inverse NTT determines the selection for the precomputed values of ω_m. The selection of the first rounds or last round activates or deactivates the swapping of coefficients.

NTT operation class. This class contains all instructions for the optimized $\text{NTT}^{CT}_{br \leftarrow no}$ computation. The *pq.ntt_multiple_bf* instruction triggers the automatic calculation of the five merged NTT layers. The *pq.ntt_single_bf* instruction is used to calculate two parallel decimation-in-time butterfly operations. The instructions *pq.update_m* and *pq.update_omega* are used to control the *Twiddle Update Unit*. The *pq.mul_gamma1*, *pq.mul_gamma2*, and *pq.update_gamma* instructions are provided for the multiplications with n^{-1} and γ_n^{-i} at the inverse transform.

Modular arithmetic class. This class contains instructions for the vectorized modular multiplication, addition, subtraction, and butterfly operations (decimation-in-time and decimation-in-frequency). For these operations, $rs1$ and $rs2$ are used as source registers and rd as the destination register. At butterfly instructions, $rs1$ is also used for the

Table 4.8: PQC ISA extension for lattice-based cryptography

Opcode	Funct3	Funct7	Operation Name	Cycles
111 0111	000	0000000	NTT Configuration: *pq.set_kyber*	1
111 0111	001	0000000	NTT Configuration: *pq.set_newhope512*	1
111 0111	010	0000000	NTT Configuration: *pq.set_newhope1024*	1
111 0111	011	0000000	NTT Configuration: *pq.set_fwd_ntt*	1
111 0111	100	0000000	NTT Configuration: *pq.set_inv_ntt*	1
111 0111	101	0000000	NTT Configuration: *pq.set_first_rounds*	1
111 0111	110	0000000	NTT Configuration: *pq.set_last_round*	1
111 0111	000	0000001	NTT Operation: *pq.ntt_multiple_bf*	83
111 0111	001	0000001	NTT Operation: *pq.ntt_single_bf*	1
111 0111	010	0000001	NTT Operation: *pq.update_m*	1
111 0111	011	0000001	NTT Operation: *pq.update_omega*	1
111 0111	100	0000001	NTT Operation: *pq.mul_gamma1*	1
111 0111	101	0000001	NTT Operation: *pq.mul_gamma2*	1
111 0111	110	0000001	NTT Operation: *pq.update_gamma*	1
111 0111	000	0000010	Modular Arithmetic: *pq.mod_mul*	1
111 0111	001	0000010	Modular Arithmetic: *pq.mod_add*	1
111 0111	010	0000010	Modular Arithmetic: *pq.mod_sub*	1
111 0111	011	0000010	Modular Arithmetic: *pq.bf_dit*	1
111 0111	100	0000010	Modular Arithmetic: *pq.bf_dif*	1
111 0111	000	0000011	Bit-reversal: *pq.br256*	1
111 0111	001	0000011	Bit-reversal: *pq.br512*	1
111 0111	010	0000011	Bit-reversal: *pq.br1024*	1
111 0111	000	0000100	Hash: *keccak.f1600*	1
111 0111	000	0000101	Binomial Sampling: *pq.bs_k2*	1
111 0111	001	0000101	Binomial Sampling: *pq.bs_k3*	1
111 0111	010	0000101	Binomial Sampling: *pq.bs_k4*	1
111 0111	011	0000101	Binomial Sampling: *pq.bs_k5*	1
111 0111	100	0000101	Binomial Sampling: *pq.bs_k8*	1
111 0111	000	0000110	Vectorized Modular Multiply Accumulate: *pq.mac*	1

second output. In contrast to the *pq.ntt_single_bf* operation, used for the optimized $\text{NTT}^{CT}_{br\leftarrow no}$, the *pq.bf_dit* operation does not use registers from the FPR but the GPR.

Bit-reversal class. This class contains the bit-reversal instructions for the polynomial lengths $n = 256$, $n = 512$, and $n = 1024$. The register rd contains the value that has to be stored, $rs1$ the base address of the store location, and $rs2$ the offset value. The LSU handles the bit-reversal and stores the coefficient to the desired location.

PQ-MAC class. This class contains the vectorized modular multiply-accumulate instruction *pq.mac* as described in Section 4.3.4.

Hash class. This class contains the Keccak instruction *keccak.f1600*. The register $rs1$ selects the current Keccak round, and $rs2$ is used to reset the state.

Binomial sampling class. This class contains instructions to turn uniform samples stored in $rs1$ and $rs2$ into binomially distributed coefficients stored in rd.

Except for the *pq.ntt_multiple_bf* operation, all operations are single cycled. The *pq.ntt_multiple_bf* function requires 83 clock cycles (80 cycles for the functionality and 3 cycles to fill the address generation pipeline) to complete the automized NTT operations for the merged layers. While the *pq.ntt_multiple_bf* writes the last results to the registers, the processing element can already read the first results after 68 cycles.

4.3.9 Experimental Performance Results

Table 4.9 summarizes the benchmark results of the proposed tightly coupled accelerators. Four implementation categories are benchmarked to analyze the influence of the proposed accelerators on the two performance bottlenecks: (i) baseline implementation; (ii) implementation with optimized modular arithmetic and NTT computations (using *NTT and Modular Arithmetic Unit*, bit reversal, and *pq.mac*); (iii) implementation with optimized polynomial sampling (using Keccak and *Binomial Sampling Unit*); and (iv) implementation with all optimizations presented in this chapter. The baseline implementations are derived from the reference implementations in [AAB+19a], [ABD+19], and [DKRV20].

Comparison with baseline implementations. Compared to the baseline implementation, a cycle count speedup factor of up to 11.39 for NewHope, up to 9.62 for Kyber, and up to 2.65 for Saber was achieved for a complete algorithm run. It must be noted that the baseline implementation might be improved using assembly optimizations. However, assembly optimizations for RISC-V are still largely unexplored. Although the *NTT and Modular Arithmetic Unit* is not particularly designed for the NTT types used in Kyber, a significant performance improvement was measured. Due to the early abort of the NTT, Kyber has a complex basecase multiplication, which highly benefits from the proposed

vectorized modular arithmetic instructions. For the Saber instances, the polynomial multiplication remains the performance bottleneck, although the *pq.mac* operation already brings a considerable improvement.

Comparison with ARM Cortex-M4 implementations. The presented work beats the cycle count of the latest assembly optimized ARM Cortex-M4 implementations of NewHope and Kyber in [ABCG20, KRSS19]. Compared to [ABCG20], a clock cycle speedup factor of up to 4.46 for NewHope and up to 3.84 for Kyber was achieved for a complete algorithm run. When comparing with the Karatsuba/Toom–Cook optimized Saber implementations on ARM Cortex-M4 [BMKV20], the achieved speedup factors are between 0.97 (Firesaber) and 1.16 (Lightsaber). For this comparison, it must be considered that the deployed RISC-V core and compiler are less advanced than the commercial ARM Cortex-M4 products.

Comparison with RISC-V implementations. The work in [AEL+20] has shown that NewHope and Kyber can benefit from instruction set extensions for finite field multiplications. The more powerful accelerators proposed in this work lead to a further significant speedup. In comparison to the design in [AEL+20], this work achieves a cycle count speedup factor of up to 7.11 for NewHope and up to 6.15 for Kyber. The authors in [AEL+20] use a Barrett multiplier to accelerate the modular arithmetic. In contrast to the work in [AEL+20], which can calculate one modular multiplication at a time, the proposed solution of this work is more powerful and versatile as it can calculate complete vectorized butterfly operations and vectorized modular additions, subtractions, and multiplications.

Optimizing the modular and NTT arithmetic turns out to be not sufficient to alleviate all performance bottlenecks. For all schemes and security levels, the ISA extensions for accelerating the polynomial sampling have shown a more significant impact on the performance when compared to the extensions for the modular and NTT arithmetic. Using the sampling extensions alone will already beat the cycle count of the latest ARM Cortex-M4 implementations for NewHope and Kyber.

Furthermore, the tightly coupled design is faster than the loosely coupled NewHope implementation in Section 3.3 (improvement factor of 3.58 for the whole algorithm). The better performance can be explained by the reduced communication overhead (due to the tight coupling of the NTT and Keccak accelerators) and by the introduction of vectorized modular arithmetic and binomial sampling instruction set extensions. Moreover, the generic *Modular Arithmetic Unit* allows accelerating coefficient-wise multiplications, additions, and subtractions.

Code size. Table 4.10 summarizes the measured code size for the baseline and optimized implementations. In particular, for the optimized NewHope implementations, the code size was significantly decreased compared to the baseline implementation. This is mainly achieved due to the elimination of large LUTs for the Twiddle factors and bitreversal. Moreover, the ISA extensions have the side effect that fewer instructions for

Table 4.9: Cycle count of NewHope, Kyber, and Saber with tightly coupled accelerators.

Algorithm	Device	KeyGen	Encaps	Decaps
NewHope-512 CCA [ABCG20]	ARM Cortex-M4	578 890	858 982	806 300
NewHope-512 CCA [AEL+20]	RISC-V (VexRiscv)	904 000	1 424 000	1 302 000
NewHope-512 CCA baseline	RISC-V (PULPino)	1 370 735	2 153 075	2 091 823
NewHope-512 CCA opt. arithmetic	RISC-V (PULPino)	1 144 997	1 781 652	1 585 400
NewHope-512 CCA opt. sampling	RISC-V (PULPino)	350 939	569 945	719 027
NewHope-512 CCA opt. (all)	**RISC-V (PULPino)**	**116 991**	**195 449**	**209 915**
NewHope-1024 CCA [KRSS19]	ARM Cortex-M4	1 219 908	1 903 231	1 927 505
NewHope-1024 CCA [ABCG20]	ARM Cortex-M4	1 157 222	1 674 899	1 587 107
NewHope-1024 CCA [AEL+20]	RISC-V (VexRiscv)	1 776 000	2 742 000	2 528 000
NewHope-1024 CCA baseline	RISC-V (PULPino)	2 767 270	4 282 504	4 239 534
NewHope-1024 CCA loosely	RISC-V (PULPino)	784 734	1 534 879	1 229 142
NewHope-1024 CCA opt. arithmetic	RISC-V (PULPino)	2 238 982	3 425 458	3 076 761
NewHope-1024 CCA opt. sampling	RISC-V (PULPino)	751 826	1 220 705	1 572 218
NewHope-1024 CCA opt. (all)	**RISC-V (PULPino)**	**218 367**	**363 658**	**409 444**
Kyber-512 CCA [KRSS19]	ARM Cortex-M4	514 291	652 769	621 245
Kyber-512 CCA [ABCG20]	ARM Cortex-M4	455 191	586 334	543 500
Kyber-512 CCA [Gre20]	RISC-V (VexRiscv)	1 218 557	1 592 689	1 515 876
Kyber-512 CCA [AEL+20]	RISC-V (VexRiscv)	710 000	971 000	870 000
Kyber-512 CCA baseline	RISC-V (PULPino)	1 137 052	1 547 789	1 525 621
Kyber-512 CCA opt. arithmetic	RISC-V (PULPino)	939 932	1 223 887	1 051 003
Kyber-512 CCA opt. sampling	RISC-V (PULPino)	356 758	514 652	678 938
Kyber-512 CCA opt. (all)	**RISC-V (PULPino)**	**150 106**	**193 076**	**204 843**
Kyber-768 CCA [KRSS19]	ARM Cortex-M4	976 757	1 146 556	1 094 849
Kyber-768 CCA [ABCG20]	ARM Cortex-M4	864 008	1 032 540	969 867
Kyber-768 CCA [Gre20]	RISC-V (VexRiscv)	2 288 109	2 771 517	2 653 584
Kyber-768 CCA baseline	RISC-V (PULPino)	2 102 505	2 625 824	2 573 963
Kyber-768 CCA opt. arithmetic	RISC-V (PULPino)	1 768 400	2 138 810	1 889 930
Kyber-768 CCA opt. sampling	RISC-V (PULPino)	625 943	832 137	1 048 473
Kyber-768 CCA opt. (all)	**RISC-V (PULPino)**	**273 370**	**325 888**	**340 418**
Kyber-1024 CCA [KRSS19]	ARM Cortex-M4	1 575 052	1 779 848	1 709 348
Kyber-1024 CCA [ABCG20]	ARM Cortex-M4	1 404 695	1 605 707	1 525 805
Kyber-1024 CCA [Gre20]	RISC-V (VexRiscv)	3 686 344	4 280 420	4 123 722
Kyber-1024 CCA [AEL+20]	RISC-V (VexRiscv)	2 203 000	2 619 000	2 429 000
Kyber-1024 CCA baseline	RISC-V (PULPino)	3 378 603	4 024 887	3 949 039
Kyber-1024 CCA opt. arithmetic	RISC-V (PULPino)	2 856 302	3 312 957	2 989 896
Kyber-1024 CCA opt. sampling	RISC-V (PULPino)	872 686	1 118 704	1 385 263
Kyber-1024 CCA opt. (all)	**RISC-V (PULPino)**	**349 673**	**405 477**	**424 682**
Lightsaber CCA [BMKV20]	ARM Cortex-M4	466 000	653 000	678 000
Lightsaber CCA [KRSS19]	ARM Cortex-M4	459 965	651 273	678 810
Lightsaber CCA baseline	RISC-V (PULPino)	1 071 836	1 503 594	1 537 939
Lightsaber CCA opt. arithmetic	RISC-V (PULPino)	947 777	1 317 503	1 289 533
Lightsaber CCA opt. sampling	RISC-V (PULPino)	495 211	719 084	914 072
Lightsaber CCA opt. (all)	**RISC-V (PULPino)**	**366 837**	**526 496**	**657 583**
Saber CCA [BMKV20]	ARM Cortex-M4	853 000	1 103 000	1 127 000
Saber CCA [KRSS19]	ARM Cortex-M4	896 035	1 161 849	1 204 633
Saber CCA baseline	RISC-V (PULPino)	2 110 283	2 737 181	2 797 400
Saber CCA opt. arithmetic	RISC-V (PULPino)	1 824 799	2 354 078	2 317 110
Saber CCA opt. sampling	RISC-V (PULPino)	1 036 707	1 367 795	1 661 214
Saber CCA opt. (all)	**RISC-V (PULPino)**	**760 893**	**1 000 043**	**1 201 524**
Firesaber CCA [BMKV20]	ARM Cortex-M4	1 340 000	1 642 000	1 679 000
Firesaber CCA [KRSS19]	ARM Cortex-M4	1 448 776	1 786 930	1 853 339
Firesaber CCA baseline	RISC-V (PULPino)	3 427 099	4 215 630	4 328 885
Firesaber CCA opt. arithmetic	RISC-V (PULPino)	2 918 509	3 576 818	3 560 557
Firesaber CCA opt. sampling	RISC-V (PULPino)	1 790 609	2 235 737	2 633 554
Firesaber CCA opt. (all)	**RISC-V (PULPino)**	**1 300 272**	**1 622 818**	**1 898 051**

Table 4.10: Code size in bytes of NewHope, Kyber, and Saber with tightly coupled accelerators.

Algorithm	Device	Code Size
NewHope-512 CCA [KRSS19]	ARM Cortex-M4	11 000
NewHope-512 CCA baseline	RISC-V (PULPino)	17 658
NewHope-512 CCA opt.	**RISC-V (PULPino)**	**9 998**
NewHope-1024 CCA [KRSS19]	ARM Cortex-M4	12 176
NewHope-1024 CCA baseline	RISC-V (PULPino)	21 548
NewHope-1024 CCA opt.	**RISC-V (PULPino)**	**11 688**
Kyber-512 CCA [KRSS19]	ARM Cortex-M4	11 000
Kyber-512 CCA baseline	RISC-V (PULPino)	16 928
Kyber-512 CCA opt.	**RISC-V (PULPino)**	**12 532**
Kyber-768 CCA [KRSS19]	ARM Cortex-M4	11 400
Kyber-768 CCA baseline	RISC-V (PULPino)	17 266
Kyber-768 CCA opt.	**RISC-V (PULPino)**	**11 658**
Kyber-1024 CCA [KRSS19]	ARM Cortex-M4	12 424
Kyber-1024 CCA baseline	RISC-V (PULPino)	17 670
Kyber-1024 CCA opt.	**RISC-V (PULPino)**	**12 874**
Lightsaber CCA [KRSS19]	ARM Cortex-M4	44 916
Lightsaber CCA baseline	RISC-V (PULPino)	18 772
Lightsaber CCA opt.	**RISC-V (PULPino)**	**12 544**
Saber CCA [KRSS19]	ARM Cortex-M4	44 468
Saber CCA baseline	RISC-V (PULPino)	17 912
Saber CCA opt.	**RISC-V (PULPino)**	**11 802**
Firesaber CCA [KRSS19]	ARM Cortex-M4	44 184
Firesaber CCA baseline	RISC-V (PULPino)	17 794
Firesaber CCA opt.	**RISC-V (PULPino)**	**11 680**

complex operations are required. It should also be noted that the Saber implementations in [KRSS19] have a significantly larger memory consumption. The authors developed an auto-generated assembly code for a fast polynomial multiplication, which, however, has a large code size.

4.3.10 Experimental Resource Consumption Results

The enhanced RISC-V core with the proposed post-quantum instruction set extensions is in the following named as RISQ-V. The complete design is first synthesized for an FPGA prototype and then for an ASIC technology.

FPGA results. The Xilinx Zynq-7000 programmable SoC (Zedboard) was used for the FPGA evaluation. The resource utilization of the complete RISQ-V implementation and the costs for the single accelerators are provided in Table 4.11. The circuit size of RISQ-V is 9 058 LUTs and 1 268 registers larger than the size of the original PULPino. For this comparison, the FPU/FPR of the original PULPino is omitted as it is not necessarily required for the considered PQC algorithms.

Compared to the loosely coupled NTT design, the tightly coupled one requires a higher amount of LUTs but a lower number of registers (FFs). The increase in LUTs can be

Table 4.11: RISQ-V resource utilization for FPGA.

Complete Cores	LUT	FF	DSP	BRAM
PULPino original	15 248	9 569	6	32
RISQ-V	**24 306**	**10 837**	**18**	**32**

Single Accelerators	LUT	FF	DSP	BRAM
NTT accelerator Section 3.3 (loosely)	647	416	9	1
NTT and Modular Arithmetic Unit (tightly)	**2 908**	**170**	**9**	**0**
Keccak accelerator Section 3.3 (loosely), 0.5 cycle/round	11 049	4 097	0	0
Keccak low-area[a)] [BDH+20], 375 cycle/round	1 159	236	0	0
Keccak high-speed[a)] [BDH+20], 1 cycle/round	4 189	2 641	0	0
Keccak (tightly), 1 cycle/round	**3 847**	**0**	**0**	**0**
Binomial Sampling Unit (tightly)	**124**	**0**	**0**	**0**
MULT Unit with PQ-MAC (tightly)	**304**	**4**	**5**	**0**

[a)] VHDL design of [BDH+20] was synthesized for the same FPGA (Xilinx Zynq-7000). Resources do not include the costs for the connection to the system bus.

explained by the higher flexibility of the *NTT and Modular Arithmetic Unit*. The tightly coupled design supports the computation of two parallel butterfly operations (decimation-in-time and decimation-in-frequency) and vectorized modular arithmetic instead of only single decimation-in-time butterfly operations. Another advantage of the tightly coupled approach is that it does not need any further memory block to store the input/output data. Instead of using extra multipliers for the NTT post-processing, the proposed tightly coupled approach uses the same multipliers as the butterfly operation.

The tightly coupled Keccak implementation only uses combinatorial logic. No additional registers are used in this design as the state is stored in the FPR and GPR. The tight coupling saves all logic and registers that are used for buffering the input/output data. Moreover, no control logic for the Keccak absorption and squeezing phase is required. In contrast to the loosely coupled Keccak implementation, the tightly coupled variant calculates one round of the permutation function per clock cycle instead of two. This further decreases the resource utilization.

The two hardware accelerators of the execution stage have a negligible hardware overhead. While the *Binomial Sampling Unit* only requires 124 LUTs, the overhead for supporting the *pq.mac* operation is nearly eliminated by reusing the resources of the *MULT Unit*.

ASIC results. The same UMC 65 nm technology as in Section 3.3.3 is used for the ASIC design. The main objective of this chapter is to achieve a low energy consumption. Therefore, a low leakage standard cell library with a high threshold voltage but lower speed is chosen. Table 4.12 shows the number of logic cells and the area consumption of the original PULPino and RISQ-V. The results are provided for the final ASIC designs (after synthesis, floorplanning, clock tree synthesis, placement, and routing). Compared

Table 4.12: RISQ-V area of final ASIC layout (UMC 65 nm).

	Cell Count	Combinat.	Sequent.	Buffer+Inv.	Clk-Gate	Memory
PULPino orig.	42 115	78 373 μm^2 (54 kGE)	92 261 μm^2 (64 kGE)	20 534 μm^2 (14 kGE)	365 μm^2 (0.25 kGE)	669 345 μm^2 (465 kGE)
RISQ-V	**68 853**	**148 941 μm^2** (103 kGE)	**102 203 μm^2** (71 kGE)	**27 186 μm^2** (19 kGE)	**346 μm^2** (0.24 kGE)	**669 345 μm^2** (465 kGE)

to the original design, the cell area of RISQ-V increased by 70 568 μm^2 for the combinatorial logic and by 9 942 μm^2 for the sequential logic. However, this increase does not significantly impact the overall area because the memory blocks are the largest parts of both designs. The maximum clock frequency for a typical operating condition was reduced from 75.67 MHz to 46.33 MHz as the *Modular Arithmetic Unit* has a relatively long critical path. A solution to break this critical path could be to add pipeline registers within the modular multipliers. As the achieved frequency is acceptable for most embedded applications, these registers are omitted and have been left as future work.

The simulated measurements for the power and energy consumption were performed at a frequency of 10 MHz, a nominal supply voltage of 1.2 V, and a temperature of 25°C. In order to obtain realistic results, the dynamic post-layout power consumption is extracted with an analysis of the switching activity of the circuit. For this purpose, a Toggle Count Format (TCF) file was generated with a back annotated post-layout simulation using Cadence's Incisive Enterprise Simulator. Subsequently, the power consumption was calculated using the Cadence power analyzer Joules. The results of the power measurements of the original PULPino and the optimized RISQ-V implementation are summarized in Table 4.13. The leakage power belongs to the category of static power consumption. Due to the higher area consumption, the leakage power is marginally higher for RISQ-V compared to the original design. Interestingly, the total power consumption depends mainly on the applied scheme but not on the security level. This is probably the case because similar operations are used for all security levels. Due to the shorter execution time, the energy consumption is significantly lower when using the tightly coupled accelerators.

Figure 4.11 illustrates the averaged power consumption of the optimized RISQ-V implementation when NewHope-512 is executed. The placement of the accelerators can be viewed in Figure 5.8 (Chapter 5). However, the averaged power dissipation is not conspicuous for the accelerators. In particular, the two large memory blocks in the left corners of the core area and the clock pad on the right boundary have a large power consumption. The remaining circuit has a relatively moderate power consumption. The standard cells for the peripherals consume nearly no power at the execution of the PQC algorithms.

4.4 Instruction Set Extensions for the PQC Scheme SIKE

This section illustrates that other PQC categories than lattice-based cryptography also benefit from a tightly coupled approach. Similar methods as in the previous section can

Table 4.13: RISQ-V power and energy results of the final ASIC design (UMC 65 nm).

	Leakage	Internal	Switching	Tot. Power	Cycles	Energy
NewHope-512 CCA (baseline)	3.50e-05 W	1.42e-03 W	3.50e-04 W	1.81e-03 W	5 615 725	1 016.45 μJ
NewHope-512 CCA (opt.)	3.61e-05 W	1.49e-03 W	4.55e-04 W	1.98e-03 W	522 687	**103.49 μJ**
NewHope-1024 CCA (baseline)	3.50e-05 W	1.41e-03 W	3.48e-04 W	1.79e-03 W	11 289 402	2 020.80 μJ
NewHope-1024 CCA (opt.)	3.61e-05 W	1.49e-03 W	4.54e-04 W	1.98e-03 W	991 801	**196.38 μJ**
Kyber-512 CCA (baseline)	3.50e-05 W	1.46e-03 W	3.62e-04 W	1.85e-03 W	4 210 556	778.95 μJ
Kyber-512 CCA (opt.)	3.61e-05 W	1.54e-03 W	5.03e-04 W	2.07e-03 W	548 119	**113.46 μJ**
Kyber-768 CCA (baseline)	3.50e-05 W	1.46e-03 W	3.61e-04 W	1.86e-03 W	7 302 385	1 358.24 μJ
Kyber-768 CCA (opt.)	3.61e-05 W	1.55e-03 W	5.05e-04 W	2.09e-03 W	940 008	**196.46 μJ**
Kyber-1024 CCA (baseline)	3.50e-05 W	1.47e-03 W	3.61e-04 W	1.86e-03 W	11 352 622	2 111.59 μJ
Kyber-1024 CCA (opt.)	3.61e-05 W	1.55e-03 W	5.21e-04 W	2.11e-03 W	1 183 372	**249.69 μJ**
Lightsaber CCA (baseline)	3.50e-05 W	1.56e-03 W	3.58e-04 W	1.95e-03 W	4 113 463	802.13 μJ
Lightsaber CCA (opt.)	3.61e-05 W	1.71e-03 W	4.86e-04 W	2.24e-03 W	1 551 010	**347.43 μJ**
Saber CCA (baseline)	3.50e-05 W	1.56e-03 W	3.60e-04 W	1.95e-03 W	7 645 196	1 490.81 μJ
Saber CCA (opt.)	3.61e-05 W	1.72e-03 W	4.86e-04 W	2.24e-03 W	2 962 792	**663.67 μJ**
Firesaber CCA (baseline)	3.50e-05 W	1.56e-03 W	3.60e-04 W	1.96e-03 W	11 971 708	2 346.45 μJ
Firesaber CCA (opt.)	3.61e-05 W	1.72e-03 W	4.88e-04 W	2.25e-03 W	4 821 233	**1084.78 μJ**

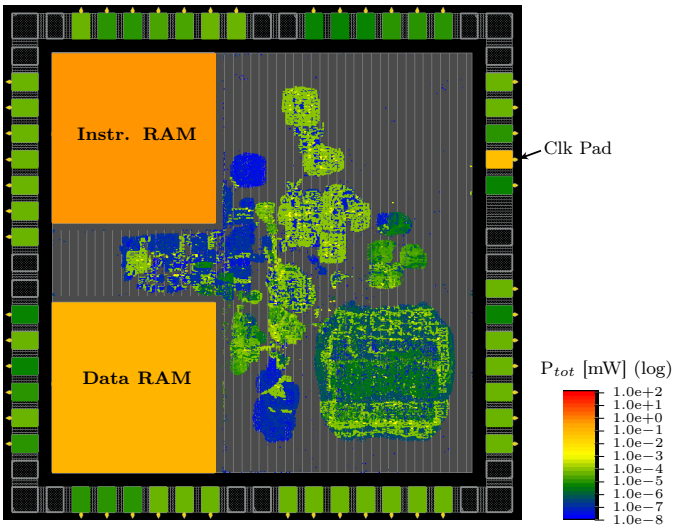


Figure 4.11: Power distribution optimized RISQ-V ASIC implementation (averaged for NewHope-512).

be applied to accelerate, for instance, the isogeny-based scheme SIKE.

4.4.1 Bottlenecks of Isogeny-Based Cryptography

An isogeny $\phi : E_A \leftarrow E_B$ is a structure-preserving map between two elliptic curves E_A and E_B over finite fields [JDF11]. The security of SIKE relies on the hardness of solving the isogeny map between supersingular elliptic curves over a quadratic extension field $\mathbb{F}_q = \mathbb{F}_{p^2}$ with $p = 2^n \cdot 3^m \pm 1$ and the public integers n, m [J+20]. The finite field elements of $\mathbb{F}_{p^2}$ can be represented as $a = a_0 + a_1 \cdot i$ with $a_0, a_1 \in \mathbb{F}_p$ and i being the complex root of unity. All operations used in SIKE can be reduced on the lowest abstraction layer to $\mathbb{F}_p$ arithmetic. The field multiplication in $\mathbb{F}_p$ poses a particular performance bottleneck, but field additions and subtractions are also costly. The actual sizes of the field elements depend on the chosen parameter set. The evaluation of this work is based on SIKEp434 with operand sizes of 434-bit. The same principles can be applied to other parameter sets.

4.4.2 System Integration of SIKE Accelerators

This section discusses a system integration strategy for tightly coupled field multiplication, addition, and subtraction accelerators. The deployed finite field multiplier is based on the Montgomery algorithm with redundant number system as presented in [RM19]. The accelerator for the field addition is based on an optimized adder circuit with conditional subtraction to realize the reduction. The field subtraction follows a similar principle. Further details can be found in the publication related to this section [RFS20].

Figure 4.12 presents the tightly coupled accelerator integration. Similar as in the previous section, the accelerators can access all operands in parallel from the registers. The first operand for the finite field accelerators is stored in the registers `f0`–`f13` and the second operand in `f14`–`f27`. The control signals of this architecture are `mul`, `add`, and `sub`. The result is stored back to the registers `f0`–`f13`. A single field multiplication requires 22 cycles and a field addition or subtraction 7 cycles assuming the operands to be available within the registers.

4.4.3 Experimental Results of SIKE System Design

The Xilinx Zynq-7000 programmable SoC (Zedboard) was used to evaluate the performance and area of the proposed approach.

Performance evaluation. Table 4.14 provides a comparison with existing assembly optimized SIKEp434 implementations on ARM. The RISC-V baseline implementation uses the source code provided in the NIST submission [J+20]. The results show that the proposed design has a significantly lower cycle count than the assembly optimized ARM Cortex-M4 implementation in [SAJA21]. It even has a smaller cycle count than the implementations on the much more powerful 64-bit ARM processors (Cortex-A55 and Cortex-A75). The PULPino platform achieves a frequency of around 25 MHz for the

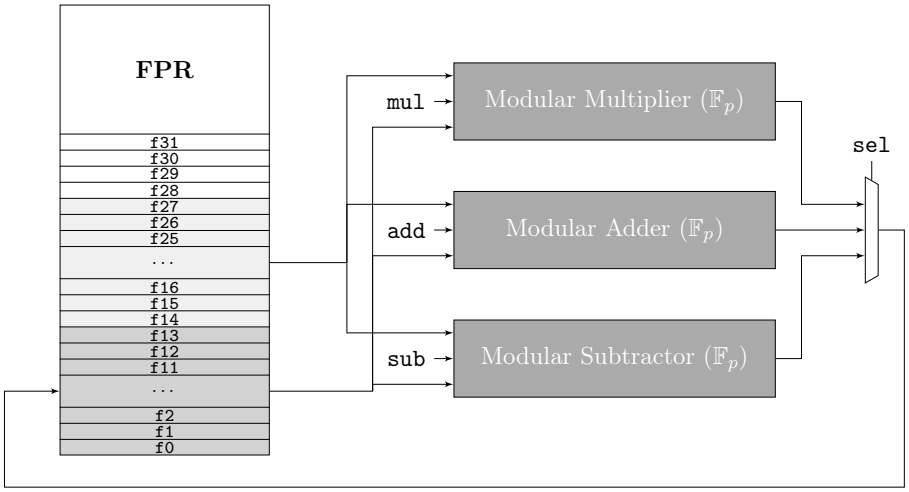


Figure 4.12: SIKE accelerator coupling.

Table 4.14: Cycle count of SIKEp434 with tightly coupled finite field accelerator (total indicates cycles for Encaps+Decaps).

Work	Platform	CC			CC $\times 10^6$			
		$\mathbb{F}_p$ mul	$\mathbb{F}_p$ add	$\mathbb{F}_p$ sub	KeyGen	Encaps	Decaps	Total
[SAJA21]	ARM M4	1011	253	207	54.0	87.0	94.0	181.0
[SSJA20]	ARM A55	602	74	67	27.2	44.8	47.7	92.5
[SSJA20]	ARM A75	561	34	28	22.8	37.5	40.0	77.5
This work baseline	RISC-V	7 111	341	656	271.5	444.4	474.3	918.7
This work opt.	**RISC-V**	**96**	**79**	**79**	**10.7**	**17.8**	**19.1**	**36.9**

target FPGA. The maximum frequency for the finite field accelerators is 150 MHz. They are, therefore, not part of the critical path. Nevertheless, the achieved frequencies are several times smaller than most ARM Cortex-A/M platforms achieve. But when integrating the proposed design at a small technology node on ASIC, the operating frequency could be significantly increased (e.g., up to 938 MHz for the same processor was reported in [SRP+18]).

Resource consumption. Table 4.15 summarizes the resource utilization of the developed design. The amount of LUT/FF/DSP increased by factors 1.45/1.76/11.33 when the finite field arithmetic is integrated.

Table 4.15: Resource utilization of SIKE hardware/software codesign with tightly coupled accelerators.

Design	LUT	FF	DSP	BRAM
PULPino original	15 248	9 569	6	32
PULPino with accelerators	22 166	16 882	68	32

4.5 Summary

This chapter illustrated the great advantages of integrating the accelerators directly into the main processor. Usually, data is first loaded from the main processor and then distributed over a bus interface to the loosely coupled accelerator. In order to circumvent a part of this overhead, this chapter first presented tightly coupled accelerators for LAC. The ternary hardware multiplier used for NTRU was extended to support the negative wrapped convolution of LAC and was directly integrated into a RISC-V core. The proposed architecture supports multiple polynomial lengths due to a software-based splitting. Further, a modular reduction unit, SHA256 accelerator, and Chien search accelerator were integrated into the main processor.

While the hardware/software codesign of LAC already achieves a good performance, it still does not exploit the full potential of a tightly coupled approach. For the use-cases NewHope, Kyber, and Saber, it was demonstrated how memory accesses can be systematically reduced and how system resources can be reused. It was shown that keeping coefficients during the NTT computation within the register file significantly reduces time-consuming memory accesses. The proposed versatile NTT and modular arithmetic accelerator and the applied SIMD principles significantly boosted the performance. For non-NTT based schemes as Saber, a vectorized modular MAC function was developed. It comes at almost no area overhead as it reuses the multipliers of the existing RISC-V multiplication unit. Another essential building block presented in this chapter is the tightly coupled Keccak accelerator. As the register sets of the RISC-V core are reused, it requires only combinatorial logic. It was shown that an extreme speedup is achieved when keeping the state for the complete sampling process within the register sets. A binomial sampling accelerator complements the proposed design.

In comparison to the loosely coupled designs of the previous chapter, it was illustrated that the tightly coupled approach can be faster while decreasing the overall area overhead. Tightly coupled accelerators do not require any control circuit as instruction set extensions are used to control them. This does not only decrease the circuit size but also leads to a high flexibility. The experiments verified that tightly coupled accelerators and instruction set extensions for lattice-based cryptography significantly improve performance and energy characteristics. Such improvements are also possible for other PQC schemes. The results for the isogeny-based scheme SIKE show that resource-constrained open-source RISC-V platforms, in many ways, can compete with commercial ARM microcontrollers when hardware acceleration is applied.

In conclusion, tightly coupled PQC approaches seem to be more appealing for hardware/software codesigns when flexibility and area are relevant. The main reason is that the large operand (e.g., polynomial) sizes of PQC involve a high communication overhead between main processor and loosely coupled coprocessor. This decreases the achieved performance gain of a design with loosely coupled accelerators. In order to counteract this problem, larger parts of the algorithm must be computed directly within the accelerator. Thus, taking a loss of flexibility into account.

5 Generalization of the NTT Algorithm and Masking of Non-Linear Operations

Implementation attacks are highly critical because they can break cryptographic systems even if they are considered as mathematically secure. In order to prevent such attacks, cryptographic implementations must integrate countermeasures. This chapter introduces measures to protect PQC hardware/software codesigns, like those presented in the last chapters, against implementation attacks. The related content presented in this chapter was published in [FVBBR+21]. Moreover, the previous chapters have shown that accelerated polynomial multiplications using the NTT achieve a particular good speedup with a moderate increase in area. Therefore, this chapter also analyzes how the NTT can be applied to schemes that natively do not support it. The proposed method contains parts of the author's publications [FSS20b, FVBBR+21].

5.1 Introduction of Side-Channel Protection Mechanisms

The protection of cryptographic algorithms against implementation attacks is highly relevant for applied cryptography. Fault attacks or SCA are able to retrieve secret information during the execution of mathematically secure algorithms. Countless attacks have been demonstrated in recent years. Protecting implementations against all of them is an extremely difficult, maybe unsolvable, task. It is therefore of high relevance to identify particularly critical attack scenarios. One highly critical attack type is the category of timing attacks. They are applicable for most applications. Current PQC implementations address this topic by ensuring that the algorithm's execution time does not depend on secret data. Simple Power Analysis (SPA) and Differential Power Analysis (DPA) are two attack types that are more difficult to prevent. They do need access to the physical device, which is possible for many applications. SPA relies on single or few measurements. DPA takes a large number of measurements to reduce the influence of noise and to allow the identification of extremely small secret-dependent correlations [KJJ99]. Masking is a typical countermeasure to protect implementations against DPA [CJRR99]. The technique splits every secret-dependent variable randomly into multiple parts—also known as shares. The cryptographic algorithm is then executed on these shares separately. The recombination of the shares at the end of the critical operation returns the desired result. Due to the random splitting, the power consumption does not depend on the original secret anymore.

Related works. Different works analyzed countermeasures to protect PQC implementations against DPA. The first masking method for the basic LWE protocol was presented in [RRVV15] and later refined in [RdCR+16]. Several digital signature schemes were analyzed regarding side-channel protection. The signature scheme GLP was analyzed in [BBE+18], and blinding countermeasures for BLISS were proposed in [Saa18]. The NIST second-round signature candidate qTESLA was masked in [GR19] and the finalist Dilithium in [MGTF19]. In the category of PKE/KEM, a predecessor of the NewHope NIST submission was masked in [OSPG18]. Further, a masked ARM Cortex-M4 implementation of the NIST finalist Saber was proposed in [BDK+21]. The authors of [BGR+21] published, concurrently to the work of the thesis author in [FVBBR+21], a masked Kyber implementation. The publication [FVBBR+21] is the basis for the content of this chapter. Further works discuss DPA protection mechanisms for the binomial sampling [SPOG19] and polynomial multiplication [HP21].

Despite the remarkable progress due to these prior works, DPA protection for PQC is a research area with numerous open questions. Previous works focused on protected software implementations. It remains unclear how DPA countermeasures can be efficiently designed for pure hardware or hardware/software codesign solutions. DPA-protected implementations are significantly slower than unprotected implementations and can greatly

benefit from hardware acceleration. Hardware circuits have not only the advantage of a high performance but also ensure a controlled execution, which is particularly important to prevent side-channel leakages. A clear picture of different implementation strategies and the related costs for the protection is still not available for the lattice-based PKE/KEM finalists Kyber, Saber, and NTRU. NIST indicated that only one of the lattice-based finalists would be standardized. NTRU has the advantage of a long history and a low risk of new patent issues but is slower than the other two competitors [AASA+20]. The difference between Kyber and Saber in terms of DPA protection is less obvious. A decision between Kyber, Saber, and NTRU as a winner of the NIST competition might be influenced by the suitability for SCA protection.

Contributions. This chapter presents masked implementations of Kyber and Saber on a RISC-V platform to provide an analysis of the DPA-protection overhead. This chapter further proposes hardware accelerators for critical operations of the masked designs. Since performance bottlenecks are even more pronounced in a masked setting, acceleration becomes more important. Among others, the NTT accelerator of the previous chapter is extended to support Saber in order to further improve the performance of both schemes.

The contributions of this chapter can be summarized as:

- Investigating a generic NTT multiplier (featuring positive/negative wrapped convolutions, incomplete NTTs, and prime lifts) to support a variety of PKE/KEM and signature schemes;
- Proposing masked hardware accelerators for hashing, binomial sampling, A2B/B2A conversions, and compression;
- Presenting measures towards secure system design with share separation;
- Developing the first masked hardware/software codesigns for PQC with Saber and Kyber as a case study (one of the first masked Kyber implementations in general);
- Achieving new cycle count records for Saber and Kyber on a RISC-V platform with instruction set extensions and accelerators.

5.2 Preliminaries Masking

This section discusses masking—a provable secure method to protect lattice-based PQC against DPA. In the remainder of this chapter, the subscript is used to access an element of an array/matrix and the superscript to access a share of a masked variable. As an example, a^i accesses the i-th share of the variable $\mathbf{a}^{\{0:n-1\}}$ with n shares. In the following description, two shares are considered for a first-order secure masking. The secret variable x is randomly split into two parts, either using the arithmetic sharing $x = a^0 + a^1$ or Boolean sharing $x = b^0 \oplus b^1$. A2B and B2A are conversion methods to securely switch between both sharing types. Let $r = a^1 = b^1$ be a random mask, then the two equations

Algorithm 15: A2B [CGV14]

Input: $\mathbf{x}^{\{0:1\}} = (a^0, a^1)$ s.t. $x = a^0 + a^1 \mod 2^k$
Result: $\mathbf{x}^{\{0:1\}} = (b^0, b^1)$ s.t. $x = b^0 \oplus b^1$

1 $r^0, r^1 \xleftarrow{\$} \mathbb{Z}_{2^k}$
2 $\mathbf{b}_1^{\{0:1\}} \leftarrow (a^0 \oplus r^0, r^0)$, $\mathbf{b}_2^{\{0:1\}} \leftarrow (a^1 \oplus r^1, r^1)$
3 $\mathbf{x}^{\{0:1\}} \leftarrow \textsc{SecAdd}(\mathbf{b}_1^{\{0:1\}}, \mathbf{b}_2^{\{0:1\}})$

Algorithm 16: A2B$_q$ [BBE+18]

Input: $\mathbf{x}^{\{0:1\}} = (a^0, a^1)$ s.t. $x = a^0 + a^1 \mod q$
Result: $\mathbf{x}^{\{0:1\}} = (b^0, b^1)$ s.t. $x = b^0 \oplus b^1$

1 $r^0, r^1 \xleftarrow{\$} \mathbb{Z}_q$
2 $\mathbf{b}_1^{\{0:1\}} \leftarrow (a^0 \oplus r^0, r^0)$, $\mathbf{b}_2^{\{0:1\}} \leftarrow (a^1 \oplus r^1, r^1)$
3 $\mathbf{x}^{\{0:1\}} \leftarrow \textsc{SecAdd}_q(\mathbf{b}_1^{\{0:1\}}, \mathbf{b}_2^{\{0:1\}})$

$b^0 = (a^0 + r) \oplus r$ and $a^0 = (b^0 \oplus r) - r$ describe how to switch between the two types. The problem is that these equations require the recombination of the secret variable x.

The first mathematically secure conversion methods were proposed in [Gou01]. While the presented B2A conversion is already fast in software, the A2B conversion was further optimized using table lookups in [CT03, Deb12]. The drawback of these conversion algorithms is that they are not well suited for non-power-of-two moduli and, therefore, not for Kyber. Some workarounds were proposed in [OSPG18, BBE+18, SPOG19]. In this work, the approach of [BBE+18] is used as it supports non-power-of-two moduli conversions (denoted as B2A$_q$/A2B$_q$), it is extensible to higher-order masking, and it is suitable for hardware. Their approach is based on the secure masked addition algorithm SecAdd [CGV14].

The secure addition is defined as $\mathbf{s}^{\{0:1\}} = \textsc{SecAdd}(\mathbf{x}^{\{0:1\}}, \mathbf{y}^{\{0:1\}})$ such that $(s^0 \oplus s^1) = (x^0 \oplus x^1) + (y^0 \oplus y^1)$. Similarly, the secure addition modular q is defined as $\mathbf{s}^{\{0:1\}} = \textsc{SecAdd}_q(\mathbf{x}^{\{0:1\}}, \mathbf{y}^{\{0:1\}})$ such that $(s^0 \oplus s^1) = (x^0 \oplus x^1) + (y^0 \oplus y^1) \mod q$. The $\textsc{SecAdd}_q$ operation can be realized using two conventional SecAdd operation. The first operation computes $(s^0 \oplus s^1) = (x^0 \oplus x^1) + (y^0 \oplus y^1)$ and the second one securely adds $-q$ to the result. Depending on the sign bit either the result of the first or second addition is selected. For more details, please refer to the related publication of this section [FVBBR+21].

A2B/A2B$_q$. The A2B and A2B$_q$ conversions are provided in Algorithms 15 and 16. Computing A2B and A2B$_q$ with SecAdd is straightforward. Each input share is masked with a different random variable, and the result is then assigned to the input of SecAdd. The SecAdd operation already returns the desired Boolean masking as output.

B2A/B2A$_q$. The B2A and B2A$_q$ conversions are provided in Algorithms 17 and 18. The two algorithms B2A and B2A$_q$ are very similar. They start with sampling two random variables, while the first random variable is directly assigned to the first output share a^0. The second random variable is then used to mask $-a^0$ in order to securely compute $a^1 = (b^0 \oplus b^1) - a^0$ using SecAdd.

Glitch-resistant masking. Glitches, caused by different propagation delays within the circuit, can lead to severe side-channel leakages of theoretically secure systems. This is

Algorithm 17: B2A [CGV14]

Input: $\mathbf{x}^{\{0:1\}} = (b^0, b^1)$ s.t. $x = b^0 \oplus b^1$
Result: $\mathbf{x}^{\{0:1\}} = (a^0, a^1)$ s.t. $x = a^0 + a^1 \mod 2^k$

1 $a^0 \xleftarrow{\$} \mathbb{Z}_{2^k}$
2 $r \xleftarrow{\$} \mathbb{Z}_{2^k}$
3 $\mathbf{b}_1^{\{0:1\}} \leftarrow ((2^k - a^0) \oplus r, r)$
4 $\mathbf{b}_2^{\{0:1\}} \leftarrow \textsc{SecAdd}(\mathbf{x}^{\{0:1\}}, \mathbf{b}_1^{\{0:1\}})$
5 $\mathbf{x}^{\{0:1\}} \leftarrow (a^0, b_2^0 \oplus b_2^1)$

Algorithm 18: B2A$_q$ [BBE+18]

Input: $\mathbf{x}^{\{0:1\}} = (b^0, b^1)$ s.t. $x = b^0 \oplus b^1$
Result: $\mathbf{x}^{\{0:1\}} = (a^0, a^1)$ s.t. $x = a^0 + a^1 \mod q$

1 $a^0 \xleftarrow{\$} \mathbb{Z}_q$
2 $r \xleftarrow{\$} \mathbb{Z}_{2^w}$
3 $\mathbf{b}_1^{\{0:1\}} \leftarrow ((q - a^0) \oplus r, r)$
4 $\mathbf{b}_2^{\{0:1\}} \leftarrow \textsc{SecAdd}_q(\mathbf{x}^{\{0:1\}}, \mathbf{b}_1^{\{0:1\}})$
5 $\mathbf{x}^{\{0:1\}} \leftarrow (a^0, b_2^0 \oplus b_2^1)$

particularly critical for non-linear operations as they process data from multiple shares. Threshold Implementation (TI) is an effective method to prevent leakages caused by glitches [NRR06]. The concept is based on multi-party computation with the following properties: correctness (the result after the computations remains correct), non-completeness (the partial functions and computations are at least independent of one input share), and uniformity (input and output are uniformly distributed). For a security order of d and a function of algebraic degree t, the number of input shares must be $s \geq td + 1$ [RBN+15]. Thus, first-order TIs require a minimum of three input shares for non-linear functions.

Another glitch-resistant method is Domain-Oriented Masking (DOM) [GMK16]. DOM assigns each share of a variable to a domain and strictly keeps each domain separated from the other domains. At non-linear operations, domain crossing operations use fresh randomness to preserve the independence of the domains. Additional registers prevent signal propagation and glitches. The AND-gate proposed in [GMK16] is an important example of DOM.

5.3 Masking PKE/KEM

Running a single key exchange is not vulnerable against DPA as only one measurement can be taken by an attacker. DPA is more critical in a PKE setting. While the key generation is still executed only once, the encryption and decryption are repeated multiple times. The decryption uses the long-term secret key, which must be protected against DPA. In order to avoid CCA, a secret-dependent re-encryption is performed during the decryption (decapsulation). Hence, the decryption and re-encryption are vulnerable and must be masked. Note that further countermeasures are required to protect against SPA attacks that analyze the side-channel trace horizontally [CFG+10]. This type of attack uses information from multiple time instances of an algorithm. Shuffling and the random insertion of dummy operations are typical countermeasures against this kind of attack. A detailed analysis has been left as future work.

The CCA secure decapsulation requires, in addition to the decryption operation (Algorithm 6, Chapter 2), a re-encryption (Algorithm 5, Chapter 2) with a consecutive equality

Table 5.1: Operations of LWE-based schemes in a masked setting.

Operation	Type	Sharing
Ring arithmetic $(\cdot, +, -)$	linear	arithmetic sharing
Randomness generation for sampling (XOF/Keccak)	non-linear	Boolean sharing
Binomial sampling (ψ_η)	non-linear	Boolean sharing
A2B/B2A (Saber), A2B$_q$/B2A$_q$ (Kyber)	non-linear	both
Decoding/compression ($\texttt{compress}_q(x, d)$)	non-linear	both
Ciphertext comparison ($\stackrel{?}{=}$)	non-linear [a)]	Boolean sharing

[a)] Non-linear ciphertext comparison is not critical after hash computations.

check that verifies that a malicious party did not modify the ciphertext. Table 5.1 summarizes the main operations required for the masked CCA secure decapsulation of Kyber and Saber.

Linear operations, such as ring arithmetic, can be performed on each share separately. As a result, the execution time doubles, and it becomes even more important to improve the performance of these operations. An optimized hardware accelerator for the linear arithmetic in Kyber and Saber is presented in Section 5.4.

Non-linear operations are more critical to mask as they need access to multiple shares simultaneously. Partly non-linear operations are Keccak, binomial sampling, A2B/B2A (A2B$_q$/B2A$_q$) conversions, decoding/compression, and the ciphertext comparison after the re-encryption. Accelerators for the non-linear operations are provided in Section 5.5. Section 5.5.1 presents a masked version of Keccak to securely generate uniform randomness for the sampling process. Section 5.5.2 proposes generic hardware accelerators for the masked binomial sampling. Finally, Section 5.5.3 presents the non-linear secure addition (SecAdd) required for the A2B/B2A (A2B$_q$/B2A$_q$) conversion, as discussed in the previous section. Apart from SecAdd, all operations for these conversions can be efficiently performed in software. The masked decoding/compression and ciphertext comparison require the same accelerators as the other non-linear operations. They are briefly discussed in the following paragraphs.

In a masked setting, the decoding/compression $\texttt{compress}_q(a, d) = \lceil (2^d/q) \cdot a \rfloor \mod 2^d$ takes an arithmetic sharing of a polynomial as input. The required division by a power-of-two modulus corresponds to a simple shift, which can be efficiently performed with a Boolean sharing. Hence, an A2B conversion can be applied in this context. In the related publication of this section [FVBBR+21], an efficient method was proposed that is also applicable for prime moduli. The method exploits that $\lceil (2^d/q) \cdot a \rfloor \mod 2^d$ tolerates an approximate quotient and, therefore, a bounded failure. Hence, it is possible to perform the decoding/compression operation with a finite precision on both shares individually without having inaccuracies at the recombination of both shares. The complete decoding/compression operation requires only simple arithmetic and an A2B$_q$ conversion. As this conversion type is already covered, the masked decoding/compression is not further discussed in this thesis. For more details, please refer to the publication [FVBBR+21].

The masked ciphertext comparison check applied in this work is based on the method proposed in [BDK+21], which uses concepts of [OSPG18]. The equality check still has

Table 5.2: NTT parameters of several lattice-based algorithms.

Scheme	n	q	$\phi(x)$	NTT-based	$\lceil log_2(q') \rceil$
NewHope-512/1024	512/1024	12289	$x^n + 1$	yes	14
Kyber	256	3329	$x^n + 1$	yes	12
Dilithium	256	8380417	$x^n + 1$	yes	23
Falcon-512/1024	512/1024	12289	$x^n + 1$	yes	14
Saber	256	8192	$x^n + 1$	no	34
ntruhps2048509	509	2048	$x^n - 1$	no	31
ntruhps2048677	677	2048	$x^n - 1$	no	32
ntruhps4096821	821	4096	$x^n - 1$	no	34
ntruhrss701	701	8192	$x^n - 1$	no	36
LAC-128	512	251	$x^n + 1$	no	25
LAC-192/256	1024	251	$x^n + 1$	no	26

sensitive inputs, and recombining the re-encrypted ciphertext must be prevented. In order to avoid leakages, first two hash values $\mathcal{H}(\overline{\boldsymbol{u}} \oplus \overline{\boldsymbol{u}'}^0 \| \overline{v} \oplus \overline{v}'^0)$ and $\mathcal{H}(\overline{\boldsymbol{u}}'^1 \| \overline{v}'^1)$ are computed using the ciphertext $ct = (\overline{\boldsymbol{u}}, \overline{v})$ and the re-encrypted ciphertext $ct' = (\overline{\boldsymbol{u}'}^{\{0,1\}}, \overline{\boldsymbol{v}'}^{\{0,1\}})$. The comparison is then performed on these hash values, which contain no sensitive information anymore. A Keccak accelerator can be used to improve the performance of this operation. The masked comparison is less critical than the other non-linear operations because the hash operations require only one share, respectively. All subsequent operations do not contain sensitive data.

5.4 Accelerators for Linear Operations

The ring arithmetic accelerators proposed in the previous chapters can be directly applied in a masked setting. Linear operations are consecutively performed with each share. Therefore, eliminating bottlenecks becomes more relevant in a masked setting. The presented small Karatsuba/Toom–Cook accelerator for Saber of Section 4.3.4 is still not optimal. This section investigates the application of the NTT for Saber to achieve a comparably good speedup at the ring arithmetic as for Kyber.

5.4.1 Increasing the Flexibility of NTT

Table 5.2 summarizes parameters for the polynomial arithmetic of several lattice-based algorithms. While some schemes are tailored for an efficient NTT usage, others choose a prime not suitable for a direct application of the NTT.

NTT with prime lift. The original prime q can be lifted to any "NTT-friendly" prime $q' > n \cdot q^2$. This can be done because the result of a polynomial multiplication without reductions has coefficients not larger than $n \cdot q^2$. If q' is set sufficiently large, precision errors caused by the modular arithmetic are avoided [PNPM15]. After the polynomial multiplication with the lifted prime, the coefficients can be reduced by the original prime

q. When using signed arithmetic, the maximum absolute value of the coefficients during the computation of the polynomial multiplication is $n \cdot q^2/4$ when the coefficients are represented in $[-q/2, q/2)$. Some schemes always multiply large polynomials with small polynomials sampled from the error distribution, allowing to decrease the value of q' if signed arithmetic is used. However, as unsigned arithmetic is more practical for hardware circuits, the rule $q' > n \cdot q^2$ is applied in this work.

Reusing the tightly coupled accelerator for Saber. While the moduli of NewHope and Kyber fit in one halfword (16-bit), Saber requires a 34-bit modulus to apply the NTT with unsigned arithmetic (see Table 5.2). The polynomial multiplication of Saber can be computed with several smaller moduli suitable for the 16-bit NTT hardware architecture of Section 4.3.2. The final result can then be retrieved applying the Chinese Remainder Theorem (CRT).

Chinese Remainder Theorem. Let $q'_0, q'_1, \ldots, q'_{k-1}$ be pairwise coprime, $q' = \prod_{i=0}^{k-1} q'_i$, and $c_0, c_1, \ldots, c_{k-1}$ integers with $0 \leq c_i < q'_i$. Then, the system of linear congruences

$$\begin{aligned} x &\equiv c_0 \mod q'_0 \\ x &\equiv c_1 \mod q'_1 \\ &\ldots \\ x &\equiv c_{k-1} \mod q'_{k-1} \end{aligned}$$

has a unique solution solving all congruences simultaneously [Sch13]. The CRT problem can be solved using Gauss's algorithm with

$$x \equiv c_0 d_0 + c_1 d_1 + \cdots + c_{k-1} d_{k-1} \mod q' \ , \tag{5.1}$$

where $d_i = (q'/q'_i)^{-1} \mod q'_i$ [DI 10]. As d_i can be precomputed, the results of the polynomial multiplication with smaller primes can be combined by multiplications, additions, and one reduction by q' as in Equation 5.1.

Parameters for Saber with tightly coupled accelerator. Lifting to a higher prime only works if no reduction errors are introduced during the convolution. Negacyclic convolutions involve negative intermediate results that lead to an erroneous output when reduced by q'. These reductions can be avoided using signed arithmetic. For unsigned arithmetic, polynomial multiplications with polynomials of length $n' = 2n$, zero-padding, and consecutive polynomial reduction by $\phi(x)$ can be used. Positive wrapped convolutions can still be realized with an NTT of length $n' = n$.

For the experiments of the polynomial multiplication in Saber with the tightly coupled NTT accelerator, this work chooses the parameters $n' = 512$, $q' = q'_0 \cdot q'_1 \cdot q'_2$ with $q'_0 = 12289$, $q'_1 = 13313$, and $q'_2 = 15361$. These primes are suitable for the tightly coupled accelerator and are the smallest ones fulfilling $q \equiv 1 \mod 2n$ such that the corresponding n-th root of unity exists [ADPS16b] and the NTT is applicable. Moreover, these primes fulfill the condition $q' > n \cdot q^2$.

Results for Saber with tightly coupled accelerator. The complete polynomial multiplication of one share with zero-padding requires 36 102 cycles on the RISC-V platform with the NTT accelerator. In total, 108 306 cycles are necessary to compute the polynomial multiplication for all three shares. Further cycles are then required for the recombination of the intermediate results applying the CRT. However, using the proposed accelerator for Karatsuba/Toom–Cook turns out to be more efficient (71 349 cycles).

Parameters for a generic NTT. In order to avoid the costly CRT decomposition, a flexible NTT accelerator is designed that directly supports larger prime values. All NTT-based schemes of Table 5.2 have primes smaller than or equal to 23-bit. In this chapter, a flexible Montgomery multiplier for any prime up to 24-bit is developed to cover all of them. For algorithms that are not NTT-based, a lifted prime q' must be found that covers the remaining algorithms. To allow an efficient reduction, the Solinas prime $q' = 2^{39} - 2^{12} + 1$ is chosen. For this prime, the condition $q' \equiv 1 \mod 2n$ holds, and the n-th as well as the $2n$-th root of unity exists (e.g., for $n \in [256, 512, 1024, 2048]$).

Incomplete NTT. The prime q is usually chosen such that $\phi(x)$ can be factored into $n = 2^k$ linear terms $\phi(x) = \prod_{i=0}^{n-1} \phi_i(x) \mod q$ [LS19]. This allows the full application of the NTT, and the basecase multiplication of two transformed polynomials corresponds to a simple coefficient-wise multiplication. The concept of the incomplete NTT for lattice-based cryptography was first proposed in [LS19], and a similar concept was later adopted in the second-round Kyber specification. Kyber reduced its prime value (consequently key and ciphertext sizes) and chose a value where the n-th root of unity exists but not the $2n$-th root of unity. This prevents the application of a full NTT, and only $l - 1$ layers of the NTT are computed, resulting in $n/2$ polynomials of degree two. More precisely, the cyclic polynomial is factored to $\phi(x) = x^n + 1 = \prod_{i=0}^{n/2-1} (x^2 - \omega_n^{2i+1}) = \prod_{i=0}^{n/2-1} (x^2 - \omega_n^{2br(i)+1})$ with br denoting the bit-reversal function.

Generalization of NTT/INVNTT algorithms. Algorithms 19 and 20 illustrate the operations for the proposed flexible NTT. The original $\text{NTT}^{CT}_{no \to br}$ and $\text{INVNTT}^{GS}_{br \to no}$ algorithms are first modified to support incomplete NTTs, as required for Kyber. The incomplete NTT can be activated using the `early_abort` signal. The proposed algorithms further support either positive or negative wrapped convolutions. The desired convolution mode can be switched with the `negacyclic` signal. Thus, all schemes of Table 5.2 can use the same algorithms. Note that the INVNTT requires a final scaling by n^{-1}.

This chapter mainly focuses on increasing the performance of masked Kyber and Saber implementations. As these two schemes have relatively small polynomial lengths, no on-the-fly Twiddle factor computation is used, and a moderate amount of precomputations is accepted. The combination of $\text{NTT}^{CT}_{no \to br}$ and $\text{INVNTT}^{GS}_{br \to no}$ complicates on-the-fly twiddle factor computations but allows circumventing the bit-reversal. Moreover, the

Algorithm 19: $\text{NTT}^{CT}_{no\rightarrow br}$ transform

Input: $a \in \mathbb{Z}_q/\phi(x)$, *twiddle_table*, `early_abort`, `negacyclic`
Result: $\hat{a} \in \mathbb{Z}_q/\phi(x)$

```
1  if early_abort then
2  |  stop ← n/2
3  else
4  |  stop ← n
5  end
6  t ← n
7  for m = 1 to stop − 1 by m = 2m do
8  |  t ← t/2
9  |  for i = 0 to m − 1 by 1 do
10 |  |  j1 ← 2it,  j2 ← j1 + t
11 |  |  if negacyclic then
12 |  |  |  ω ← twiddle_table[m + i]
13 |  |  else
14 |  |  |  ω ← twiddle_table[i]
15 |  |  end
16 |  |  for j = j1 to j2 − 1 by 1 do
17 |  |  |  z1 ← a_{j+t} · ω  mod q
18 |  |  |  a_j ← a_j + z1  mod q
19 |  |  |  a_{j+t} ← a_j − z1  mod q
20 |  |  end
21 |  end
22 end
```

Algorithm 20: $\text{INVNTT}^{GS}_{br\rightarrow no}$ transform

Input: $\hat{a} \in \mathbb{Z}_q/\phi(x)$, *invtwiddle_table*, `early_abort`, `negacyclic`
Result: $a \in \mathbb{Z}_q/\phi(x)$

```
1  if early_abort then
2  |  m ← n/2,  t ← 2,  k ← 0
3  else
4  |  m ← n,  t ← 1
5  end
6  for m to m > 1 by m = m/2 do
7  |  j1 ← 0
8  |  if negacyclic then
9  |  |  m' ← m/2
10 |  else
11 |  |  m' = 0
12 |  end
13 |  for i = 0 to m/2 − 1 by 1 do
14 |  |  j2 ← j1 + t
15 |  |  if early_abort then
16 |  |  |  ω ← invtwiddle_table[k++]
17 |  |  else
18 |  |  |  ω ← invtwiddle_table[m' + i]
19 |  |  end
20 |  |  for j = j1 to j2 − 1 by 1 do
21 |  |  |  z1 ← a_j,  z2 ← a_{j+t}
22 |  |  |  a_j ← z1 + z2  mod q
23 |  |  |  a_{j+t} ← (z1 − z2) · ω  mod q
24 |  |  end
25 |  |  j1 ← j1 + 2t
26 |  end
27 |  t ← 2t
28 end
```

post-processing can be integrated into the precomputed twiddle factor tables. This is only possible when a DIF butterfly operation is performed within the inverse transform. Another difference to the optimized $\text{NTT}^{CT}_{br\rightarrow no}$ presented in Chapters 3 and 4 is that no memory access optimizations are applied. The reason is that the operand widths are large and, therefore, only one coefficient is stored in one word. However, a dual-port RAM is used to alleviate the memory access bottleneck.

For NTT-based schemes, the Twiddle table is stored in the Montgomery domain to allow the application of the flexible Montgomery multiplier. For negacyclic NTT-based schemes, the Twiddle table contains n ($n/2$ at early aborts) merged values for the powers of ω_n and γ_n in bit-reversed order and the same number of precomputed values for the inverse transform. For schemes with positive wrapped convolutions or schemes not based on NTT, n precomputed values of the powers of ω_n are stored in the Twiddle table.

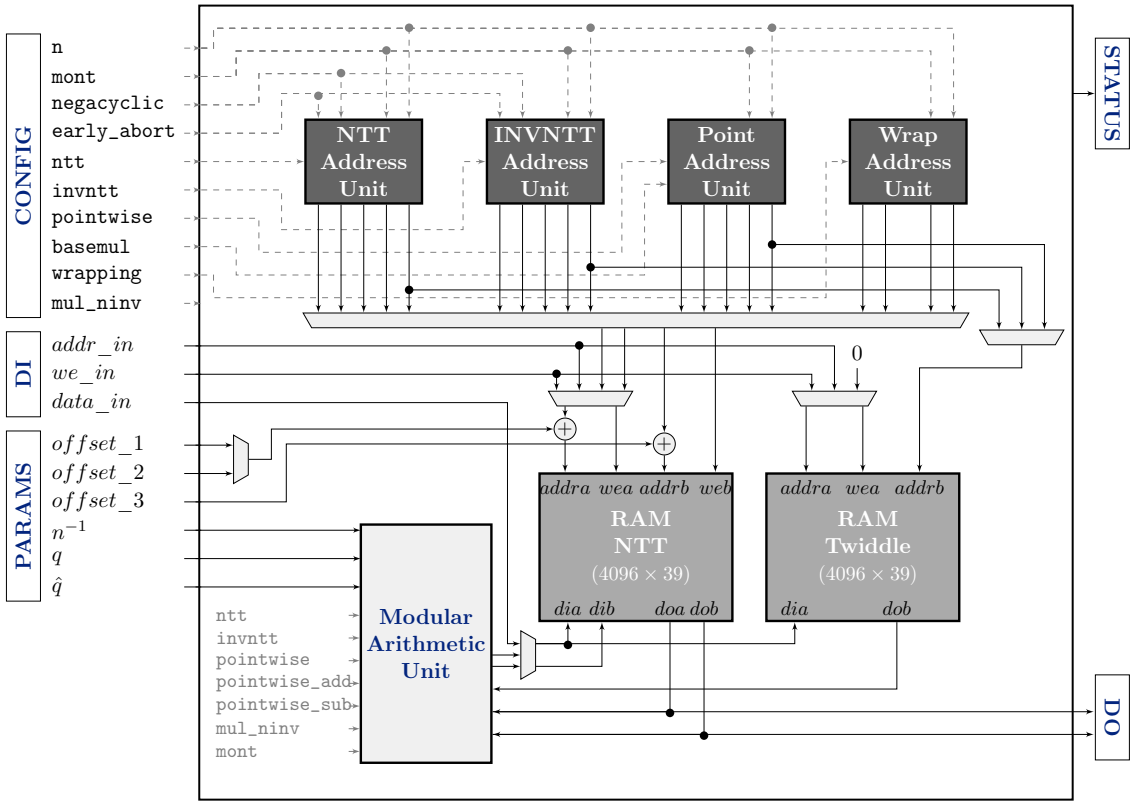


Figure 5.1: Loosely coupled generic NTT (dashed lines for configuration signals).

5.4.2 Flexible NTT Accelerator

Designing an efficient and flexible NTT that supports all mentioned lattice-based schemes requires a new design approach. Figure 5.1 illustrates the developed hardware architecture. The 39-bit NTT operations are not very suitable for instruction set extensions because two registers would be required in a 32-bit architecture for a single operand. This doubles load/store latencies and complicates instruction encodings. Therefore, a loosely coupled approach is the preferred solution to clearly separate the 39-bit operations of the NTT and the 32-bit operations of the processor. The proposed loosely coupled NTT accelerator consists of seven different main modules: two RAM blocks (NTT and Twiddle RAM), four address units (NTT, INVNTT, Point, and Wrap), and a Modular Arithmetic Unit.

NTT and Twiddle RAM. The two memory blocks are used for storing the input/output coefficients and the precomputed Twiddle table, respectively. The size of these memory blocks is chosen large enough to support all parameter sets. The dual-port capabilities of the RAM blocks are exploited to increase the efficiency. In order to keep the bus communication overhead small, the input *data_in* (and the output) can store two coefficients in one word.

Algorithm 21: Basecase multiplication (incomplete NTT)

Input: $\hat{f}, \hat{g} \in \mathbb{Z}_q/\phi(x)$, *twiddle_table*, *invtwiddle_table*
Result: $\hat{h} = \hat{f} \circ \hat{g} \in \mathbb{Z}_q/\phi(x)$

1 **for** $i = 0$ **to** $n/4$ **by** 4 **do**
2 $\quad \omega \leftarrow twiddle\_table[n/4 + i]$
3 $\quad \hat{h}_{4i} \leftarrow \hat{f}_{4i+1} \cdot \hat{g}_{4i+1} \cdot \omega + \hat{f}_{4i} \cdot \hat{g}_{4i} \mod q$
4 $\quad \hat{h}_{4i+1} \leftarrow \hat{f}_{4i} \cdot \hat{g}_{4i+1} + \hat{f}_{4i+1} \cdot \hat{g}_{4i} \mod q$
5 $\quad \omega \leftarrow invtwiddle\_table[n/4 - i - 1]$
6 $\quad \hat{h}_{4i+2} \leftarrow \hat{f}_{4i+3} \cdot \hat{g}_{4i+3} \cdot \omega + \hat{f}_{4i+2} \cdot \hat{g}_{4i+2} \mod q$
7 $\quad \hat{h}_{4i+3} \leftarrow \hat{f}_{4i+2} \cdot \hat{g}_{4i+3} + \hat{f}_{4i+3} \cdot \hat{g}_{4i+2} \mod q$
8 **end**

NTT/INVNTT Address Unit. It generates the two read and write addresses to load and store two coefficients according to Algorithms 19 and 20. It further outputs the read address for the Twiddle factor. The signals `ntt` and `invntt` trigger the corresponding address computations. Optionally, `early_abort` and `negacyclic` can be set. The signal `mont` is used to select the number of pipeline stages to delay the write signals according to the delay in the arithmetic units.

Point Address Unit. It computes the addresses for coefficient-wise multiplications, additions, and subtractions. The signal `basemul` is used to select the basecase multiplication for schemes with early abort. Let $f, g \in \mathbb{Z}_q/\phi(x)$ and let $\mathrm{NTT}(f) \circ \mathrm{NTT}(g) = \hat{f} \circ \hat{g} = \hat{h}$ be the basecase multiplication. This basecase multiplication has for Kyber $n/2$ products [ABD+20], which are computed by

$$\begin{aligned}
\hat{h}_{2i} + \hat{h}_{2i+1}x &= (\hat{f}_{2i} + \hat{f}_{2i+1}x)(\hat{g}_{2i} + \hat{g}_{2i+1}x) \mod x^2 - \omega_n^{2br(i)+1} && (5.2)\\
&= (\hat{f}_{2i} \cdot \hat{g}_{2i}) + (\hat{f}_{2i} \cdot \hat{g}_{2i+1})x + (\hat{f}_{2i+1} \cdot \hat{g}_{2i})x && (5.3)\\
&+ (\hat{f}_{2i+1} \cdot \hat{g}_{2i+1})x^2 \mod x^2 - \omega_n^{2br(i)+1} && (5.4)\\
&= (\hat{f}_{2i} \cdot \hat{g}_{2i} + (\hat{f}_{2i+1} \cdot \hat{g}_{2i+1}) \cdot \omega_n^{2br(i)+1}) + (\hat{f}_{2i} \cdot \hat{g}_{2i+1} + \hat{f}_{2i+1} \cdot \hat{g}_{2i})x && (5.5)
\end{aligned}$$

To ideally exploit the NTT hardware architecture and to reuse the precomputed Twiddle factor tables (which are stored in bit-reversed order), the basecase computation is split into four parts according to Algorithm 21. Each multiplication and addition step can be carried out in $n/4$ cycles (plus pipeline slack), whereas the address is always incremented by four. Note that the Kyber reference implementation in [ABD+20] computes $-twiddle\_table[n/4+i]$ in Line 5. As negative values are disadvantageous for hardware implementations, the symmetry property of the Twiddle factor is used ($-\omega_n^{n-i} \equiv \omega_n^{-i}$) in this work. Thus, the *invtwiddle_table* can be used to circumvent negative numbers.

Wrap Address Unit. This address unit is used for schemes not based on NTT to reduce the length-n' polynomial product by $\phi(x) = x^n + 1$. At this negative wrapping, the lower part of the polynomial is subtracted with the higher part.

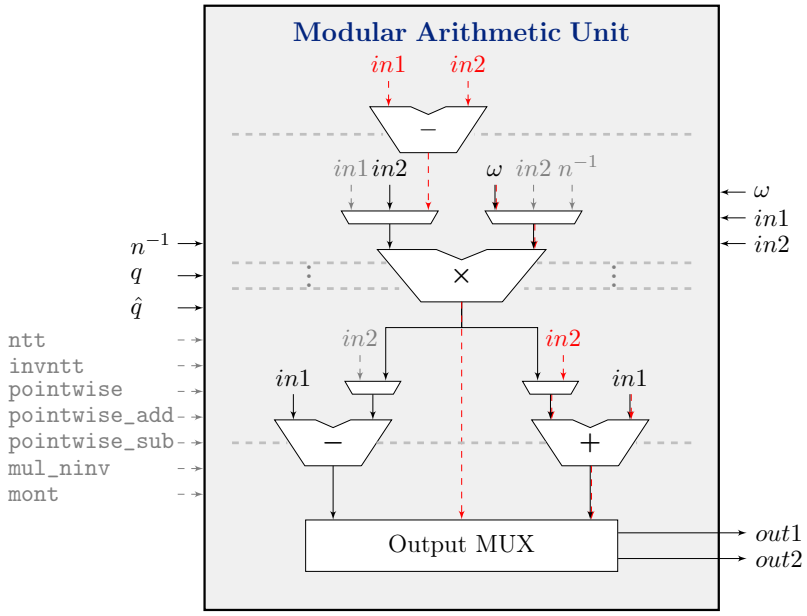


Figure 5.2: Modular Arithmetic Unit. Black: DIT butterfly, red: DIF butterfly, gray: pipeline stages and other functionalities.

Modular Arithmetic Unit. It performs the butterfly operation of Algorithm 19 (Lines 17–19) and Algorithm 20 (Lines 21–23). Figure 5.2 illustrates the architecture of this unit. Its main components are a generic modular multiplier, a modular adder, and two modular subtractors. The signals `ntt`, `invntt`, `pointwise`, `pointwise_add`, `pointwise_sub`, `mul_ninv` are used to configure the multiplexers to either perform DIT or DIF butterfly operations, pointwise multiplications ($out1 = in1 \cdot in2 \mod q$), pointwise additions ($out1 = in1 + in2 \mod q$), pointwise subtractions ($out1 = in1 - in2 \mod q$), or multiplications by constants ($out1 = in1 \cdot n^{-1} \mod q$).

Generic Modular Multiplier. The proposed generic modular multiplier architecture supports Montgomery modular multiplications with up to 24-bit moduli. Moreover, it supports modular multiplications with the reduction-friendly Solinas prime $(2^{39} - 2^{12} + 1)$ for multiplications with lifted primes. The `mont` signal is used to switch between the two operation modes. Costly resources like FPGA DSP blocks are shared between the two multiplication modes. The adders and subtractors are implemented using fast carry chains [KG16]. The reduction logic for multiplications with Solinas prime reduction is implemented using the target prime structure and involves only two additions and three subtractions. To achieve a high operating frequency, the Montgomery and Solinas multiplier have pipeline registers included (12 and 6 stages, respectively).

5.4.3 Results of the Flexible NTT Accelerator

The platform used for the experimental results of this chapter is the NewAE Technology Target Board CW305 equipped with an Artix 7 FPGA XC7A100T. It is designed for measuring side-channel leakages, which is important for the subsequent sections. Table 5.3 compares flexible NTT design solutions. None of the previous works provides a similar degree of versatility. The proposed design supports the following features: (i) configurable on runtime; (ii) the highest parameter range covering all mentioned lattice-based algorithms with n up to 4096 and q up to 39-bit; (iii) positive and negative convolutions; (iv) early abort; and (v) pointwise multiplications, additions, and subtractions.

The required clock cycle count of the loosely coupled NTT architecture is $2n \cdot log_2(n)$ plus 14 or 8 cycles latency depending on whether Montgomery or Solinas prime reductions are performed. The tightly coupled NTT accelerator of the previous chapter achieves a lower cycle count due to its more powerful modular arithmetic unit. The BRAM consumption is another drawback of the loosely coupled approach. Therefore, the tightly coupled method is still more suitable for schemes with small coefficients.

The cycle count can be further reduced when using multiple data RAM blocks (e.g., 8 in [BUC19]). This mitigates the memory access bottleneck because multiple coefficients can be processed in parallel. As shown in [MKÖ+20], this can significantly reduce the cycle count. However, using multiple RAM blocks and butterfly units increases the design complexity and is expensive in terms of area.

Using the generic NTT for polynomial multiplications is not a natural choice for Saber due to its power-of-two modulus. The works in [BMTK+20, SRB20, BR21, ZZY+20] presented high-speed multipliers based on shift-register structures to compute the convolution-like multiplication. The approaches follow similar principles as the ternary multiplier presented in the previous chapters. These types of multipliers can get extremely large for Saber as it does not have ternary polynomial multiplications. Moreover, they are not suitable for NTT-based schemes as Kyber. The NTT design of this chapter provides an appropriate balance between resource costs and performance, and at the same time supports a wide range of parameters and schemes.

5.5 Accelerators for Non-Linear Operations

In this section, hardware architectures for the non-linear operations of Kyber and Saber are described. These operations need to combine information from both shares and require special treatment in a masked design. In contrast to the NTT accelerator, the accelerators proposed in this section are designed for a tight processor coupling.

5.5.1 Masking Keccak

As already pointed out, Keccak is highly relevant for the sampling of random polynomials. While Theta, Rho, Pi, and Iota of the Keccak `f-1600` operation are linear functions consisting of XOR and rotation operations, the function Chi is a non-linear operation, which additionally requires AND as well as NOT operations. The linear functions can

Table 5.3: Resource and performance overview for loosely coupled NTT.

Design	Device	LUT	FF	DSP	BRAM	NTT Cycles (+ pipeline slack)
[BUC19]	ASIC	–	–	–	–	$n = 256$: 1 289 $n = 512$: 2 826 $n = 1024$: 6 155
[MKÖ+20]	FPGA	7 400 8 100 16 000 22 000	5 000 5 200 14 000 17 000	24 24 56 248	24 24 24 96	$n = 256$: 160 $n = 512$: 345 $n = 1024$: 490 $n = 4096$: 3 276
Tightly NTT	FPGA	2 908	170	9	0	$n = 256$: 1 935 $n = 512$: 8 169 [a)] $n = 1024$: 18 537 [a)]
Loosely NTT (this chapter)	FPGA	2 454	1 917	7	4.5	$n = 256$: 4 096(+14/8) $n = 512$: 9 216(+14/8) $n = 1024$: 20 480(+14/8) $n = 2048$: 45 056(+14/8) $n = 4096$: 98 304(+14/8)

[a)] Does not include time for bit-reversal.

be performed on the shares individually. Therefore, only the non-linear function Chi is discussed in more detail.

Masked Chi operation (χ). Let $A[x, y]$ be the input lane (a 64-bit word) of a specific operation and $B[x, y]$ the corresponding output lane. The χ operation is defined by $B[x, y] = A[x, y] \oplus (A[x + 1, y] + 1) \wedge A[x + 2, y]$ for x and y in $[0, 4]$ (see Chapter 2.4). As proposed in [BDPVA10], the output shares $B^0[x, y]$ and $B^1[x, y]$ of the masked input shares $A^0[x, y]$ and $A^1[x, y]$ can be computed with

$$B^0[x,y] \leftarrow A^0[x,y] \oplus (A^0[x+1,y]+1) \wedge A^0[x+2,y] \oplus A^0[x+1,y] \wedge A^1[x+2,y] \quad \text{and} \quad (5.6)$$

$$B^1[x,y] \leftarrow A^1[x,y] \oplus (A^1[x+1,y]+1) \wedge A^1[x+2,y] \oplus A^1[x+1,y] \wedge A^0[x+2,y] \quad . \quad (5.7)$$

If the operations in Equations 5.6 and 5.7 are executed from left to right, the authors in [BDPVA10] argue that all intermediate computations are independent of native variables. Instead of using fresh randomness, different parts of the state are reused to form independent computations.

Masked Chi accelerator. The hardware design of Figure 5.3 is developed to accelerate the computations of Equations 5.6 and 5.7. In order to break down the Chi operation into smaller sub-operations with 32-bit operands, each of the two shared states is split into 2×5 chunks. More precisely, share A^0 is split into the sub-state $\underline{A}^0 \in \{A^0[0{:}4, y, 0{:}31], A^0[0{:}4, y, 32{:}63]\}$ for y in $[0, 4]$ and A^1 is split into $\underline{A}^1 \in \{A^1[0{:}4, y, 0{:}31], A^1[0{:}4, y, 32{:}63]\}$ for y in $[0, 4]$. The following description of the accelerator is divided

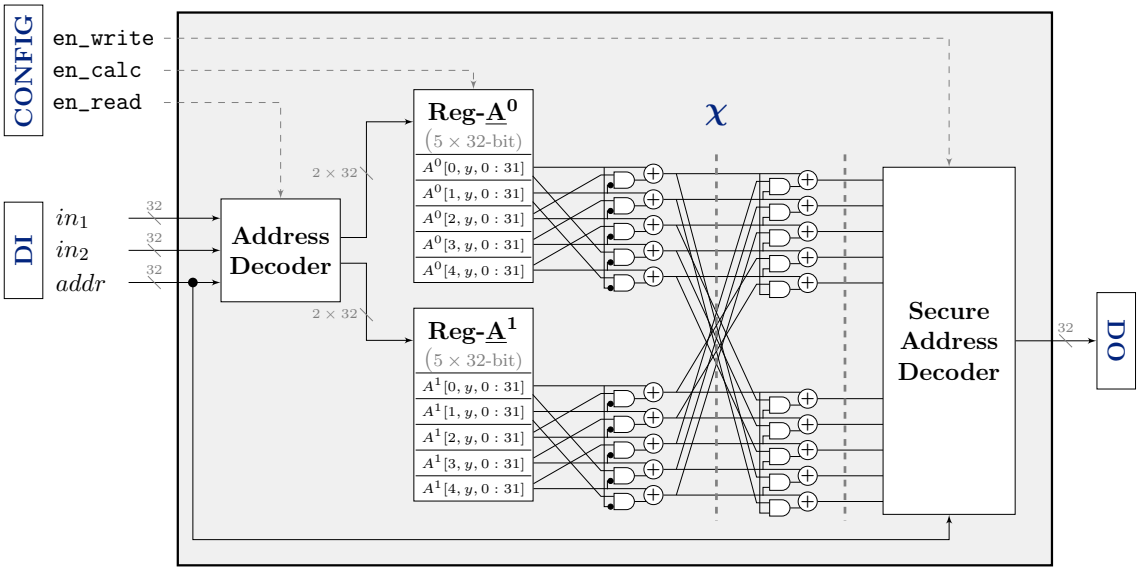


Figure 5.3: Masked Chi accelerator. Dashed lines illustrate register stages and control signals.

into three main steps. In the first step, the first chunks $A^0[0{:}4, 0, 0{:}31]$ and $A^1[0{:}4, 0, 0{:}31]$ for $y = 0$ are written via an address decoder into two separated register files. Depending on the address value, the input in_1 and in_2 are either stored in the registers Reg-$\underline{\text{A}}^0$ or Reg-$\underline{\text{A}}^1$. In the second step, the Chi operation is computed. Therefore, Equation 5.6 is split into two parts: $\hat{B}^0[x, y] = A^0[x, y] + (A^0[x + 1, y] + 1) \wedge A^0[x + 2, y]$ and $B^0[x, y] = \hat{B}^0[x, y] + A^0[x + 1, y] \wedge A^1[x + 2, y]$. These computations can be further split into 2×5 parts such that the Chi operation is performed on each chunk separately. Equation 5.7 is split and processed in the same way. While the first part of these equations ($\hat{B}^0[x, y]$ and $\hat{B}^1[x, y]$) contains only computations with a single share, the second part includes both shares. However, the critical shares are already blinded by independent state bits. In order to avoid leakages due to glitch effects, the computations are separated by registers. In the third and final step, the result of the first chunk is ready and is transferred via a secure address decoder back to the GPR. This procedure is repeated for all 2×5 chunks until the whole Chi operation is performed. Loading the complete state into an accelerator would only lead to a small performance improvement as the actual Chi computation of the proposed accelerator requires only two cycles. However, it would significantly increase the area costs.

Secure address decoder. It prevents leakages during the transfer of the output shares to the GPR. A standard address decoder is equivalent to a MUX that forwards the desired 32-bit value of the Chi accelerator output depending on the *addr* signal. When the bits of the *addr* signal toggle to their new value, glitches can lead to temporarily wrong addresses. In this case, the two shares are possibly forwarded simultaneously, and secret-dependent leakages might occur within the combinatorial path. In order to prevent such leakage effects, the secure address decoder uses a one-hot-encoding. The

one-hot-encoding allows controlling and blocking undesired signal propagation from the wrong share domain.

5.5.2 Masking Binomial Sampling

A few works have already presented masked software implementations for the binomial sampling. The authors in [OSPG18] proposed a masked implementation of the binomial sampling that relies on fresh randomness and a carefully elaborated execution order. Their method is only suitable for a first-order masking. In [SPOG19], the authors enhanced the method of [OSPG18] and further developed another approach that is extensible to a higher-order masking. Their presented approach computes the Hamming weight (number of bits set to one) of x and x' in Equation 4.4 (Section 4.3.6) using some linear arithmetic and a secure AND operation. The method of this work is based on similar principles, but instead of a software solution a hardware circuit is developed. The proposed generic binomial sampling accelerator is suitable for various values of η. It supports Kyber-512 ($\eta = 2$, $\eta = 3$), Kyber-768 ($\eta = 2$), Kyber-1024 ($\eta = 2$), Lightsaber ($\eta = 5$), Saber ($\eta = 4$), and Firesaber ($\eta = 3$). The binomial sampling comprises two different accelerators: a bit-slicing accelerator and a masked adder tree.

Bit-slicing accelerator. Boolean operations on single bits—as required for the masked sampling—are inefficient. The bit-slicing method allows the computation of multiple samples in parallel. It uses the full word-size of the processor to compute, in this case, 32 samples in parallel. But the conversion from the Keccak output into bit-sliced format turns out to be costly in software if the sampling is performed according to the specification of, e.g., Kyber or Saber [BDK+21]. However, turning the Keccak output into a bit-sliced format corresponds to a simple rewiring in hardware. Figure 5.4 shows the top-level architecture of the bit-slicing accelerator. The uniformly distributed Keccak squeeze is stored in up to $2\eta_{max}$ registers within the accelerator with $\eta_{max} = 5$. The transformation in the accelerator is performed according to Algorithm 22 for different values of η. Depending on the value of *addr*, the address decoder at the output selects the desired 32-bit values of $x_i[0:31]$ or $x'_i[0:31]$ with $i \in [0, \eta - 1]$. The bit-slicing accelerator also supports the reverse operation for unpacking bit-sliced values of $x_i[0:31]$ into normal representation. Note that in the following description not only one but 32 samples are processed. This is emphasized by the brackets.

Masked adder tree. After transforming the Keccak squeeze into a bit-sliced format, the two sums $\boldsymbol{s}[0:31] = \sum_0^{\eta-1} x_i[0:31]$ and $\boldsymbol{s'}[0:31] = \sum_0^{\eta-1} x'_i[0:31]$ must be computed and subtracted according to Equation 4.4 (Section 4.3.6). This time i denotes the index of 32-bit variables and not the bit location. These sums compute 32 binomial samples in parallel. The required computations can be performed in hardware using the adder tree shown in Figure 5.5 (a). In order to simplify the subsequent description of the binomial sampler, the brackets are omitted, e.g., instead of $x_i[0:31]$ the notation x_i is used. The adder tree consists of η stages with η half adders each. The computation of the binomial sampling can be split into two steps. In the first step, the sum $\boldsymbol{s}$ is computed using the

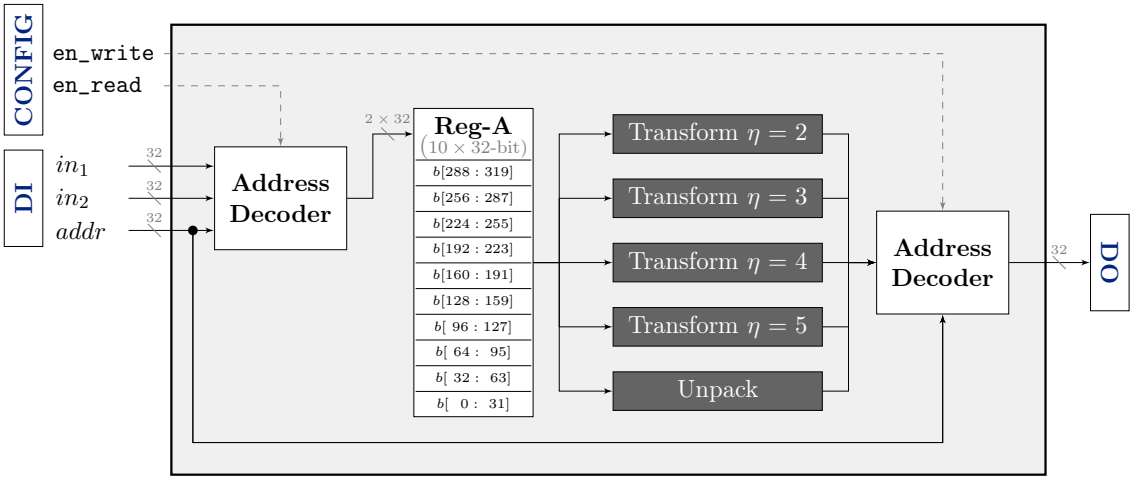


Figure 5.4: Bit-slicing accelerator. Dashed lines illustrate control signals.

Algorithm 22: Bit-slicing transform η

Input: Byte array $\boldsymbol{b} = \{b_0, b_1, \ldots, b_{8\eta-1}\} \in \mathbb{B}^{8\eta}$ with $\mathbb{B}$ denoting the set $[0, 255]$
Result: Bit-sliced 32-bit terms $x_i[0:31]$ and $x'_i[0:31]$ with $i \in [0, \eta - 1]$

$s \leftarrow$ `BytesToBitstream`$(\boldsymbol{b})$
for $i = 0$ **to** 31 **by** 1 **do**
  for $j = 0$ **to** $\eta - 1$ **by** 1 **do**
    $x_j[i] \leftarrow (s \gg (2\eta \cdot i + j)) \wedge 1$
    $x'_j[i] \leftarrow (s \gg (2\eta \cdot i + j + \eta)) \wedge 1$
  end
end

input $\boldsymbol{x}$ and $\boldsymbol{z}$, whereas $\boldsymbol{z}$ is initially set to zero. Now, in each stage, the 32-bit values of $\boldsymbol{x}$ are subsequently added to the intermediate sum of the previous stage. The carries of these computations are always forwarded to the next half adder of the same stage, and the intermediate sums are forwarded to the next stage. After η stages, the circuit outputs the sum $\boldsymbol{s}$. In the second step, the sum of the previous step is assigned to the input $\boldsymbol{z} = \boldsymbol{s}$, and additions with the inverse of $\boldsymbol{x'}$ are performed within the stages. This corresponds to the desired subtraction of $\boldsymbol{s}$ and $\boldsymbol{s'}$.

Masked adder tree based on TI. Let the sum and carry computations in the adder tree be split into two functions $f_1 : (x_0, z_0) \rightarrow (s_0)$ with $s_0 = x_0 \oplus z_0$ and $f_2 : (c_{i-1}, z_{i-1}, z_i) \rightarrow (c_i, s_i)$ with $c_i = c_{i-1} \wedge z_{i-1}$ and $s_i = z_i \oplus c_i$ for $i \neq 0$. The direct sharing approach presented in [BNN+12] can be used to construct functions that are in accordance to the TI principles. With the three input shares of $\boldsymbol{x}_i^{\{0:2\}}$, the sum $\boldsymbol{s}_i^{\{0:2\}}$ and carry $\boldsymbol{c}_i^{\{0:2\}}$ can be computed with the following splits:

$$s_0^0 = x_0^0 \oplus z_0^0, \quad s_0^1 = x_0^1 \oplus z_0^1, \quad s_0^2 = x_0^2 \oplus z_0^2 \tag{5.8}$$

$$c_i^0 = (c_{i-1}^1 \wedge z_{i-1}^1) \oplus (c_{i-1}^1 \wedge z_{i-1}^2) \oplus (c_{i-1}^2 \wedge z_{i-1}^1); \quad s_i^0 = z_i^0 \oplus c_i^0 \tag{5.9}$$

$$c_i^1 = (c_{i-1}^2 \wedge z_{i-1}^2) \oplus (c_{i-1}^0 \wedge z_{i-1}^2) \oplus (c_{i-1}^2 \wedge z_{i-1}^0); \quad s_i^1 = z_i^1 \oplus c_i^1 \tag{5.10}$$

$$c_i^2 = (c_{i-1}^0 \wedge z_{i-1}^0) \oplus (c_{i-1}^0 \wedge z_{i-1}^1) \oplus (c_{i-1}^1 \wedge z_{i-1}^0); \quad s_i^2 = z_i^2 \oplus c_i^2 \tag{5.11}$$

While the linear functions in these equations can always be computed with a single share, for the non-linear functions, at least one share is always missing during the computations (non-completeness property). When converting the proposed adder tree using TI principles and the functions f_1 and f_2, the architecture of Figure 5.5 (b) is obtained. It is impossible to fulfill the uniformity property of a non-linear Boolean operation that has two inputs and one output [NRS11]. Therefore, the uniformity property for each output of the function f_2 needs to be recovered using fresh randomness. Changing the adder tree to use full adders where three-input operations are used to avoid the refreshing step is theoretically possible. However, such an architecture would lose the flexibility as for each η another circuit would be required. Therefore, another alternative to reduce the randomness requirements is investigated.

Masked adder tree based on DOM. When the uniformity property is preserved, secure TI implementations can be realized with a low amount of randomness. As this is not the case for the proposed adder tree and the generation of fresh randomness is in most platforms expensive, the behavior of the adder tree architecture with DOM principles is investigated. The DOM approach significantly reduces the complexity, as shown in Figure 5.5 (c). Instead of three instances of f_1 and $3 \cdot (\eta_{max} - 1)$ instances of f_2 in each level, only two of f_1 and $\eta_{max} - 1$ of f_2-DOM are required. The computation $c_i = c_{i-1} \wedge z_{i-1}$ in f_2-DOM is realized with the secure DOM-AND. For the generation of 32 binomially distributed coefficients, the adder tree based on TI requires $4 \cdot \eta_{max} \cdot (\eta_{max} - 1)$

random 32-bit values plus $(2 \cdot \eta_{max})$ values for randomizing the zero-input of $\boldsymbol{z}$. In contrast, the DOM approach requires $2 \cdot \eta_{max} \cdot (\eta_{max} - 1)$ plus η_{max} random values. For instance, with $\eta_{max} = 5$ the amount of randomness reduces from 90×32-bit to 45×32-bit.

5.5.3 Secure Adder

The secure arithmetic addition for masked Boolean shares is an essential element for the generic B2A and A2B conversions. Two secure adder designs based on the ripple-carry adder and Kogge–Stone adder were proposed in [SMG15]. The Kogge–Stone adder achieves a lower latency as it belongs to the class of carry-lookahead adders. It splits the carry computation into a generate and propagate part. Due to its good performance, the suggested Kogge–Stone adder was adopted for the hardware design of this subsection. The TI-based Kogge–Stone adder for three shares, shown in Figure 5.6, is constructed using three stages for performing 4-bit additions. The vertical stages create propagate bits $\boldsymbol{p}_i^{\{0:2\}}$ and generate bits $\boldsymbol{g}_i^{\{0:2\}}$. The first stage, requires the linear function $f_1 : (\boldsymbol{x}_i^{\{0:2\}}, \boldsymbol{y}_i^{\{0:2\}}) \rightarrow \boldsymbol{p}_i^{\{0:2\}}$ with

$$p_i^0 = x_i^0 \oplus y_i^0, \quad p_i^1 = x_i^1 \oplus y_i^1, \quad p_i^2 = x_i^2 \oplus y_i^2 \tag{5.12}$$

and the non-linear function $f_2 : (\boldsymbol{x}_i^{\{0:2\}}, \boldsymbol{y}_i^{\{0:2\}}, r_i) \rightarrow \boldsymbol{g}_i^{\{0:2\}}$ with

$$g_i^0 = (x_i^1 \wedge y_i^1) \oplus (x_i^1 \wedge y_i^2) \oplus (x_i^2 \wedge y_i^1) \oplus r_i \ , \tag{5.13}$$

$$g_i^1 = (x_i^2 \wedge y_i^2) \oplus (x_i^0 \wedge y_i^2) \oplus (x_i^2 \wedge y_i^0) \oplus (x_i^0 \wedge r_i) \oplus (y_i^0 \wedge r_i) \ , \tag{5.14}$$

$$g_i^2 = (x_i^0 \wedge y_i^0) \oplus (x_i^0 \wedge y_i^1) \oplus (x_i^1 \wedge y_i^0) \oplus (x_i^0 \wedge r_i) \oplus (y_i^0 \wedge r_i) \oplus r_i \ . \tag{5.15}$$

The remaining stages require f_2 and $f_3 : (\boldsymbol{g}_{i+j}^{\{0:2\}}, \boldsymbol{g}_i^{\{0:2\}}, \boldsymbol{p}_{i+j}^{\{0:2\}}) \rightarrow \boldsymbol{g}_{i+j}^{\{0:2\}}$ with $j = 2^{stage-1}$ and

$$g_{i+j}^0 = (g_i^1 \wedge p_{i+j}^1) \oplus (g_i^1 \wedge p_{i+j}^2) \oplus (g_i^2 \wedge p_{i+j}^1) \oplus g_{i+j}^1 \ , \tag{5.16}$$

$$g_{i+j}^1 = (g_i^2 \wedge p_{i+j}^2) \oplus (g_i^0 \wedge p_{i+j}^2) \oplus (g_i^2 \wedge p_{i+j}^0) \oplus g_{i+j}^2 \ , \tag{5.17}$$

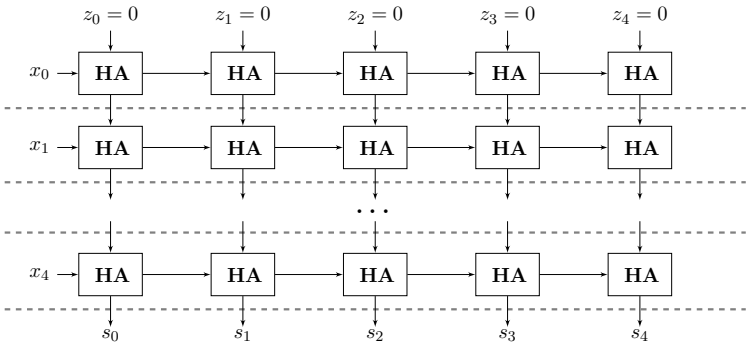
$$g_{i+j}^2 = (g_i^0 \wedge p_{i+j}^0) \oplus (g_i^0 \wedge p_{i+j}^1) \oplus (g_i^1 \wedge p_{i+j}^0) \oplus g_{i+j}^0 \ . \tag{5.18}$$

While the first stage requires further randomness for recovering the uniformity property after f_2, the remaining stages can use the independent bit values of g_i^0 instead of r_i to keep uniformity.

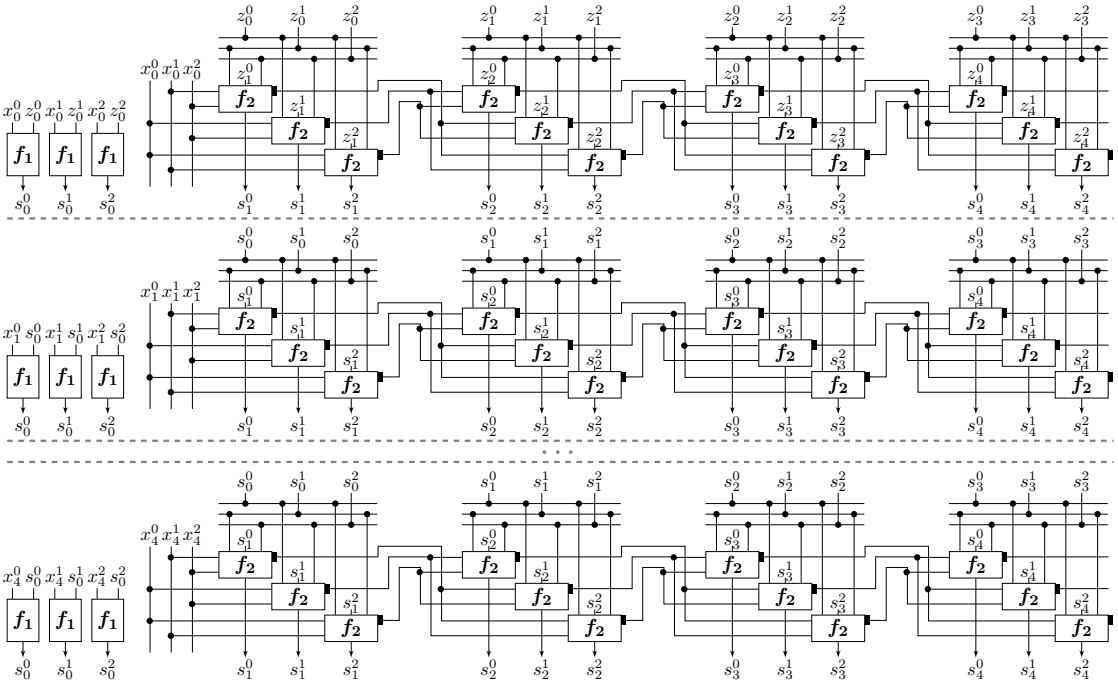
5.5.4 Results of Non-Linear Accelerators

Table 5.4 summarizes the resource utilization and performance of the developed accelerators. Critical signals and components that involve non-linear operations were defined with the Verilog `dont_touch` attribute, preventing the synthesis tool from optimizations. A lower resource utilization can be expected without this attribute, but a higher risk for undesired optimizations exists.

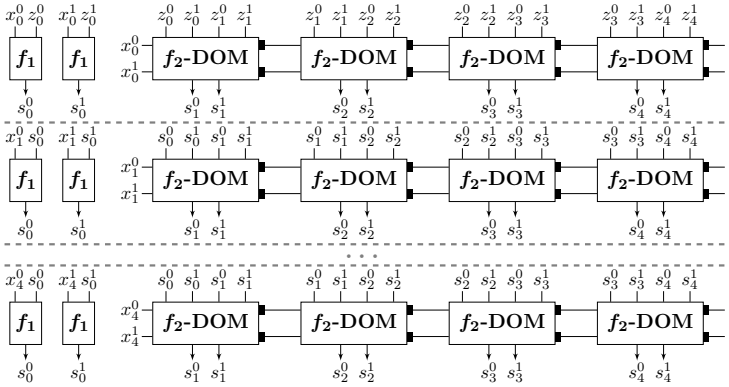
The Keccak `f-1600` accelerator supports complete round computations for non-masked settings and incomplete round computations (only Theta, Rho, Pi) for masked settings.



(a) Adder tree for binomial sampling accelerator



(b) Masked adder tree for binomial sampling accelerator (TI)



(c) Masked adder tree for binomial sampling accelerator (DOM)

Figure 5.5: Adder tree for binomial sampling (Binom Tree) with $\eta_{max} = 5$.

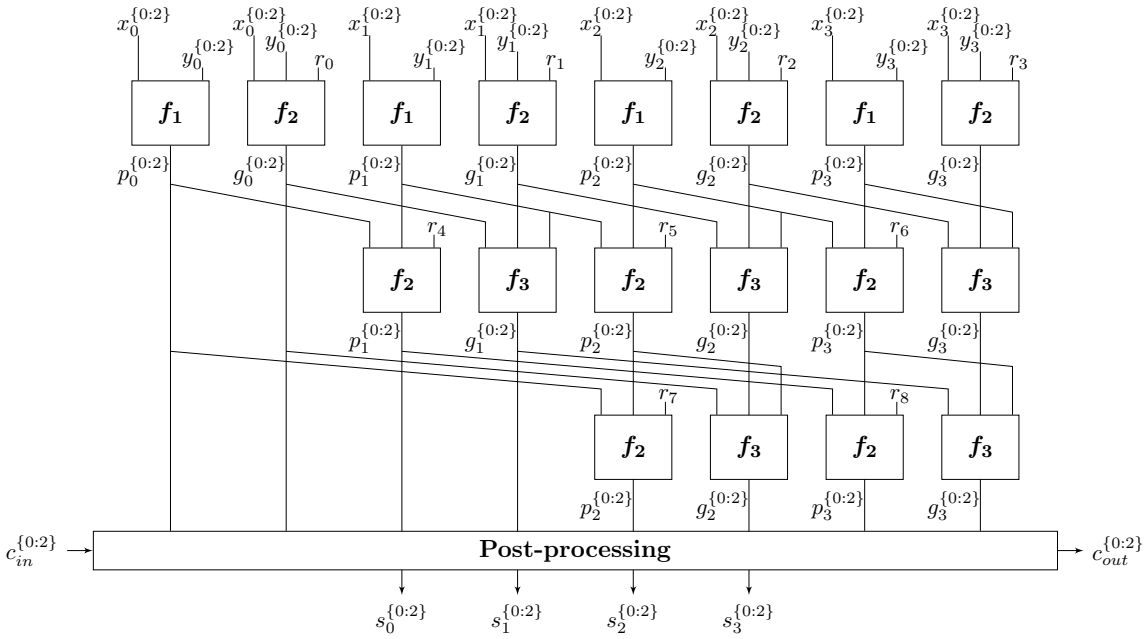


Figure 5.6: Secure Kogge–Stone adder (SecAdd) for 4-bit additions.

It is an enhanced version of the design presented in Section 4.3.5. The masked Chi accelerator is used to securely accelerate the non-linear operation of Keccak. So far, no other masked hardware/software codesign of Keccak was found. The masked pure hardware designs in [BDPVA10, GSM17, ABP+18] report only ASIC results in gate equivalents making a comparison to this thesis difficult.

The results show that the DOM variant of the binomial sampling accelerator (Binom Tree) does not only decrease the required amount of randomness but also leads to a significant area reduction compared to the TI variant. Therefore, only the DOM variant is considered for further measurements in the remainder of this thesis.

Compared to [SMG15], the secure adder of this work is very similar. Both are designed for 32-bit operations. The higher resource consumption can be explained by the `dont_touch` attributes, an additional secure address decoder, and an additional feature that allows computing 32-bit additions with and without input carry.

Cycle count of non-linear accelerators. The cycle counts in Table 5.4 are the latencies within the accelerator without load/store and clean operations. The Keccak `f-1600` operation is single cycled. Although the complete `f-1600` operation is performed without any pipeline stages, it achieves a high frequency. The masked Chi accelerator requires, due to the two pipeline stages within the Chi operation, two clock cycles for the arithmetic computations (see Figure 5.3). The bitslicing accelerator consists of only combinatorial logic without any register stages. The binomial sampling accelerator (Binom Tree) has five stages with five half-adders each for $\eta_{max} = 5$. The largest propagation delay from the input $\{x_0, z_0\}$ to the output s_4 is nine cycles (propagation from upper-left to lower right

Table 5.4: Resource and performance overview for the non-linear accelerators.

Design	LUT	FF	Slices	DSP	BRAM	Max. Freq.	Cycles
Keccak (`f-1600`)	3 100	0	920	0	0	303 MHz	1/round
Masked Chi	1 488	971	632	0	0	370 MHz	2
Bit-slice	277	320	123	0	0	357 MHz	1
Binom Tree (DOM)	5 352	2 570	1 706	0	0	112 MHz	9
Binom Tree (TI)	9 232	4 166	2 839	0	0	140 MHz	9
Secure Adder (TI) [SMG15]	937[a)]	1 330[a)]	–	0	0	62 MHz	6
Secure Adder (TI)	2 464	1 323	1 054	0	0	454 MHz	6 (7)

[a)] Does not contain resources for secure address decoder and no support for an input carry.

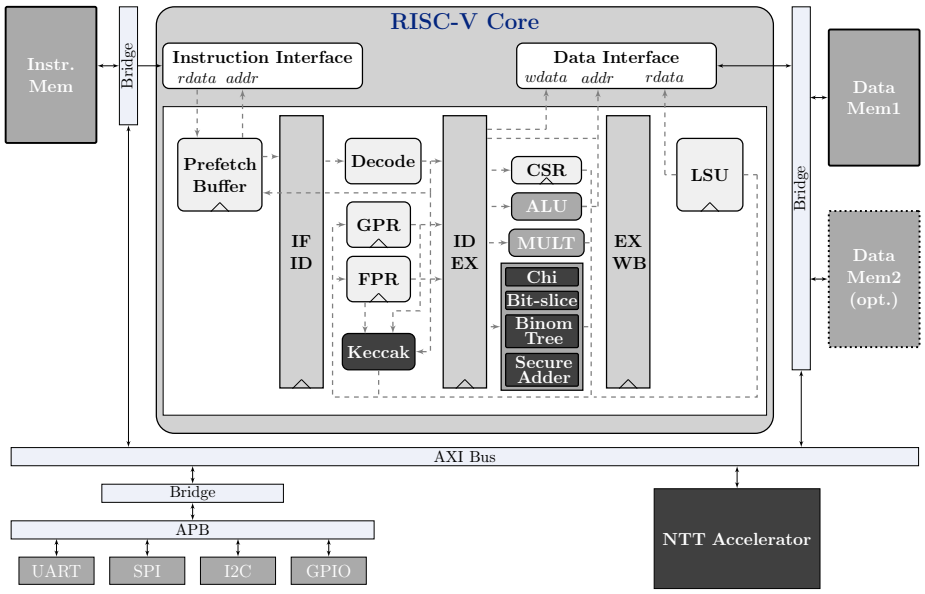


Figure 5.7: RISC-V system with masked post-quantum accelerators.

corner in Figure 5.5). The secure adder has for a 32-bit addition six register stages. The secure addition requires one additional clock cycle when the support for computations with input carry is enabled.

5.6 System Integration

Figure 5.7 shows the architecture of the complete RISC-V system with accelerators. It includes the tightly coupled accelerators (Keccak, Chi, Bit-slice, Binom Tree, Secure Adder) and the loosely coupled NTT accelerator. The same environment is used as in Chapter 4. The optional FPR without FPU is again activated as it is required for the Keccak `f-1600` accelerator.

5.6.1 Accelerator Integration

The configuration registers and memory of the loosely coupled NTT are memory-mapped. The addresses starting at 0x1B10 8000 include: (i) the parameters *offset_1*, *offset_2*, *offset_3*, n^{-1}, q, and $\hat{q}$, (ii) a configuration register containing the polynomial length n and the configuration signals `mont`, `negacyclic`, `early_abort`, `ntt`, `invntt`, `pointwise`, `basemul`, `wrapping`, and `mul_ninv` (see Section 5.4.2).

The tightly coupled Keccak accelerator for the `f-1600` round function is placed, similar as in Chapter 4, in the decode stage. All non-linear accelerators require at most three input and one output operand and are placed in the execution stage. The Keccak accelerator can be configured to perform complete rounds for non-masked settings and incomplete rounds for masked settings (without the non-linear Chi operation as it is performed with a separate accelerator). The register $rs1$ controls this configuration. The register $rs2$ is used for the Keccak round selection. The remaining accelerators have *write* instructions (input in $rs1/rs2$, address in rd) and *read* instructions (output and address in rd) to securely copy the shares between register file and accelerator. In addition to the compute operation, the Binom Tree accelerator has instructions for resetting $\boldsymbol{z}^{\{0:n-1\}}$ and copying the sum $\boldsymbol{s}^{\{0:n-1\}}$ to the input $\boldsymbol{z}^{\{0:n-1\}}$. The instruction *pq.mbincinv* is used for the computation of the subtraction.

5.6.2 Architectural Leakage Reduction

Storing two shares in the same register file can lead to exploitable leakages, even if both shares are not accessed simultaneously [SR15]. The reason is that the registers can be connected to the same internal bus and combinatorial circuit. Although influencing the performance, only one share is located within the register file at each time step in this work. Before processing the second share, the first share is cleared. At the non-linear accelerators in the execution stage, the shares are always stored in different register files. Register values are only accessed via a secure address decoder, as discussed in Section 5.5.1.

The pipeline registers between the decode and execution stage are another typical source of leakage at the transition of operations with another share. This affects three operand registers for the ALU, multiplier unit, and post-quantum accelerators, respectively. These pipeline registers must be cleared after critical operations. Moreover, the serial divider, which can perform divisions and remainder computations, contains pipeline registers that must be cleared to avoid leakages.

The instruction and data memories in an FPGA design are constructed using BRAM resources. The main elements of a BRAM are an input register, memory array, output latch, and an optional output register to improve the critical path. Overwriting one of the registers/latches with another share can lead to exploitable leakages [BDGH15]. The routing nets in the memory array have buffers to improve the signal quality. Charging and discharging the nets can thus lead to amplified leakages. In order to avoid such effects, a second data memory is placed in the design. It can be optionally used to clearly separate the shares for critical operations. Variables can be relocated using the

Table 5.5: FPGA resource overview and estimated max. frequency for the system with and without accelerators.

Design	Platform	LUT	FF	Slices	DSP	BRAM	Max. Freq.
Baseline	PULPino	13 010	8 318	4 821	6	32	62 MHz
Accel.	PULPino	20 697	11 833	6 852	13	36.5	62 MHz
Accel. masked	PULPino	29 889	17 152	9 641	13	52.5	58 MHz

Table 5.6: ASIC resource overview and estimated max. frequency (UMC 65 nm).

	Cell Count	Combinat.	Sequent.	Buffer+Inv.	Clk-Gate	Memory	Max. Freq.
PULPino orig.	42 115	78 373 μm^2 (54 kGE)	92 261 μm^2 (64 kGE)	20 534 μm^2 (14 kGE)	365 μm^2 (0.25 kGE)	669 345 μm^2 (465 kGE)	75 MHz
Accel. (Chap. 4)	68 853	148 941 μm^2 (103 kGE)	102 203 μm^2 (71 kGE)	27 186 μm^2 (19 kGE)	346 μm^2 (0.24 kGE)	669 345 μm^2 (465 kGE)	46 MHz
Accel. (Chap. 5)	74 197	151 050 μm^2 (105 kGE)	131 592 μm^2 (91 kGE)	36 760 μm^2 (26 kGE)	397 μm^2 (0.28 kGE)	992 336 μm^2 (689 kGE)	77 MHz
Accel. masked (Chap. 5)	97 987	195 562 μm^2 (136 kGE)	180 843 μm^2 (126 kGE)	48 481 μm^2 (34 kGE)	392 μm^2 (0.27 kGE)	1 327 008 μm^2 (922 kGE)	75 MHz

`section` attribute of the compiler.

5.6.3 Results of System Integration

Table 5.5 provides the FPGA resource consumption with three different configurations: (i) RISC-V baseline implementation without accelerators and FPR; (ii) accelerated implementation that includes the loosely coupled NTT and tightly coupled Keccak accelerators; (iii) accelerated masked implementation that further includes the Chi, Bit-slice, Binom Tree, and Secure Adder accelerators.

When comparing the accelerated implementation with the original RISC-V platform, the number of LUTs and registers increased by factors of 1.59 and 1.42, respectively. A comparison to the implementation of Chapter 4 is provided for an ASIC synthesis below. When comparing the accelerated configuration with the configuration that further integrates masking accelerators, the number of LUTs increased by a factor of 1.44 and the amount of registers by a factor of 1.45. A direct resource comparison to [AEL+20], which uses a finite field accelerator to accelerate lattice-based cryptography on RISC-V, is barely possible as a completely different platform was used. However, the resource overhead in [AEL+20] is expected to be smaller as only a single Barrett multiplier is added to the original core.

ASIC design results. Table 5.6 summarizes the resource overview of the ASIC UMC 65 nm designs after complete layout generation. The same technology and low leakage library with high threshold voltage were chosen as in the previous chapters. This choice trades performance in favor of a low power and energy consumption and is thus well suited for embedded devices. Compared to the accelerated system design in Chapter 4.3,

the accelerated design of this chapter requires a similar amount of combinatorial cells and about 29 % more sequential cells. Due to the loosely coupled NTT, the memory size is about 48 % larger. While FPGAs offer a high number of BRAMs with dual-port capabilities, memory is usually very costly for ASIC designs. However, one advantage of an ASIC design is the higher flexibility as also the unusual word length of 39-bit can be directly supported. The presented NTT design uses one dual-port RAM (207 178 μm^2) for the coefficients and one single port RAM (115 812 μm^2) for the Twiddle factors (each of size $4k \times 39$-bit). When only Kyber and Saber are targeted, smaller memory sizes would be sufficient. For example, for a $1k \times 39$-bit single-port RAM, the area reduces to 37 526 μm^2.

A visual comparison of the different design approaches is provided in Figure 5.8. It shows the placement of the standard cells and macro blocks. It is clearly recognizable that the additional memory blocks of the generic loosely coupled NTT presented in this chapter require much area. Tailoring the design for specific schemes might reduce this area. However, the tightly coupled approach and on-the-fly twiddle factor computation are more suitable when targeting a low area. For a core size of $1.46\,mm \times 1.46\,mm$, the core utilization is 40.39% (baseline), 44.48% (accelerated Chapter 4.3), 61.56% (accelerated this chapter), and 82.21% (masked accelerated this chapter). The measurements exclude physical cells such as filler and well-tap cells.

Masked designs benefit from a second data memory, as discussed in Section 5.6.2. Moreover, masked designs have a higher memory consumption compared to non-masked designs as they need to store multiple shares and a large number of random values. Computing the random values with an efficient TRNG on the fly could reduce the memory consumption and hence the required area. However, the development, integration, and evaluation of the masked design with such a TRNG has been left as future work.

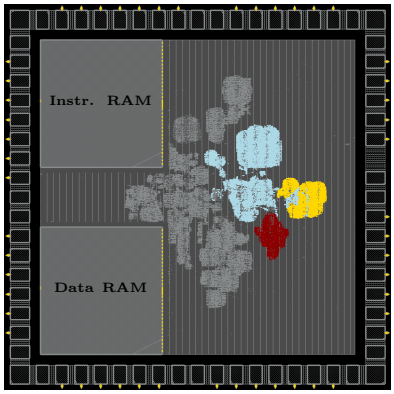
5.7 Experimental Results

This section provides an overview of the performance results for the optimized non-masked and masked implementations of Kyber and Saber, and the leakage assessment of the developed routines and accelerators.

5.7.1 Performance of Unmasked Implementations

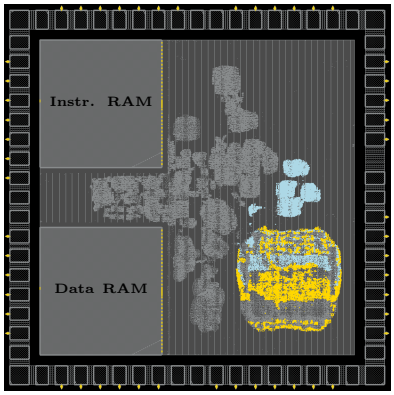
Table 5.7 summarizes the benchmark results of the accelerated non-masked Kyber and Saber implementations. They only use the loosely coupled generic NTT and Keccak `f-1600` accelerators. The results show that the new design beats the cycle count of all ARM Cortex-M4 implementations and RISC-V hardware/software codesigns. Compared to the fastest assembly-optimized ARM Cortex-M4 implementations, cycle count improvement factors of 3.47 for Kyber-768 and 2.63 for Saber were achieved (whole algorithm execution). While the ARM Cortex-M4 is more advanced than the deployed open-source RISC-V core, the area overhead of the accelerators must be considered. Compared to the RISC-V instruction set extensions for finite field operations in [AEL+20], an improvement factor of 7.06 for Kyber-1024 was achieved. Clearly, it must be noted

RISC-V baseline



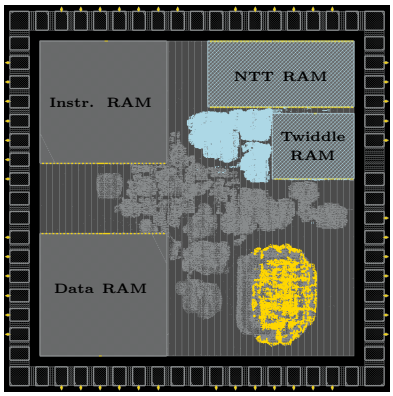
- RISC-V core region
- RISC-V ALU
- RISC-V MULT

Accelerated (Chapter 4.3)



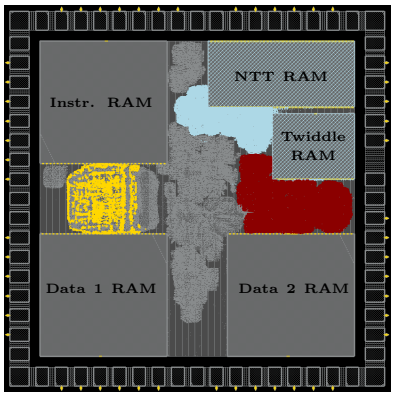
- NTT and modular arithmetic accelerator
- Keccak accelerator

Accelerated (Chapter 5)



- Loosely generic NTT
- Keccak accelerator

Accelerated masked (Chapter 5)



- Loosely generic NTT
- Masking accelerators
- Keccak accelerator

Figure 5.8: Placement of cells for different ASIC design versions.

that more powerful accelerators are larger, which, however, is justified by the achieved performance gain.

Compared to the RISC-V design in Chapter 4, a performance improvement factor of 1.14 for Kyber-768 and 3.30 for Saber was achieved (whole algorithm execution). Due to the genericity of the NTT unit, a clear performance advantage for the non-NTT based scheme Saber gets visible. The slight performance advantage of Kyber with the new NTT design is achieved by a systematic reduction of the data transfer between core and accelerator. The matrix-vector multiplications in MLWE/MLWR schemes require multiplying different ring elements from the matrix with always the same vector. In order to reduce the communication overhead, the transformed vector is left within the NTT memory, and the result is only written back at the very end when also subsequent operations like polynomial additions/subtractions are completed.

Further cycle count improvements can be achieved with coprocessor solutions where the main processor is mostly used for configuration purposes, as in Sapphire [BUC19] and VPQC [XHY+20]. These almost standalone solutions compute large parts of the complete scheme within the accelerator. However, this work focuses on a solution that uses the RISC-V processor as the main computing element to keep the flexibility high. This facilitates spontaneous algorithmic changes and the integration of SCA countermeasures.

5.7.2 Performance of Masked Implementations

This section provides an overview of the results for the masked Kyber and Saber implementations and compares them to prior and concurrent works [OSPG18, BDK+21, BGR+21]. The masked RLWE implementation presented in [OSPG18] is based on the NewHope algorithm, which has many similarities to Kyber. Both are NTT-based and use a prime modulus, leading to similar masking requirements and approaches. The masked RLWE scheme in [OSPG18] can be categorized to *NIST Level V*. Although the comparison between ARM Cortex-M4 and the deployed RISC-V platform is difficult, the measurements in Table 5.8 indicate that the presented accelerators and masking methods lead to a significantly lower cycle count.

Compared to the masked Saber implementation in [BDK+21], a cycle count improvement of factor 3.10 (including randomness generation) is achieved. Larger improvements might be possible using specialized accelerators for Saber. However, this work focuses on a high flexibility. It also must be noted that the chosen algorithms of [BDK+21] are not easily extensible to higher masking orders. The masked Kyber implementation in [BGR+21] was initially developed for the ARM Cortex-M0, an energy-efficient and resource-constrained platform. A direct comparison with the deployed RISC-V core is again difficult. But in absolute cycle counts, the presented implementation is a factor of 8.70 faster (including randomness generation).

For the target platform of this work, Kyber is more costly to mask than Saber. The reason is that Kyber requires sampling masked error polynomials, while Saber avoids this using a deterministic rounding operation. Moreover, the prime modulus of Kyber turns out to be more costly at some points. For example, the secure addition modular q operation for $\mathrm{B2A}_q/\mathrm{A2B}_q$ conversions requires more instructions than the normal secure

Table 5.7: Cycle count and code size in bytes of optimized non-masked Kyber and Saber.

Algorithm	Device	KeyGen	Encaps	Decaps	Code size
Kyber-512 [KRSS19]	ARM M4	514 291 (×4.42)	652 769 (×3.71)	621 245 (×3.33)	11 000
Kyber-512 [ABCG20]	ARM M4	455 191 (×3.91)	586 334 (×3.33)	543 500 (×2.92)	–
Kyber-512 [Gre20]	RISC-V (VexRiscv)	1 218 557 (×10.46)	1 592 689 (×9.05)	1 515 876 (×8.13)	–
Kyber-512 opt. [AEL+20]	RISC-V (VexRiscv)	710 000 (×6.10)	971 000 (×5.52)	870 000 (×4.67)	–
Kyber-512 baseline (Chap. 4)	RISC-V (PULPino)	1 137 052 (×9.76)	1 547 789 (×8.79)	1 525 621 (×8.19)	16 928
Kyber-512 opt. (Chap. 4)	RISC-V (PULPino)	150 106 (×1.29)	193 076 (×1.10)	204 843 (×1.10)	12 532
Kyber-512 opt. (Chap. 5)	**RISC-V (PULPino)**	**116 454 (×1.00)**	**176 034 (×1.00)**	**186 341 (×1.00)**	**14 208**
Kyber-768 [KRSS19]	ARM M4	976 757 (×4.57)	1 146 556 (×3.85)	1 094 849 (×3.50)	11 400
Kyber-768 [ABCG20]	ARM M4	864 008 (×4.04)	1 032 540 (×3.46)	969 867 (×3.10)	–
Kyber-768 [Gre20]	RISC-V (VexRiscv)	2 288 109 (×10.70)	2 771 517 (×9.30)	2 653 584 (×8.48)	–
Kyber-768 baseline (Chap. 4)	RISC-V (PULPino)	2 102 505 (×9.83)	2 625 824 (×8.81)	2 573 963 (×8.22)	17 266
Kyber-768 opt. (Chap. 4)	RISC-V (PULPino)	273 370 (×1.28)	325 888 (×1.09)	340 418 (×1.09)	11 658
Kyber-768 opt. (Chap. 5)	**RISC-V (PULPino)**	**213 862 (×1.00)**	**298 048 (×1.00)**	**313 034 (×1.00)**	**13 028**
Kyber-1024 [KRSS19]	ARM M4	1 575 052 (×5.92)	1 779 848 (×4.83)	1 709 348 (×4.35)	12 424
Kyber-1024 opt. [ABCG20]	ARM M4	1 404 695 (×5.28)	1 605 707 (×4.36)	1 525 805 (×3.88)	–
Kyber-1024 [Gre20]	RISC-V (VexRiscv)	3 686 344 (×13.85)	4 280 420 (×11.62)	4 123 722 (×10.50)	–
Kyber-1024 [AEL+20]	RISC-V (VexRiscv)	2 203 000 (×8.28)	2 619 000 (×7.11)	2 429 000 (×6.18)	–
Kyber-1024 baseline (Chap. 4)	RISC-V (PULPino)	3 378 603 (×12.69)	4 024 887 (×10.93)	3 949 039 (×10.05)	17 670
Kyber-1024 opt. (Chap. 4)	RISC-V (PULPino)	349 673 (×1.31)	405 477 (×1.10)	424 682 (×1.08)	12 874
Kyber-1024 opt. (Chap. 5)	**RISC-V (PULPino)**	**266 209 (×1.00)**	**368 409 (×1.00)**	**392 873 (×1.00)**	**14 442**
Lightsaber [KRSS19]	ARM M4	459 965 (×3.12)	651 273 (×3.23)	678 810 (×3.00)	44 916
Lightsaber [BMKV20]	ARM M4	466 000 (×3.16)	653 000 (×3.24)	678 000 (×2.99)	–
Lightsaber [CHK+21]	ARM M4	360 000 (×2.44)	513 000 (×2.55)	498 000 (×2.20)	–
Lightsaber baseline (Chap. 4)	RISC-V (PULPino)	1 071 836 (×7.27)	1 503 594 (×7.46)	1 537 939 (×6.79)	18 772
Lightsaber opt. (Chap. 4)	RISC-V (PULPino)	366 837 (×2.49)	526 496 (×2.61)	657 583 (×2.90)	12 544
Lightsaber opt. (Chap. 5)	**RISC-V (PULPino)**	**147 472 (×1.00)**	**201 457 (×1.00)**	**226 528 (×1.00)**	**11 442**
Saber [KRSS19]	ARM M4	896 035 (×3.84)	1 161 849 (×3.72)	1 204 633 (×3.43)	44 468
Saber [BMKV20]	ARM M4	853 000 (×3.65)	1 103 000 (×3.52)	1 127 000 (×3.21)	–
Saber [CHK+21]	ARM M4	658 000 (×2.82)	864 000 (×2.77)	835 000 (×2.38)	–
Saber baseline (Chap. 4)	RISC-V (PULPino)	2 110 283 (×9.04)	2 737 181 (×8.76)	2 797 400 (×7.96)	17 912
Saber opt. (Chap. 4)	RISC-V (PULPino)	760 893 (×3.26)	1 000 043 (×3.20)	1 201 524 (×3.42)	11 802
Saber opt. (Chap. 5)	**RISC-V (PULPino)**	**233 452 (×1.00)**	**312 477 (×1.00)**	**351 370 (×1.00)**	**10 988**
Firesaber [KRSS19]	ARM M4	1 448 776 (×4.13)	1 786 930 (×3.94)	1 853 339 (×3.63)	44 184
Firesaber [BMKV20]	ARM M4	1 340 000 (×3.82)	1 642 000 (×3.62)	1 679 000 (×3.29)	–
Firesaber [CHK+21]	ARM M4	1 008 000 (×2.88)	1 255 000 (×2.77)	1 227 000 (×2.40)	–
Firesaber baseline (Chap. 4)	RISC-V (PULPino)	3 427 099 (×9.78)	4 215 630 (×9.29)	4 328 885 (×8.47)	17 794
Firesaber opt. (Chap. 4)	RISC-V (PULPino)	1 300 272 (×3.71)	1 622 818 (×3.58)	1 898 051 (×3.71)	11 680
Firesaber opt. (Chap. 5)	**RISC-V (PULPino)**	**350 524 (×1.00)**	**453 564 (×1.00)**	**511 088 (×1.00)**	**11 070**

Table 5.8: Cycle count and code size in bytes of optimized masked Kyber and Saber.

Algorithm	Device	Decapsulation unmasked	Decapsulation masked	Generate randomness	Code size masked
Masked RLWE [OSPG18]	ARM M4	4 416 918	25 334 493 (×5.74)	+0 (×5.74)[a)]	–
Kyber-512 (this work)	**RISC-V**	**186 341**	**929 072 (×4.99)**	**+125 770 (×5.66)**	**30 518**
Kyber-768 [BGR+21]	ARM M0	5 530 000	12 208 000 (×2.21)	–	–
Kyber-768 (this work)	**RISC-V**	**313 034**	**1 235 460 (×3.95)**	**+167 190 (×4.48)**	**28 554**
Kyber-1024 (this work)	**RISC-V**	**392 873**	**1 628 467 (×4.15)**	**+200 697 (×4.66)**	**30 314**
Lightsaber (this work)	**RISC-V**	**226 528**	**604 457 (×2.67)**	**+7 154 (×2.70)**	**21 778**
Saber [BDK+21]	ARM M4	1 123 280	2 833 348 (×2.52)	+0 (×2.52)[a)]	–
Saber (this work)	**RISC-V**	**351 370**	**905 395 (×2.58)**	**+9 530 (×2.60)**	**21 042**
Firesaber (this work)	**RISC-V**	**511 088**	**1 156 406 (×2.26)**	**+11 745 (×2.29)**	**20 768**

[a)] Randomness generation included in decapsulation measurement as onboard TRNG available.

addition. For some non-linear operations, Kyber further requires a rejection sampling to obtain uniform randomness modular q.

5.7.3 Side-Channel Leakage Evaluation

This section presents the leakage evaluation for all non-linear operations of this work.

Leakage assessment. The Test Vector Leakage Assessment (TVLA) method [GJJR11] is a well-established procedure to detect side-channel leakages of an implementation. The method is very powerful as it does not require information about the actual implementation or algorithm. It uses Welch's t-test [Wel47] to test the hypothesis that two data sets have the same means. The method first determines a t-value according to

$$t = \frac{\mu_0 - \mu_1}{\sqrt{\frac{\sigma_0^2}{n_0} + \frac{\sigma_1^2}{n_1}}} \quad , \tag{5.19}$$

where μ_0, μ_1 are the means, s_0^2, s_1^2 the variances, and n_0, n_1 the cardinality of the two sets. If the value is larger than a certain threshold, then the null hypothesis, both means are equal, is rejected. The value 4.5 has been established as a threshold in the side-channel community, leading to a confidence of $> 99.999\,\%$ [GJJR11]. In practice, multiple power measurements (called traces) are recorded and categorized into two different sets for the side-channel leakage assessment. The first set $\mathcal{Q}_{fix}$ contains all measurements with a fixed input $x_{fix} = x^0 + x^1$ of any algorithm. The second set $\mathcal{Q}_{rand}$ contains only randomly masked inputs $x_{rand} = x'^0 + x'^1$. The t-value is then evaluated for each time instance (sample point). If the t-value is always lower than the threshold, the implementation is considered secure against first-order univariate attacks for the given number of traces.

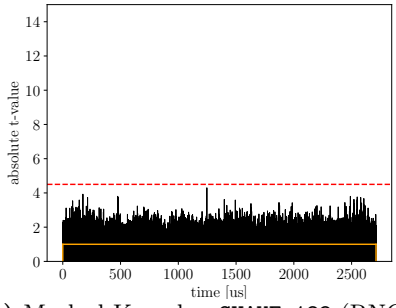
Measurement setup. As already mentioned, the RISC-V design was implemented on a NewAE CW305 target board. The test programs are loaded via SPI to the instruction and data memory of the RISC-V platform. The UART interface is used to send the input data for the TVLA tests from the host PC to the RISC-V target. The Picoscope 6402D USB

oscilloscope with a sampling frequency of 156.25 MHz was used to measure the voltage drop across a 100 mΩ shunt resistor (with a 20 dB low-noise amplifier). For the tests, a total of 100 000 traces were recorded as done in other related works [BDK+21, BGR+21].

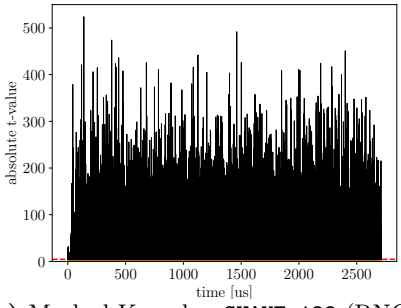
Evaluation results. In order to verify the measurement setup, each leakage test is performed twice: once with activated Random Number Generator (RNG) and once with deactivated RNG. The results for the validation of the first-order SCA resistance of the hardware architectures and the non-linear operations are shown in Figure 5.9. It can be clearly seen that the resulting t-values contain high peaks far above the confidence boundary of $|t| > 4.5$ for the tested operations when the RNG is turned off. This validates the setup and shows that all considered operations are leaking information in an unmasked setting or with deactivated RNG. In order to cover all accelerators and non-linear operations, which require processing two shares simultaneously, the following tests are performed: (i) masked Keccak `SHAKE-128` (includes `f-1600` and Chi accelerators), (ii) masked binomial sampling Ψ_4 (includes Bit-slicing and Binom Tree accelerators), (iii) masked B2A (includes Secure Adder accelerator), (iv) masked B2A$_q$ (includes Secure Adder accelerator), and (v) `MaskedCompress`$_q$ (includes Secure Adder accelerator). Note that the experiment for the compression includes the A2B conversion. Thus, the experiments cover all critical non-linear operations for masking Kyber and Saber. Except for the masked Keccak operation, all experiments of the non-linear operations were performed with 32 polynomial coefficients, which is one function call of the bit-sliced binomial sampler. The masked binomial sampling was measured with Saber parameters $\eta = 4$. The evaluation results with activated RNG show that all implementations remain below the threshold of $|t| < 4.5$. This validates the univariate first-order SCA resistance of the non-linear functions, and thus all corresponding accelerators for the given amount of measurement traces. Stronger adversaries might be able to increase the SNR (e.g., with EM measurements) and the amount of recorded traces. Hence, additional evaluations, which have been left as future work, might be necessary to protect against stronger attacks.

5.8 Summary and Open Problems

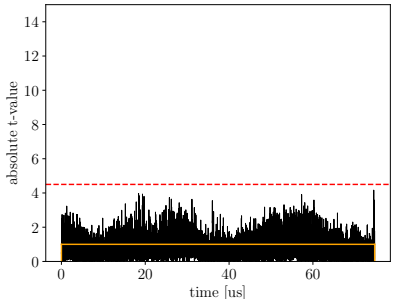
Implementation attacks are a threat to cryptographic implementations. While most implementations already prevent timing attacks, more advanced attacks, such as DPA, are more challenging to protect. This chapter presented the first masked hardware/software codesigns for the two lattice-based NIST PQC finalists Saber and Kyber. It investigated masked tightly coupled accelerators for the non-linear operations to achieve a controlled and efficient execution. This includes the Keccak Chi, binomial sampling with bit-slicing, and secure addition operations. The accelerators are designed to achieve a high flexibility and provide resistance against leakages caused by glitches. As bottlenecks are getting worse in a masked setting, a more powerful accelerator for the polynomial arithmetic of Saber is analyzed. It turned out that a flexible NTT can be efficiently developed that covers most lattice-based algorithms. Future work should consider protection against



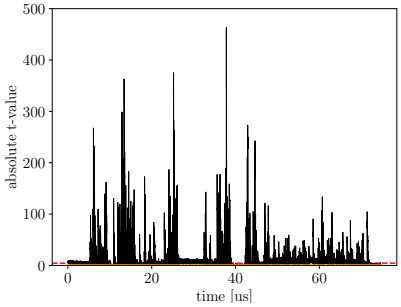
(a) Masked Keccak – `SHAKE-128` (RNG on)



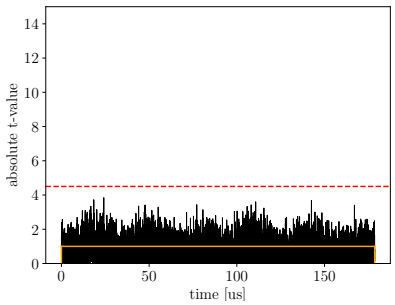
(b) Masked Keccak – `SHAKE-128` (RNG off)



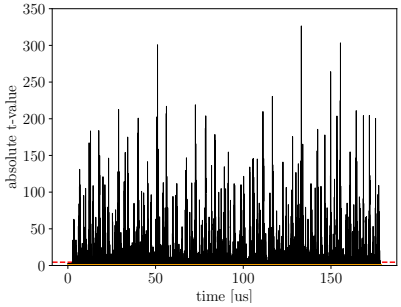
(c) Masked sampling – Ψ_4 (RNG on)



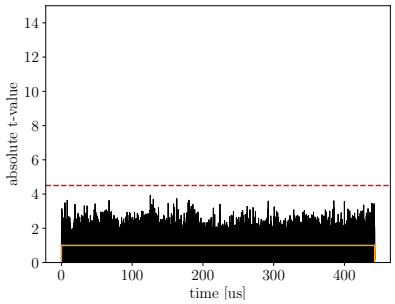
(d) Masked sampling – Ψ_4 (RNG off)



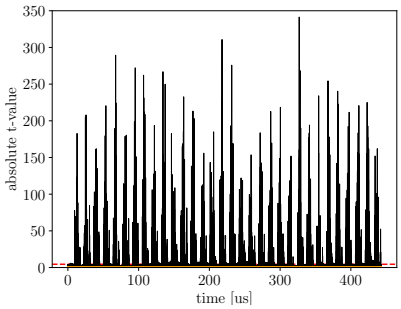
(e) Masked B2A(RNG on)



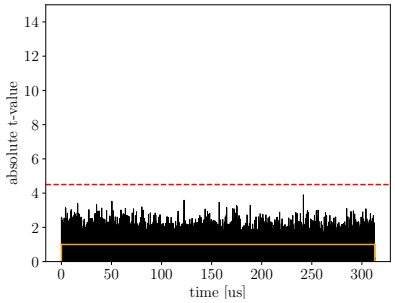
(f) Masked B2A(RNG off)



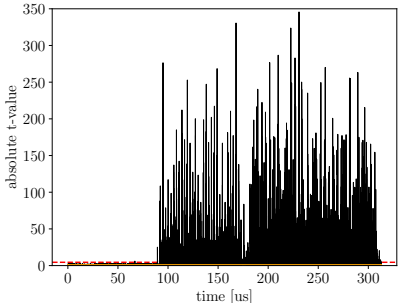
(g) Masked B2A$_q$(RNG on)



(h) Masked B2A$_q$(RNG off)



(i) `MaskedCompress`$_q$ (RNG on)



(j) `MaskedCompress`$_q$ (RNG off)

Figure 5.9: TVLA results for critical non-linear operations and corresponding accelerators (100 000 traces, confidence interval in red and trigger interval in orange).

higher-order attacks and other implementation attacks (e.g., horizontal SPA).

6 Analysis of Error-Correcting Codes for Lattice-Based Cryptography

In this chapter, the application of strong error-correcting codes is analyzed to improve the failure probability, security level, and bandwidth of lattice-based cryptography. Different error-correcting codes are explored, and related advantages and disadvantages are highlighted. As a use case, strong BCH and LDPC error-correcting codes were integrated into the PQC scheme NewHope. This chapter further analyzes the stochastic dependence between decryption errors, which influences the failure rate computations. In order to evaluate the performance of a constant-time BCH code, a timing-attack resistant decoder for the PQC scheme ThreeBears is presented. The error correction exploration for NewHope contains parts of the publication [FPS18], which is based on the author's master's thesis [Fri17] but was majorly revised during the author's Ph.D. studies. The influence of the stochastic dependence of decryption errors includes sections of [MFS20]. Georg Maringer and the author of this dissertation jointly elaborated the ideas and performed the measurements. The author of this thesis focused on the Pearson correlation as an independence measure, which is therefore presented in this work. The section of the constant-time error correction for ThreeBears and the corresponding description of error-correcting codes contains parts of [FVS20, FVFS21].

6.1 Introduction of Error-Correcting Codes for Lattice-Based Cryptography

While the previous chapters focused on improving implementation strategies, this chapter highlights possible algorithmic improvements of PQC. There have been significant parameter set changes during the last years due to advances in the cryptanalysis. Also, implementation aspects and a decrease of key/ciphertext sizes have led to various parameter changes. Thanks to these parameter optimizations and the intensive research in recent years, lattice-based cryptography has become highly efficient in terms of performance when compared to other PKC solutions. A drawback that remains is the relatively large key/ciphertext size (also called bandwidth)—at least large compared to traditional elliptic curve cryptography. Furthermore, it is not yet entirely clear how much the cryptanalysis will advance in the following decades. For this reason, further potential algorithmic optimizations are of great importance. The security of LWE-based schemes relies on the intentional introduction of a noise term. Increasing the variance of the noise improves the security level. In addition, the bandwidth can be optimized by applying a strong compression (see Section 2.2.4). However, raising the variance and applying a stronger compression increases the protocol's failure rate. Using an error-correcting code can help to optimize the characteristics of a scheme while keeping a low failure rate.

Related works. The authors of the lattice-based scheme NewHope recognized early the advantage of an error correction [ADPS16a]. They proposed to map each message bit into four coefficients at the encryption. At the decryption, this allows using the information of four coefficients for the message decoding. This principle corresponds to some kind of analog repetition code, in the remainder denoted as additive threshold encoding. In connection to the NIST call for PQC proposals, several candidates where proposed that explicitly make use of forward error correction. This includes the lattice-based algorithms Hila5 [Saa17], KCL [ZJGS17], ThreeBears [Ham17], and LAC [LLJ+17]. Except for LAC, which uses a powerful BCH code, all schemes mentioned above apply an error correction that can only correct a few errors. Concurrent to these works, the author of this thesis analyzed in [FPS18] the potential of a powerful error correction for NewHope. Further, the error correction capability of ThreeBears was improved in other publications of the thesis author [FVS20, FVFS21]. Together with the analysis of decryption failures in [MFS20] these works build the basis for the content of this chapter. The consecutive work [MPWZ21] further investigated in this direction and determined a theoretical lower bound of the RLWE channel capacity.

Contributions. This chapter investigates the application of error-correcting codes with a high error correction capability for lattice-based cryptography. As a use case, the influence of powerful error-correcting codes for the NewHope version presented in [ADPS16a]

is analyzed. In contrast to other related works, the error correction level is significantly increased to achieve a correction capability close to the channel capacity. This work additionally investigates the stochastic dependence of decryption failures, which complicates the failure rate estimation. Finally, open problems and SCA vulnerabilities of schemes with error correction are discussed.

The contributions can be summarized as:

- Analysis of powerful error correction methods for NewHope to improve its failure rate, security level, and bandwidth;
- Evaluation of NewHope with different combinations of three codes (additive threshold encoding, BCH, and Low-Density Parity-Check (LDPC));
- Investigation of stochastic dependence between decryption failures;
- Exploration of constant-time error correction for ThreeBears.

6.2 Decryption Errors of LWE Schemes

The decryption fails if the predicted message m' of the decryption is not equal to the original message m. LWE-based schemes have two noise sources that are responsible for such decryption errors.

Difference noise. It emerges from the LWE protocol structure, which cannot completely remove all error terms. Assuming the compression is turned off, after the computation of $\texttt{encode}(m') = v' - \boldsymbol{s}^T\boldsymbol{u'}$ in Algorithm 6 (Chapter 2), the largest noise terms cancel out but a relatively small noise term $\alpha = \boldsymbol{e}^T\boldsymbol{s'} - \boldsymbol{s}^T\boldsymbol{e'} + e''$ remains.

Proof. According to Algorithms 4–6 (Chapter 2), the deployed LWE instances are $\boldsymbol{b} = \boldsymbol{A}\boldsymbol{s} + \boldsymbol{e}$, $\boldsymbol{u} = \boldsymbol{A}^T\boldsymbol{s'} + \boldsymbol{e'}$, and $v = \boldsymbol{b'}^T\boldsymbol{s'} + e'' + \texttt{encode}(m)$. The decrypted message can be written as $\texttt{encode}(m') = v' - \boldsymbol{s}^T\boldsymbol{u'}$. When no compression is applied, the following conditions hold $\boldsymbol{b'} = \boldsymbol{b}$, $\boldsymbol{u'} = \boldsymbol{u}$, and $v' = v$. The following computations derive the difference noise.

$$
\begin{aligned}
\texttt{encode}(m') &= v - \boldsymbol{s}^T\boldsymbol{u} \\
&= \boldsymbol{b}^T\boldsymbol{s'} + e'' + \texttt{encode}(m) - \boldsymbol{s}^T(\boldsymbol{A}^T\boldsymbol{s'} + \boldsymbol{e'}) \\
&= (\boldsymbol{A}\boldsymbol{s} + \boldsymbol{e})^T\boldsymbol{s'} + e'' + \texttt{encode}(m) - \boldsymbol{s}^T(\boldsymbol{A}^T\boldsymbol{s'} + \boldsymbol{e'}) \\
&= ((\boldsymbol{A}\boldsymbol{s})^T + \boldsymbol{e}^T)\boldsymbol{s'} + e'' + \texttt{encode}(m) - \boldsymbol{s}^T(\boldsymbol{A}^T\boldsymbol{s'} + \boldsymbol{e'}) \\
&= (\boldsymbol{s}^T\boldsymbol{A}^T + \boldsymbol{e}^T)\boldsymbol{s'} + e'' + \texttt{encode}(m) - \boldsymbol{s}^T(\boldsymbol{A}^T\boldsymbol{s'} + \boldsymbol{e'}) \\
&= \boldsymbol{s}^T\boldsymbol{A}^T\boldsymbol{s'} + \boldsymbol{e}^T\boldsymbol{s'} + e'' + \texttt{encode}(m) - \boldsymbol{s}^T\boldsymbol{A}^T\boldsymbol{s'} - \boldsymbol{s}^T\boldsymbol{e'} \\
&= \texttt{encode}(m) + (\boldsymbol{e}^T\boldsymbol{s'} - \boldsymbol{s}^T\boldsymbol{e'} + e'') = \texttt{encode}(m) + \alpha
\end{aligned}
$$

□

The large term $\boldsymbol{s}^T\boldsymbol{A}^T\boldsymbol{s'}$, which contains the large public matrix A, cancels out, and only α remains. All secret and error polynomials of α are sampled from the error distribution Ψ_η. Usually, η is relatively small such that the decode operation returns with a high probability $m' = m$. An increase of the variance $\sigma^2 = \eta/2$ of the error distribution directly increases the error term α.

Difference noise. It is caused by the $\texttt{compress}_q$ operation. Let $\boldsymbol{c_b}$, $\boldsymbol{c_u}$, and c_v be uniformly distributed compression noise that appears when compressing and decompressing the elements $\boldsymbol{b}$, $\boldsymbol{u}$, and v, respectively. Then, the compression noise remaining at the end of the decryption is $\beta = \boldsymbol{c_b}^T\boldsymbol{s'} - \boldsymbol{s}^T\boldsymbol{c_u} + c_v$.

Proof. For this proof, the compression is turned on, i.e., $\boldsymbol{b'} = \boldsymbol{b} + \boldsymbol{c_b} = \boldsymbol{As} + \boldsymbol{e} + \boldsymbol{c_b}$, $v' = v + c_v = \boldsymbol{b'}^T\boldsymbol{s'} + e'' + \texttt{encode}(m) + c_v$, and $\boldsymbol{u'} = \boldsymbol{u} + \boldsymbol{c_u} = \boldsymbol{A}^T\boldsymbol{s'} + \boldsymbol{e'} + \boldsymbol{c_u}$. The following computations derive the compression noise.

$$
\begin{aligned}
\texttt{encode}(m') &= v' - \boldsymbol{s}^T\boldsymbol{u'} \\
&= \boldsymbol{b'}^T\boldsymbol{s'} + e'' + \texttt{encode}(m) + c_v - \boldsymbol{s}^T(\boldsymbol{A}^T\boldsymbol{s'} + \boldsymbol{e'} + \boldsymbol{c_u}) \\
&= (\boldsymbol{As} + \boldsymbol{e} + \boldsymbol{c_b})^T\boldsymbol{s'} + e'' + \texttt{encode}(m) + c_v - \boldsymbol{s}^T(\boldsymbol{A}^T\boldsymbol{s'} + \boldsymbol{e'} + \boldsymbol{c_u}) \\
&= \boldsymbol{s}^T\boldsymbol{A}^T\boldsymbol{s'} + \boldsymbol{e}^T\boldsymbol{s'} + \boldsymbol{c_b}^T\boldsymbol{s'} + e'' + \texttt{encode}(m) + c_v - \boldsymbol{s}^T\boldsymbol{A}^T\boldsymbol{s'} - \boldsymbol{s}^T\boldsymbol{e'} - \boldsymbol{s}^T\boldsymbol{c_u} \\
&= \texttt{encode}(m) + \alpha + (\boldsymbol{c_b}^T\boldsymbol{s'} - \boldsymbol{s}^T\boldsymbol{c_u} + c_v) = \texttt{encode}(m) + \alpha + \beta
\end{aligned}
$$

□

In the remainder of this chapter, the overall noise term is denoted as $d = \alpha + \beta$.

Computation of the failure rate. One method to compute the failure rate of lattice-based cryptography is to use the Cramér–Chernoff inequality as in [ADPS16a]. More recent works directly determine the probability distribution of the noise term [Saa17, ABD+20]. This method is also applied in this work. In the remainder of this chapter, only RLWE instances are considered. The overall failure rate of an RLWE-based scheme depends on s, s', e, e', e'', c_b, c_u, and c_v. For all coefficients of these variables, the initial distribution is known. While the secret and error terms are sampled from a binomial distribution, the compression terms can be expressed as uniform samples. The distribution of the overall noise term for a single coefficient can be exactly computed using the following theorems.

Theorem 1 (Addition of random variables)**.** *Let $\Psi_X(x)$ and $\Psi_Y(y)$ be two probability distributions of the independent random variables X and Y. Then, the probability distribution of the sum of both random variables corresponds to the convolution of the individual probability distributions, which can be written as $\Psi_Z(z) = \Psi_{X+Y} = \Psi_X(x) \circledast \Psi_Y(y)$ [GHW12].*

Theorem 2 (Product distribution)**.** *Let $\Psi_X(x)$ and $\Psi_Y(y)$ be two probability distributions of the independent random variables X and Y. Then, the product distribution $\Psi_Z(XY = c) = \sum_{x\in X, y\in Y \text{ s.t. } xy=c} \Psi_X(x)\Psi_Y(y)$.*

Theorem 3 (Polynomial product distribution)**.** *Let a and b be two polynomials of the ring $\mathcal{R}_q = \mathbb{Z}_q/\langle x^n + 1\rangle$ with degree $n-1$ and independent coefficients randomly sampled from Ψ_η. Let c be the result of the polynomial multiplication of a and b. Then, the probability distribution of an arbitrary coefficient of c equals the n-fold convolution of the product distribution Ψ_Z of two random variables sampled from Ψ_η.*

Proof Theorem 3. Let $a, b \in \mathbb{Z}_q/\langle\phi(x)\rangle$ with $\phi(x) = x^n + 1$ and with coefficients sampled from the probability distribution Ψ_η. The multiplication of these polynomials results in $c = (a_0 + a_1x + \cdots + a_{n-1}x^{n-1})(b_0 + b_1x + \cdots + b_{n-1}x^{n-1})$. Using the distributive law and grouping all terms with the same degree together, this equation can be written as $c = (a_0b_0 + \cdots - a_{n-2}b_2 - a_{n-1}b_1) + (a_0b_1 + \cdots - a_{n-2}b_3 - a_{n-1}b_2)x + \cdots + (a_0b_{n-1} + \cdots + a_{n-2}b_1 + a_{n-1}b_0)x^{n-1}$. A sum of n products determines each coefficient of polynomial c. Since all coefficients of a and b are independently sampled from the probability distribution Ψ_η, the probability distribution of the coefficients of c is an n-fold convolution of the product distribution of two random variables sampled from Ψ_η. □

Computing the failure rate. The overall noise term d can be represented as a polynomial with n coefficients. A decryption error occurs if the absolute value of one noise coefficient d_i is greater than $q/4$. In this case, the `decode` operation maps the respective coefficient to the wrong message bit.

Let p_b be the failure probability of each single bit and let $P[\overline{d}_f]$ be the probability that one or more bits fail (overall failure probability). Given p_b, the Fréchet inequality can be applied to compute an upper bound of the overall failure probability:

$$\begin{aligned} P[\overline{d}_f] &= P[|d_0| > q/4 \ \cup \ |d_1| > q/4 \ \cup \cdots \cup \ |d_{n-1}| > q/4] \\ &\leq \min(1, P[|d_0| > q/4] + P[|d_1| > q/4] + \cdots + P[|d_{n-1}| > q/4]) \\ &= \min(1, n \cdot p_b) \end{aligned} \tag{6.1}$$

Let us assume the independence of decryption failures. Then, for RLWE-based schemes using a t-bit error correction, the probability that a binary vector of S bits (in this analysis 256 message bits) has more than t errors is

$$P_t[\overline{d}_f] = \sum_{i=t+1}^{S} \binom{S}{i} p_b^i (1-p_b)^{S-i} = 1 - \sum_{i=0}^{t} \binom{S}{i} p_b^i (1-p_b)^{S-i} \ . \tag{6.2}$$

Desired failure rates. Ephemeral key exchanges and PKE have different requirements for the failure rate. The presence of errors is less critical for ephemeral key exchanges, and a failure rate of $\approx 2^{-40}$ can be acceptable since two parties can repeat the key exchange in the rare case of an error. The issue of decryption errors is more critical in a PKE setting. In order to protect against CCA, PKE schemes use the Fujisaki–Okamoto method [FO99, TU16] to transform from CPA to CCA security. A CCA secure cryptosystem that uses this transformation requires a negligible failure rate to avoid attacks [Flu16]. The failure rate is therefore desired to be lower than 2^{-128}. In this chapter, a failure rate lower than 2^{-140} is targeted for the PKE setting to have a margin.

An overall failure rate of 2^{-128} or 2^{-140} means that the bit failure rate p_b must be n times smaller if no error correction is applied. If an error correction is applied, the acceptable bit failure rate depends on the error correction capability. In order to apply Equation 6.2, the independence of decryption errors must be assumed, as in [Saa17, LLJ+17]. The influence of the dependence between decryption errors on the failure rate computation is still an open research question and is further discussed in Section 6.5.

6.3 Exploration of Error-Correcting Codes

Error-correcting codes realize reliable data transmissions over noisy channels. Instead of the additive threshold encoding used in NewHope, the effect of more powerful error-correcting codes is investigated. The desired characteristics of the error-correcting code are: (i) good error correction capability to improve the security level and bandwidth; (ii) low failure rate to avoid protocol repetitions and to apply CCA transformations; and (iii) reasonable time complexity.

Modern codes (LDPC codes). Modern codes based on probabilistic coding theory have a strong error correction capability. They can get close to the channel capacity for long code lengths. The most commonly used codes of this class are LDPC and Turbo codes. Compared to Turbo codes, LDPC codes usually have a lower time complexity [Fan12], and their error floor occurs at lower failure rates [LC04]. The error floor is a phenomenon of some modern codes that limits the performance for low failure rates. For these reasons, LDPC codes in favor of Turbo codes are selected for the investigation performed in this chapter.

LDPC codes were developed in [Gal62]. A block diagram of this code category is provided in Figure 6.1. LDPC codes are characterized by their parity check matrix H, which has a low density, i.e., a low number of ones. For the encoding, usually, the systematic form of H is computed to derive the generator matrix. The generator matrix is then used to compute the codeword c for a given message m. In the noisy channel, noise is added to the transmitted codeword. At NewHope, this would be the difference and compression noise. The sum-product algorithm is an efficient soft decision message-passing decoder. It takes as input a parity check matrix, the maximum number of iterations, and the Log-Likelihood Ratio (LLR) of the received codeword r. The decoding process can be illustrated as a bipartite graph with Check Nodes (CN) and Variable Nodes (VN), representing the rows and columns of H, respectively. The sum-product algorithm iteratively sends LLR messages from variable nodes to check nodes and vice versa until a correct codeword is found or the maximum number of iterations is reached. A full description of the algorithm can be found in works like [HEAD01, QLW13].

Classical codes (BCH codes). The advantages of algebraically structured classical error-correcting codes are that they have no error floor and that the number of correctable errors can be determined during the construction of the code. When the number

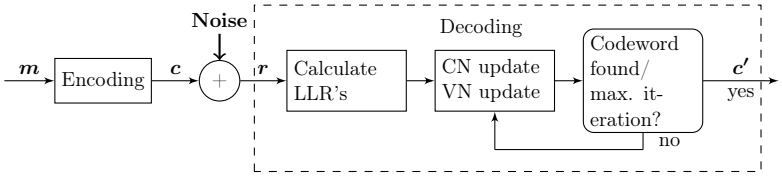


Figure 6.1: LDPC error correction with sum-product algorithm.

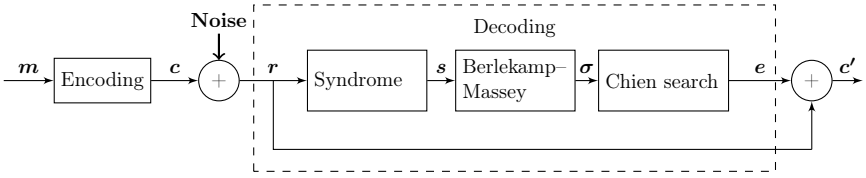


Figure 6.2: BCH error correction.

of correctable errors is known, the achievable failure rate of the code can be calculated. Otherwise, simulations are required.

BCH codes are a class of powerful classical error-correcting codes that are widely used in real-world applications due to their good performance and flexibility in terms of code length and code rate. These characteristics make the BCH code a good choice for protocols with very low failure rates. Figure 6.2 illustrates the encoding and decoding process of BCH codes. During the encoding, the codeword c is built out of a message m. The decoder is used to correct multiple errors in the received codeword r. The decoding process usually consists of three parts: computation of syndrome s, computation of error locator polynomial σ, and finding the zeros of σ (see Section 4.2).

Code concatenations. Different codes can be concatenated to achieve both a high error correction capability and a very low failure rate. The concatenation of BCH and LDPC codes is a common method, which is used, e.g., in the second generation of the digital video broadcast standard for satellite (DVB-S2).

6.4 Analysis for the Post-Quantum Scheme NewHope

Different design options using LDPC and BCH codes are investigated to maximize the error correction capability of NewHope and to achieve very low failure rates. The respective advantages and disadvantages are summarized in Table 6.1.

6.4.1 NewHope Compression Noise

Figure 6.3 illustrates the influence of the compression noise on the protocol's failure rate $P[\overline{d}_f]$. The compression is particularly visible for low values of η. For higher values, the difference noise dominates. In order to improve both the security level and bandwidth, a balance between the difference noise and compression noise must be found. When applying the described error correction options, a good trade-off was found at a compression

Table 6.1: Summary of explored coding options.

Option	Coding technique	Advantages	Disadvantages
Option 1:	BCH	Good error correction	Computationally expensive
Option 2:	BCH + additive threshold encoding	Faster than Option 1 (lower Galois field)	Weaker error correction than Option 1
Option 3:	LDPC	Closer to channel capacity than Options 1/2	Does not achieve very low error rates
Option 4:	LDPC + BCH	Lower error rates than Option 3 achievable	Computationally expensive

of v from 14-bit to 3-bit per coefficient and a compression of u from 14-bit to 10-bit per coefficient. The curve with compression of v corresponds to the original implementation of the NewHope specification in [ADPS16a]. Note that the compression noise c_u is magnified by a multiplication with the secret polynomial. Hence, the compression must be lower than the one of v. Due to some concerns regarding security proofs, public-key compression (on polynomial b) is currently not recommended [ABD+19].

6.4.2 NewHope with BCH Code

Figure 6.4 shows the improvements when BCH codes are used in NewHope. Option 1 uses a BCH(1023,258) capable of correcting up to 106-bit errors. The 256-bit message is first encoded into a 1023-bit codeword. Each codeword bit is then mapped to one coefficient (polynomial length is 1024). As computations in GF(2^{10}) are more complex than in GF(2^9), Option 2 uses the smaller BCH(511,259) capable of correcting up to 30 errors. As the code length is 511, each bit can be mapped to two coefficients. This allows further benefiting from an additive threshold encoding. The results show that both BCH variants (Options 1 and 2) allow a quasi-error-free communication for $\eta \leq 46$. While NewHope with compression of v and u has a failure rate of $1.69 \cdot 10^{-3}$ for $\eta = 46$, Options 1 and 2 achieve a failure rate of $1.83 \cdot 10^{-57}$ and $2.30 \cdot 10^{-44}$, respectively. Thus, it is possible to significantly increase η and consequently the security level.

6.4.3 NewHope with LDPC Code

Frequently, the error correction capability of an error-correcting code is first evaluated for the basic Binary Input Additive White Gaussian Noise Channel (BI-AWGNC) model. The combination of the signal-to-noise ratio and achieved bit error rate is a suitable performance indicator for the error correction capability of a code. Simulations are used to evaluate the error correction improvement of the LDPC code over the BCH code. For the code rate 1/2, the simulated results have shown that the applied LDPC(1024,512) code achieves an improvement over the BCH$(1023, 513)$ code of about 2.8 dB at a bit error rate of 10^{-6}. For the code rate 1/4, the improvement of the LDPC(1024,256) code over the BCH(1023,258) code is about 3.8 dB. This verifies that the LDPC code is able

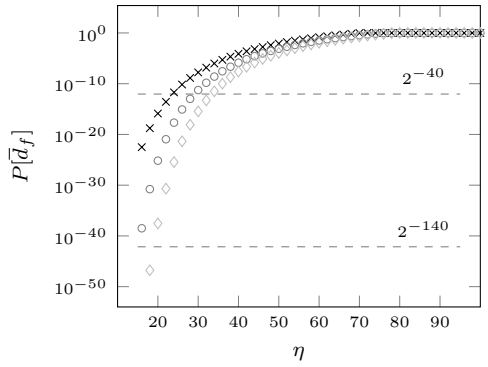


Figure 6.3: NewHope compression influence.

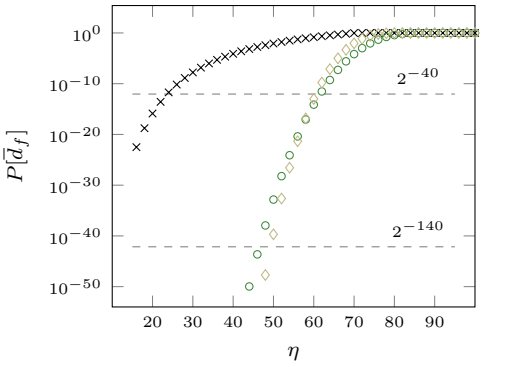


Figure 6.4: Improvement Options 1 and 2 (compression of v and u).

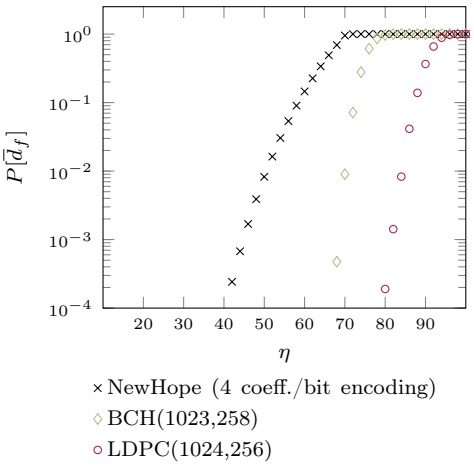


Figure 6.5: Improvement Option 3 (compression of v and u).

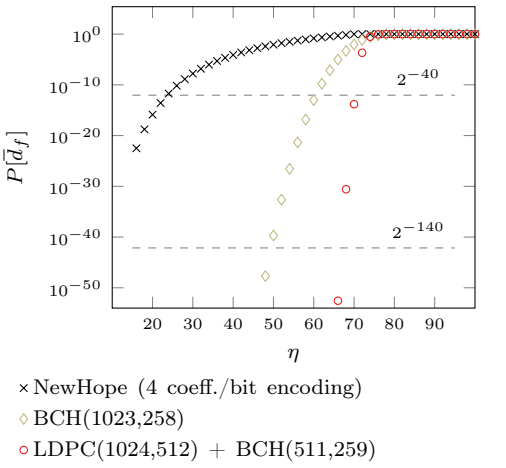


Figure 6.6: Improvement Option 4 (compression of v and u).

to improve the error correction capability. The next paragraph analyzes if this is also true for the channel model of NewHope.

Figure 6.5 compares the threshold encoding of NewHope with the implementations of NewHope using an LDPC code (Option 3) and BCH code (Option 1). The graph shows that LDPC codes can be used to further improve the error correction performance. While the BCH(1023,258) begins to operate in the *waterfall region* for $\eta < 76$, the *waterfall region* for the LDPC(1024,256) begins at $\eta < 92$. The error floor is expected to limit the performance of the LDPC code for error rates smaller than about 10^{-10} (see analysis in [Ric03]) such that BCH codes usually perform better for such a low failure rate. Note that for the PKE setting, a failure rate of $2^{-140} \approx 7 \cdot 10^{-43}$ is targeted. Unfortunately, no mathematical formula for the real failure probability of the LDPC code exists. Simulations for a failure rate of 10^{-10} are extremely time-consuming, and simulations for a failure rate of $7 \cdot 10^{-43}$ are even impossible. However, the concatenation of BCH and LDPC code circumvent this problem, as discussed in the next paragraph.

6.4.4 NewHope with Concatenation of BCH and LDPC Code

To achieve very low error rates and to get closer to the channel capacity, the BCH and LDPC codes are combined (Option 4). Figure 6.6 illustrates the performance of the code concatenation LDPC(1024,512) and BCH(511,259). The BCH code is intended as inner code and the LDPC code as outer code.

6.4.5 Comparison Coding Options

Table 6.2 summarizes the results of the different coding options. In order to achieve a failure rate of smaller than 2^{-140}, parameter η is set for Options 1, 2, and 4 to 48, 46, and 66, respectively. Such low failure rates cannot be proven for the pure LDPC implementation (Option 3) as it solely relies on simulations, and exact formulas do not exist. Option 1 has only a slightly better security strength than Option 2. Therefore, Option 2 can be seen as more suitable as it has a lower time complexity due to the smaller Galois field arithmetic. Option 3 achieves the best error correction capability for moderate failure rates, but the error floor limits the performance for failure rates lower than $\sim 10^{-10}$. Option 4 cannot get as close to the channel capacity as Option 3, but it achieves extremely low error rates. Option 4 achieves an error rate of 2^{-140}, increases the post-quantum security bit level by 20.39 %, and decreases the communication overhead by 12.80 %. If η and thus the security level is left unchanged, and only the compression on u is increased, the communication overhead can be reduced with Option 4 by 19.20 %.

6.5 Discussion and Open Problems

The application of strong error-correcting codes for lattice-based cryptography can lead to significant advantages, as shown in the previous section. On the other hand, they introduce some problems that are still not solved. The next sections present an analysis of these issues.

Table 6.2: Comparison error correction options.

Coding option	$P[\bar{d}_f]$	η	Security Classic/PQ	Exchanged bytes
NewHope Simple [ADPS16a]	$2^{-127.88}$	16	281/255 bits	4 000
Option 1, BCH(1023,258)	$< 2^{-140}$	48	324/294 bits	3 488
Option 2, BCH(511,259) + 2 coeff./bit encoding	$< 2^{-140}$	46	323/292 bits	3 488
Option 3, LDPC(1024,256)	$< 2^{-12}$ a)	80	348/315 bits	3 488
Option 4, LDPC(1024,512) + BCH(511,259)	$< 2^{-140}$	66	338/307 bits	3 488

a) With Option 3 a failure rate of $\sim 10^{-10} = 2^{-33.22}$ can be efficiently reached.

6.5.1 Stochastic Dependence of Decryption Errors and its Impact on the Failure Rate Analysis

The noise term discussed in the previous sections can be constructed using polynomial arithmetic. Each of the coefficients of the involved polynomials is independently sampled. The polynomial addition preserves the independence between coefficients. But this is not true for the polynomial multiplication. The proof of Theorem 3 shows that after the polynomial multiplication, e.g., $c_0 = (a_0b_0 + \cdots - a_{n-2}b_2 - a_{n-1}b_1)$ and $c_1 = (a_0b_1 + \cdots - a_{n-2}b_3 - a_{n-1}b_2)$. It is directly visible that the independence between the coefficients is not preserved as both c_0 and c_1 contain, e.g., the value a_0 in their equations. The probability distribution of a single coefficient from the noise term can be exactly determined as described in Section 6.2. Consequently, the single bit error rate can be computed as well. However, the joint probability distribution of all coefficients of the noise term is not easy to determine.

Lattice-based schemes without a t-bit error correction can use inequalities to bound the joint failure rate (see Equation 6.1). To apply Equation 6.2 for schemes with such an error correction, independence between the failures must be assumed. In other words, the stochastic dependence between decryption failures must be at least extremely low. The authors in [DVV19] have shown that the failure rate of the PQC scheme LAC, which uses a BCH code, was in the first round NIST submission higher than expected. The reason is that the stochastic dependence between decryption failures was underestimated for the parameter set of LAC. The authors further recognized that the norm of the noise polynomial directly influences the stochastic dependence of decryption failures. Based on the results in [DVV19], the LAC second-round submission fixed the norm of the polynomials to reduce the dependence between decryption failures. The correlation between the failures is measured in this work to verify that this approach indeed reduces the dependence between decryption failures. Moreover, the influence of the RLWE parameters (n, q, η) is investigated.

Pearson correlation as measure for stochastic dependence. While the exact computation of the correlation between decryption failures is not straightforward, stochastic methods can be used. The Pearson correlation is a well-known method for determining the linear correlation between two data sets. It is thus suitable to quantify the stochastic dependence between decryption failures. Given two sets of samples of the random variables X with samples x_i and Y with samples y_i, the Pearson correlation can be computed with

$$r_{xy} = \frac{\sum_{i=0}^{n-1}(x_i - \overline{x})(y_i - \overline{y})}{\sqrt{\sum_{i=0}^{n-1}(x_i - \overline{x})^2}\sqrt{\sum_{i=0}^{n-1}(y_i - \overline{y})^2}} \ , \tag{6.3}$$

where $\overline{x} = \frac{1}{n}\sum_{i=0}^{n-1} x_i$ and $\overline{y} = \frac{1}{n}\sum_{i=0}^{n-1} y_i$ [Uni21]. The value of the Pearson correlation ranges between -1 and 1. The correlation between the data sets decreases with the absolute value of r_{xy}.

Test setup. For the experiments, the first two coefficients of the difference noise $d = es' - se' + e''$ are sampled n times (RLWE setting without compression). The samples of d_0 (random variable X) and d_1 (random variable Y) are then mapped to one element of the set $\{S, F\}$ depending on whether a decryption failure occurs (F) or not (S). A failure occurs if $abs(d_i) > q/4$. This results into four possible combinations of XY: F_0F_1, F_0S_1, S_0F_1, and S_0S_1. The number of samples are increased such that the result of the Pearson correlation converges. The smaller the Pearson correlation gets, the smaller is the linear dependence between decryption failures of the sampled coefficients d_0 and d_1. It is difficult to determine a value of the Pearson correlation at which the dependence between decryption errors can be neglected for the failure rate analysis of LWE schemes with error correction. However, different sampling methods and parameter sets can be compared with this approach. Further, it can be evaluated which methods/parameters have a stronger dependence between decryption errors.

Fixing the norm of the polynomials. In order to demonstrate the problem of the stochastic dependence between decryption failures, the overall decryption failure probability $P[\overline{d}_f]$ depending on the error correction capability is determined with two different methods. The first method assumes independence between decryption failures and uses Equation 6.2 (Section 6.2). The second method performs an exhaustive experimental test and approximates the real failure probability. The decryption failure probability is evaluated for the LAC instance LAC-256 (NIST Level V). The following analysis distinguishes between the sampling method of LAC Round 1 and of LAC Round 2 (NIST rounds). The results are illustrated in Figure 6.7. For the sampling used in LAC Round 1, it can be observed that there is a mismatch between the computed failure probability using the independence assumption and the exhaustive experimental test. The reason for this mismatch is the stochastic dependence between decryption errors. In LAC Round 2, the norm of the polynomials sampled from the error distribution is fixed. For this case, the experimental values match the computed ones, implying a decrease in the dependence of decryption errors. This assumption matches with the measurement results

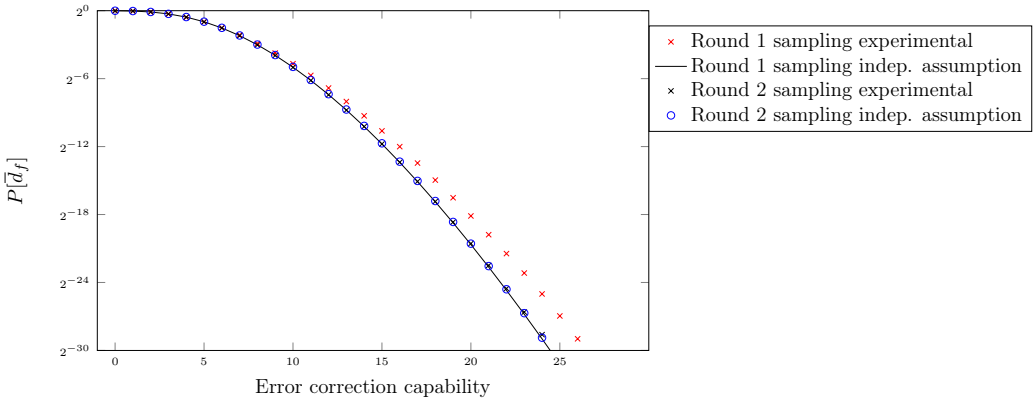


Figure 6.7: Decryption failure probability of LAC-256 depending on the error correction capability.

Table 6.3: Pearson correlation for LAC-128 and LAC-256 (Round 1/2), 10^{11} samples.

Error distribution	Pearson (abs)	$\mathbf{P}[F_0F_1]$	$\mathbf{P}[F_0S_1]$	$\mathbf{P}[S_0F_1]$	$\mathbf{P}[S_0S_1]$
LAC-128 Round 1	8.852e-06	9.230e-09	9.170e-05	9.178e-05	0.99982
LAC-128 Round 2	5.083e-06	7.430e-09	8.874e-05	8.879e-05	0.99982
LAC-256 Round 1	1.032e-04	3.201e-05	5.575e-03	5.575e-03	0.98882
LAC-256 Round 2	6.077e-06	3.143e-05	5.572e-03	5.572e-03	0.98882

given in Table 6.3. Fixing the norm of the error distribution reduced the Pearson correlation from $1.032 \cdot 10^{-04}$ to $6.077 \cdot 10^{-06}$. For LAC-128 Round 2 (NIST Level I), the Pearson correlation decreased to a value of $5.083 \cdot 10^{-06}$. These results imply that fixing the norm of the sampling process increases the accuracy of the failure rate analysis using independence assumption.

Influence of LWE parameters. The Pearson correlation is determined for different parameter sets to evaluate the influence of the LWE parameters (n,q,η) on the dependence between decryption failures. The parameter set of LAC-256 with ($n = 1024$,$q = 251$,$\eta = 1$) is used as a baseline. In order to evaluate the influence of the polynomial length, n is set to the typical values $\{512, 768, 1024\}$. For the modulus q, the values $\{251 - 20, 251, 251 + 20\}$ are evaluated. Finally, the influence of η is tested for the set $\{1, 2\}$. Note that the LWE parameters cannot be arbitrarily chosen for this test. In order to obtain stable results for the Pearson correlation, the test set must contain a sufficiently large amount of decryption failures. Table 6.4 summarizes the influence of the parameter set on the stochastic dependence between decryption failures. The results show that a decrease of the polynomial length n, an increase of the modulus q, and a decrease of the noise distribution parameter η lead to a smaller Pearson correlation, thus to a smaller dependence. Generally, the stochastic dependence between decryption failures

Table 6.4: Pearson correlation for different parameter sets (n,q,η), 10^{11} samples.

Parameter set	Pearson (abs)	$\mathbf{P}[F_0F_1]$	$\mathbf{P}[F_0S_1]$	$\mathbf{P}[S_0F_1]$	$\mathbf{P}[S_0S_1]$
(**512**,251,1)	8.852e-06	9.230e-09	9.170e-05	9.178e-05	0.99982
(**768**,251,1)	5.445e-05	2.005e-06	1.387e-03	1.387e-03	0.99722
(**1024**,251,1)	1.032e-04	3.201e-05	5.575e-03	5.575e-03	0.98882
(1024,**231**,1)	1.414e-04	1.180e-04	1.068e-02	1.068e-02	0.97853
(1024,**251**,1)	1.032e-04	3.201e-05	5.575e-03	5.575e-03	0.98882
(1024,**271**,1)	6.897e-05	7.947e-06	2.777e-03	2.777e-03	0.99444
(1024,251,**1**)	1.032e-04	3.201e-05	5.575e-03	5.575e-03	0.98882
(1024,251,**2**)	5.512e-04	2.751e-02	0.13816	0.13816	0.69617

increases with the failure probability. The applied method helps designers in selecting the parameter set. The results show that the parameter set does not only influence the performance and security level but also the failure rate analysis.

6.5.2 Side-Channel Vulnerability and Implementation Aspects

Each new component of a cryptographic algorithm introduces a potential point of attack. As already described in Section 4.2.4, constant-time error-correcting codes whose runtime is independent of the input message, the codeword, and the number of errors must be developed to avoid timing side channels. The authors in [WR20] developed a constant-time BCH code for LAC. In this work, the BCH code of ThreeBears, which is initially capable of correcting two errors, is extended to correct up to six errors and protected against timing attacks.

Constant-time error correction. Typical exploitable operations for a timing attack are if conditions and loop operations. The BCH implementation of this chapter is based on the non-protected implementation in [Glo11]. The encoding of the BCH code is straightforward to implement with a constant runtime. It can be realized with the principles of an LFSR. The execution time of such structures is usually independent of the input message. The decoding time natively depends on the received codeword, the number of errors, and the computed syndromes. This is particularly true for the Berlekamp–Massey and the Chien search algorithms. Both algorithms are modified to behave always as for the maximum number of errors. In order to achieve this, the loops were unrolled, and if conditions were modified such that all cases always execute. Invalid results are simply discarded (not used).

The extended timing protected BCH code of ThreeBears was implemented on the Automotive Realtime Integrated NeXt Generation Architecture (AURIX). AURIX is one of the most popular automotive microcontrollers in the automotive industry. Table 6.5 illustrates the timing variations of the unprotected baseline implementation and the constant cycle count of the protected version. When six errors are corrected, the protected

Table 6.5: Cycle count of the BCH code with 6-bit error correction capability.

#Errors	0	1	2	3	4	5	6
Baseline	1 383	3 764	4 188	18 771	22 146	24 409	27 267
Protected	208 993						

version is a factor of 7.66 slower than the baseline. The result illustrates that the protection for this code is relatively costly. Nevertheless, the error correction is not always the bottleneck of the cryptographic scheme. Further, hardware accelerators, as shown in Section 4.2, can be deployed. A more detailed analysis of the error correction in ThreeBears can be found in the publications of the thesis author [FVS20, FVFS21].

Open challenges. Timing attacks on the error correction of PQC were presented in [DTVV19, PT19, WTBB+20]. While these attacks can be prevented with a constant-time implementation, more powerful SCA methods, such as power or EM attacks, might still be applicable. Practical attacks of this category on the error correction have been demonstrated in [RSRCB20, SRSWZ20]. Developing protection mechanisms against these attacks is still an open research area.

The code-based cryptography schemes LEDAcrypt [BBC+18] and BIKE [ABB+20] rely on the LDPC variants QC-LDPC and QC-MDPC, respectively. Both schemes use a decoder implementation based on the bit-flipping algorithm. Prior works have already developed constant-time implementations for this decoder type [RMGS20, ZGF20]. In Section 6.3, a sum-product decoder was used instead of a bit-flipping approach. It typically achieves better error correction capabilities as it uses soft-decision information. When mapping the coefficients of the received codeword directly to binary values, information gets lost. This information loss is avoided at soft-decision decoders. Unfortunately, constant-time implementations of this kind of decoder have not been found so far. A detailed analysis of protection mechanisms is still future work.

6.6 Summary

This section demonstrated the potential of a strong error-correcting code for lattice-based cryptography. LWE-based schemes have an unavoidable noise term. This noise term is influenced by the LWE parameters and the applied ciphertext compression. Increasing the noise term and compression leads to a high security level and small bandwidth, respectively. On the other hand, it increases the protocol's failure rate. Error-correcting codes can be used to optimize the failure rate and thus the security level and bandwidth. This chapter analyzed the influence of a combination of powerful error-correcting codes for the PQC scheme NewHope. Modern LDPC codes were used to achieve an error correction capability close to the channel capacity. For a low failure rate, classical BCH codes were deployed. Combining both code categories increased the security level against quantum attacks from 255-bit to 307-bit and decreased the bandwidth by 12.8 %.

The dependence between decryption errors complicates an accurate computation of the failure rate when error-correcting codes are involved. This chapter verified that fixing the norm of the sampled noise polynomials as done in the LAC second-round NIST submission decreases the stochastic dependence between decryption failures. Moreover, the influence of the LWE parameters on the stochastic dependence between decryption errors was analyzed to provide a clearer picture for LWE designers.

Further, it was shown how constant-time implementations of the error correction help to resist against timing attacks. Despite this effort, strong error correction for lattice-based PQC is still an active research area. The error correction introduces another point of attack that has not been sufficiently analyzed so far. For this reason, NIST did not select any lattice-based scheme with forward error correction. Currently, the confidence in such schemes is not sufficiently large for the NIST standardization. This might change in the future. Due to low ciphertext sizes, particularly applications with a low data transmission rate would benefit from further research in this direction.

7 PQC Migration and Real-World Applicability

This chapter illustrates the suitability of PQC for real-world applications. At the beginning of this chapter, the first PQC chip that is based on a hardware/software codesign approach with instruction set extensions is presented. The results show that the bare die size of the complete PQC microcontroller is only $3.01\,mm^2$*. The chapter further discusses the integration of PQC into the popular AURIX microcontroller, which is widely used in the automotive industry. The content of this part was published in [FVS19, FVFS21]. Finally, this chapter explores a hardware architecture for hybrid key encapsulation to combine the security of trusted traditional cryptography and new quantum-resistant cryptography. The corresponding part is based on the publication [OFP+22], which was submitted to the Journal of Cryptographic Engineering (JCEN).*

7.1 Introduction of PQC for Real-World Applications

The previous chapters discussed how to improve fundamental design characteristics of PQC, including speed, energy consumption, protection against SCA, security level, and bandwidth. This chapter focuses on the question of whether PQC is already suitable for real-world applications.

Most NIST submissions already demonstrate the suitability of PQC for powerful standard computers. Also, the Supercop platform provides benchmark results for the NIST finalists on different high-end platforms. For the Intel Xeon E-1220 CPU running at a frequency of 3.1 GHz, the Supercop project reports 364 761 cycles for Firesaber, 195 220

cycles for Kyber-1024, 495 351 cycles for ntruhps4096821, and 234 501 487 cycles for mceliece8192128f [BL21]. Even the slowest PKE/KEM finalist with the highest parameter set requires less than 76 ms for the complete algorithm execution on such a platform.

The suitability of PQC for constrained applications with low computing power is not so apparent as for powerful platforms. They usually have frequencies in the range of a megahertz up to few hundred megahertz. For example, at a typical frequency of 25 MHz, the baseline software implementation of Kyber-1024 on PULPino (RISC-V) would require 454 ms. This will not be fast enough for many applications. It was already shown in the previous chapters that hardware acceleration could be used to meet performance and energy requirements. Only 47 ms or even less are required for the accelerated version of Kyber-1024 at such a frequency (see Chapter 4).

Post-quantum chip. The previous chapters already illustrated what hardware acceleration achieves. This chapter increases the confidence in the proposed hardware/software codesign approach and shows its suitability for real-world applications. For instance, Section 7.2 presents a real ASIC tapeout of the design presented in Chapter 4. This ASIC verifies the suitability for mass production and helps to estimate area costs. Due to the good performance/area characteristics and the generic RISC-V core, the chip targets various embedded applications. The design and presented ideas could be, e.g., used in a Hardware Security Module (HSM) or smartcard to securely handle cryptographic operations.

Microcontroller applications. The applicability of PQC has been demonstrated for several microcontrollers with moderate computing power. As described in the previous chapters, the ARM Cortex-M4 microcontroller has been the main evaluation platform. But PQC was also already deployed on other platforms. For instance, the authors of [AHH+18] investigated the application of Kyber on the security microcontroller SLE 78, and the authors of [WGY20] implemented Saber on an ESP32 microcontroller used for IoT environments. The applicability of PQC for the automotive industry was demonstrated by the thesis author in [FVS19]. The work implemented NewHope on the automotive microcontroller AURIX. In [HPS+20], the same platform and algorithm were used to develop an authenticated key exchange for automotive systems. The suitability of the PQC NIST finalists for the automotive industry was investigated by the thesis author in [FVFS21]. Parts of this work are summarized in Section 7.3. The migration towards PQC is particularly critical for automotive applications as they are characterized by long life cycles and high safety as well as security requirements.

Hybrid key encapsulation. Compared to traditional cryptography, PQC has been less studied so far. Therefore, previous works proposed to combine the well-analyzed traditional KEMs with the new PQC approaches [Bra16, BBF+19]. This method is also known as key concatenation or hybrid key encapsulation. NIST already considered hybrid key encapsulation in the official recommendation for key-derivation methods [BBCD18]. However, while leading to a higher trust in the system's security, this approach also

implies a higher algorithmic and computational complexity. Section 7.4 discusses a hardware accelerator that natively supports traditional cryptography and PQC to increase the performance of hybrid key encapsulations.

Contributions. This chapter discusses the integration of PQC into real-world applications. The specific contributions can be summarized as:

- Development of a PQC chip for embedded and low energy applications;
- Integration of PQC PKE/KEM finalists into the popular AURIX microcontroller;
- Investigation of a hybrid key encapsulation accelerator for a secure transition towards PQC.

7.2 Post-Quantum Chip Design

This section presents the PQC chip developed in connection with this work. The design contains the RISC-V processor developed in Chapter 4 with the proposed tightly coupled PQC extensions for NewHope, Kyber, Saber, and SIKE.

7.2.1 ASIC Digital Design Flow

An Application-Specific Integrated Circuit (ASIC) is a customized microchip. ASICs allow efficiently realizing specific tasks, such as crypto acceleration or digital signal processing. The mask generation for the ASIC fabrication is extremely costly. But this setup cost becomes less relevant when a high volume of chips is produced. In order to decrease the design costs and complexity, predefined and reusable standard cells from a cell library can be used. For most designs, this is sufficient, and full-custom solutions are rather rare. For low production volumes or applications that demand a high hardware flexibility, reprogrammable FPGAs can be more suitable. FPGAs have configurable logic blocks that can be programmed and connected to realize the desired functionality. In contrast, ASICs can hardly be modified once the silicon is produced. However, current ASIC technologies can achieve significantly better performance and area results than FPGAs.

The ASIC design flow involves various steps, as illustrated in Figure 7.1. The logical synthesis, physical synthesis, and sign-off steps are the major phases of an ASIC digital design flow. Each design step involves Electronic Design Automation (EDA) tools helping designers to generate the GDSII file, which is handed over to the foundry for the production of the chip. The leading EDA tool providers are Cadence, Synopsys, and Mentor Graphics.

7.2.2 The Post-Quantum Chip

The developed ASIC is based on the design presented in Chapter 4. It includes the tightly coupled NTT and Modular Arithmetic Unit, the PQ-MAC accelerator, the sampling accelerator, the generic Keccak accelerator, and the large field arithmetic accelerator for

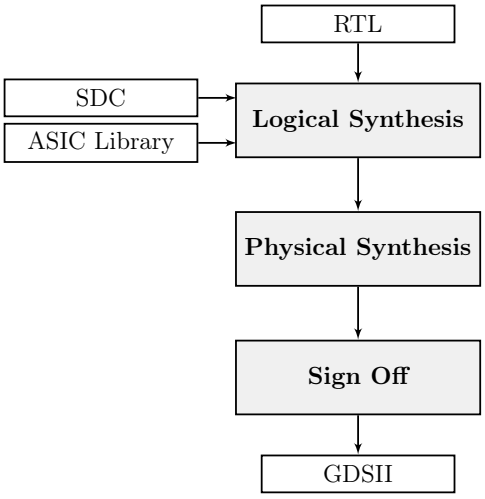


Figure 7.1: ASIC digital design flow.

SIKE (SIKEp751). The ASIC is designed for the UMC 65 nm technology with high threshold voltage cells. They trade a low power consumption with a slower signal propagation. The cell height of the standard cells is 1.8 μm, and the drawn gate length is 60 nm (physical gate length can slightly deviate).

Figure 7.2 (left) illustrates the ASIC layout after floorplanning and placement of standard cells. The overall chip size is 1 737.6μm $\times$ 1 737.6μm. The chip has in total 67 IO pads, visible at the edges in the picture. The IO pads are used for the 1.2 V core voltage, 3.3 V IO voltage, clock/reset logic, processor fetch enable, UART, SPI, I2C, GPIOs, and JTAG. The two large blocks in the upper left and lower left corners of the core area are memory blocks. They are used to store the instruction and data code for the RISC-V processor. The chip can be programmed via the SPI port. The standard cells that are part of the PQC accelerators are highlighted in the figure. The prime field arithmetic accelerator for the isogeny-based scheme SIKE is the largest part of the design. It uses several smaller multipliers, register stages, and adder circuits to realize the large field multiplications. Compared to the SIKE accelerator, the remaining accelerators are significantly smaller. The RISC-V core region is at the circular structure where the Keccak accelerator and parts of the other accelerators are placed. Other non-highlighted cells mainly belong to the peripherals.

Figure 7.2 (right) shows the final chip design of the ASIC. Vertical power stripes and a power ring around the core area of the ASIC were created to obtain a good power distribution across the chip. The highest routing congestion is at the RISC-V core region, where also the Keccak accelerator is placed. Numerous connections are the reason for the congestion at the processor core region. The high routing effort for Keccak is caused by several rewiring operations (e.g., Rho and Pi operations). Nevertheless, there is still space left for additional logic in the chip core region. The final chip is an IO limited design. This means that the size of the IO pads determines the overall chip size.

Figure 7.3 presents the post-quantum chip. The bare die is mounted on a QFN package

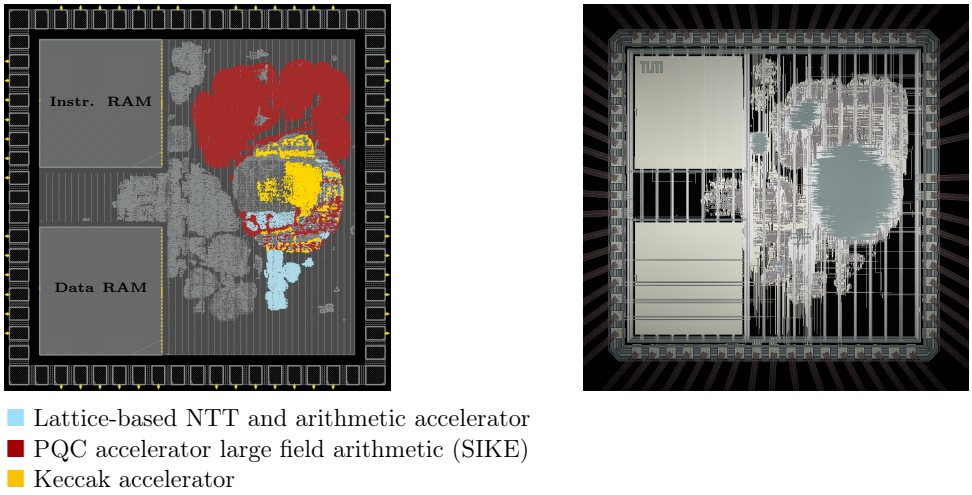


Figure 7.2: Placement of cells and highlighted PQC accelerators (left) and final chip (right).

Figure 7.3: Post-quantum chip. Open chip package (left) and chip in test setup (right).

with 80 pins and a size of 12 mm × 12 mm. This package allows connecting the chip to the Printed Circuit Board (PCB). Small wires, also called bonding wires, connect the passivation openings of the IO pads to the QFN package pins. The packaged chip is soldered on a PCB, or a socket is used for the mounting instead.

Measurement results and design characteristics. An overview of the post-quantum chip's cell count and area consumption is provided in Table 7.1. Further details about the chip size and routing are provided in Table 7.2. With a core density of 53.83 %, the design has a smaller overall cell area compared to the designs in Chapter 5, which uses the generic loosely coupled NTT accelerator. The reason is the use of separate memory macros for the loosely coupled NTT accelerator. Due to the SIKE accelerator, the amount of logic cells is larger when compared to the accelerated design in Chapter 5. The test board feeds the post-quantum ASIC with a 20 MHz clock. At typical operating

Table 7.1: RISQ-V area of ASIC tapeout (UMC 65 nm).

	Cell Count	Combinat.	Sequent.	Buffer+Inv.	Clk-Gate	Memory
Tapeout PQC Chip	104 111	268 033 μm^2 (186 kGE)	152 586 μm^2 (106 kGE)	57 156 μm^2 (40 kGE)	367 μm^2 (0.25 kGE)	669 345 μm^2 (465 kGE)

Table 7.2: RISQ-V size and routing details of ASIC tapeout (UMC 65 nm).

Routing Layers	#Nets	# Pads (w/o corner pads)	Area Pads	Core Density	Die size [μm^2]
8	118 328	67	585 654 μm^2	53.83%	1 737.6 × 1737.6

conditions, the ASIC achieves a maximum frequency of 45 MHz. For further performance and design evaluations, please refer to Chapter 4.

7.3 Application of PQC in the Automotive Industry

The automotive industry is currently going through one of its most significant transformations. The high degree of automation and the increasing effort for autonomous driving lead to unprecedented changes. One of the key aspects for the next decades is the secure interconnection of the various devices in and around the vehicle. This includes the connection to different vehicles (vehicle-to-vehicle communication) and nearby infrastructure (vehicle-to-infrastructure communication). These interconnections allow sharing important information about different events on the road, such as glazed frost, traffic jams, or accidents. Besides these safety-relevant data exchanges, less critical applications, like audio streaming (vehicle-to-everything communication), become increasingly important.

In the past, safety has always been of high relevance in the automotive industry, while security considerations were nearly not present. This changes with the increased attack surface due to the extreme interconnectivity. A high level of attention is required in the automotive industry when integrating cryptographic processes. Even small mistakes can have extreme impacts and cause serious accidents. Due to the long life cycles of automotive products, possible future attacks, including attacks with a quantum computer, are of high relevance.

The IEEE standard 1609.2-2016 "Standard for Wireless Access in Vehicular Environments - Security Services for Applications and Management Messages" [IEE16] is essential for security in the automotive industry. In order to realize secured communications, the standard recommends using the following cryptographic components:

- digital signatures using Elliptic Curve Digital Signature Algorithm (ECDSA);
- asymmetric encryption using Elliptic Curve Integrated Encryption Scheme (ECIES);
- symmetric authenticated encryption using Advanced Encryption Standard in CCM mode (AES-CCM).

The first two components will be completely broken when a large-scale quantum computer exists. In order to investigate the suitability of PQC for automotive systems, this section presents an evaluation of the PQC PKE/KEM finalists implemented on the automotive microcontroller AURIX.

AURIX. It is a widely used Harvard-based microcontroller family developed by Infineon for safety-critical real-time applications. AURIX microcontrollers are used in various automotive applications, such as secure onboard communication, powertrain applications (e.g., fuel injection), and safety applications (e.g., braking electronic control unit). They are developed to withstand demanding environmental conditions with high temperature ranges and steep temperature gradients. The lockstep architecture of AURIX with up to three independent TriCore CPUs allows achieving the highest Automotive Safety Integrity Level (ASIL) categorization for life-threatening applications. The deployed TriCore architecture of Infineon combines RISC, microcontroller, and DSP principles within one core to achieve a high performance for embedded applications.

For the evaluation of this section, the AURIX-TC297TF was used. The superscalar architecture of the core can achieve a low number of cycles per instruction. The architecture further supports MAC operations, SIMD operations, and compressed instructions. Directly at each of the cores, fast RAM blocks containing a Program Scratchpad RAM (PSPR) and a Data Scratchpad RAM (DSPR) are placed.

Performance evaluation. Table 7.3 summarizes the performance results of the three lattice-based NIST PKE/KEM finalists with different security levels. The implementations are based on the reference code submitted to NIST. For the measurements, the optimization flag `-O3` was set. Note that further assembly optimizations might lead to better performance results. The publications of the thesis author [FVS19, FVFS21] present further details about how such optimizations could be achieved for this target platform.

The results show that the cycle count reaches from 2 454 381 to 254 153 765 for the different algorithms and instantiations. At the maximum clock frequency of 300 MHz, these cycle counts lead to execution times between 8 ms and 847 ms. The highest parameter set of NTRU seems to be not directly suitable for a variety of applications. In particular, the key generation is slow for NTRU as it requires complex operations, such as polynomial inversions. Hardware acceleration might become necessary for some instantiations and schemes, although the AURIX microcontroller is already relatively powerful. Assembly optimizations can also increase the performance.

Open problems. The results have shown that some PQC algorithms are directly suitable for the integration into the AURIX microcontroller. The AURIX microcontroller has an HSM that is separated from the remaining system. It is based on a 32-bit processor and has several security features, such as secure key storage, TRNG, elliptic curve cryptography accelerator, and SHA256 accelerator. Extending the HSM to support PQC

Table 7.3: Cycle count and code size in bytes of lattice-based PKE/KEM finalists on AURIX-TC297TF.

Algorithm	Level	KeyGen	Encaps	Decaps	Total	Size
Kyber-512	I	684 696	846 402	923 283	2 454 381	18 924
Kyber-768	III	992 324	1 203 042	1 176 436	3 371 802	19 214
Kyber-1024	V	1 612 002	1 786 338	1 712 873	5 111 213	19 834
Lightsaber	I	889 026	1 162 042	1 336 895	3 387 963	19 044
Saber	III	1 736 518	2 168 152	2 449 162	6 353 832	18 588
Firesaber	V	2 867 689	3 450 745	3 847 442	10 165 876	18 908
ntruhps2048509	I	86 905 630	3 700 492	8 008 763	98 614 885	17 860
ntruhps2048677	III	153 245 662	6 084 278	14 065 691	173 395 631	18 032
ntruhps4096821	V	224 988 347	8 566 935	20 598 483	254 153 765	20 512

is still an open task. However, the described hardware architecture principles of Chapters 3–5 can be applied.

7.4 Hybrid Key Encapsulation

Previous works reused RSA and elliptic curve accelerators to accelerate lattice-based cryptography [AHH+18, WGY20, BRvV22]. The main idea of these works is to convert the polynomials of lattice-based cryptography into large integers in order to reuse the large field accelerators of traditional cryptography.

Kronecker substitution. The principle of the Kronecker substitution [Kro82] is to evaluate the polynomial $a(x)$ and $b(x)$ at a sufficiently large number $2^l \in \mathbb{Z}$ and perform the integer multiplication $c(2^l) = a(2^l) \cdot b(2^l)$. The result can then be transformed back to the polynomial representation $c(x)$. The substitution value must be large enough such that the final result has no overlaps in the coefficients.

This approach is well suited for applications where the accelerators are already developed. In that case, the Kronecker substitution overhead might be acceptable. However, new accelerators should switch the design focus from traditional cryptography to PQC.

7.4.1 Unified Post-Quantum and Elliptic Curve Accelerator

This section presents a hardware accelerator that natively supports the PQC NIST finalists NTRU and Saber and additionally elliptic curve cryptography. The developed hardware accelerator is thus ideally suited for hybrid key encapsulations. As a use case, the efficient PQC scheme Saber and the popular elliptic curve scheme Curve25519 [Ber06] are evaluated.

The idea of the unified hardware architecture is based on the shift register approach for polynomial multiplications of Sections 3.2.2 and 4.2.1. The architecture of these sections is extended to first support generic polynomial multiplications and then to support large integer multiplications. Some additional features complement the design.

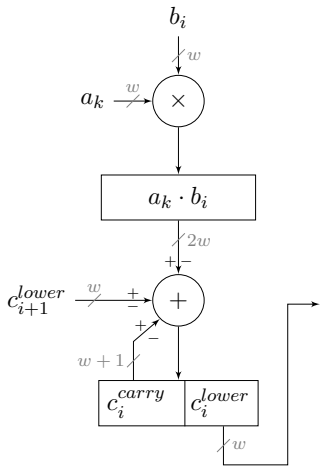


Figure 7.4: Generic MAC unit with integer multiplication support.

Extension for generic polynomial multiplications. The transformation of a ternary multiplier to a generic multiplier is straightforward for the shift-and-add multiplier architecture. Only the ternary modular arithmetic unit (see Section 3.2.2) must be replaced by a generic MAC unit. This unit multiplies the sequentially loaded coefficient a_k with the constant input coefficient b_i and accumulates the product in the shift register. The power-of-two modular reduction of each operation is implicitly performed by discarding the higher-order bits (limiting the register width). The resulting circuit is suitable for both NIST finalists Saber and NTRU.

Extension for integer multiplications. Each integer can be represented as a polynomial with w-bit coefficients. Then, the same shift-and-add multiplication method as for the polynomial multiplication can be used. But in contrast to the polynomial multiplication in Saber and NTRU, the result of the multiplications or accumulations is not reduced by the modulus. Therefore, the generic MAC unit must be extended to handle the carry propagation for the integer mode. Each MAC result has a bit-length of $2w+1$. The lower part, c_i^{low}, is shifted to the next generic MAC unit. The higher part, c_i^{carry}, is added to the content of the register in the next clock cycle. The resulting generic MAC unit is shown in Figure 7.4. The complete shift register circuit requires $n' = \lceil n/w \rceil$ generic MAC units and the same number of cycles to sequentially load the input chunks of an n-bit integer. Additional n' cycles are required to shift out the result.

Extensions for vectorized arithmetic. The multipliers and adders/subtractors of the generic MAC units are reused to perform vectorized polynomial multiplications, additions, and subtractions. Coefficient-wise operations of two vectors of length n' are supported when using all n' MAC units. This feature requires only some multiplexer logic to feed the correct inputs and read back the results from the multipliers or adders/subtractors.

Supporting reductions for Curve25519. Curve25519 requires field arithmetic in $\mathbb{F}_p$ with the pseudo-Mersenne prime $p = 2^{255} - 19$. Reductions can be simplified due to the fact $a = a^h \cdot 2^{255} + a^l \equiv a^h \cdot 19 + a^l \pmod{p}$. Hence, one multiplier of the generic MAC units is enhanced to support multiplications by 19.

Parameter selection. The shift register length is set to $n' = 16$ such that the overall circuit has a moderate area consumption. The larger polynomials in Saber and NTRU must apply a software-based split. The generic MAC unit maps well to the Xilinx FPGA DSP blocks DSP48E1/DSP48E2. DSP48E1 slices contain a 25×18-bit multiplier, a 48-bit adder/subtractor stage, a 17-bit shifter circuit, and a logic unit (supporting AND, NOT, NAND, NOR, XOR, XNOR operations). The more advanced DSP48E2 slice supports a 27×18-bit instead of a 25×18-bit multiplication and has a more flexible adder/subtractor stage. The generic MAC can be completely mapped to a DSP48E2 slice. The DSP48E1 slice has no feedback path from the output of the adder/subtractor stage to its input (for c_i^{carry}, Figure 7.4). Therefore, the carry logic must be instantiated outside of the DSP48E1 slice. DSP48E2 slices have such a feedback path. It can be used together with the 17-bit shifter circuit to forward c_i^{carry}. Given the operand widths of the DSP slices, the parameter $w = 17$ is set to reflect the supported bit-width of the shifting operation.

Core integration. The unified hardware architecture for post-quantum and elliptic curve cryptography is placed like the tightly coupled Keccak accelerator (see Section 4.3.5) into the decode stage. This provides fully parallel access to the registers in GPR/FPR. Due to the small shift register length n', no extra buffers are required.

7.4.2 Experimental Results

The next paragraphs provide the experimental results for the unified post-quantum and elliptic curve accelerator. As the side-channel evaluation is not the focus of this chapter, the measurements are again performed with the Xilinx Zynq-7000 programmable SoC (Zedboard). The Zynq-7000 family has DSP slices of type DSP48E1. The newer Ultrascale+ family has DSP48E2 slices. They are even more suitable for the mapping of the generic MAC unit as described in Section 7.4.1. For this reason, the measurements are repeated with the Zynq UltraScale+ ZCU102 FPGA.

Resource utilization evaluation. Table 7.4 provides an overview of the resource utilization. The results show that the tightly coupled design of Chapter 4 has a similar resource utilization as the approach of this chapter. While the designs in Chapters 4 and 5 also support NTT-based schemes, the architecture of this section supports the large integer arithmetic of elliptic curve cryptography and RSA. Using the Ultrascale+ FPGA slightly decreases the resource consumption. A larger difference between the two platforms can be observed at the maximum operating frequency, which is highly dependent on the technology node. The frequency of the baseline implementation and the design with unified hardware accelerator is 39/34 MHz (baseline/accelerated) for the Zynq-7000 platform and 83/76 MHz (baseline/accelerated) for the Ultrascale+ platform. Although

Table 7.4: Resource utilization results of unified post-quantum and elliptic curve design.

	FPGA type	LUT	FF	DSP	BRAM
PULPino original	Zynq-7000	15 248	9 569	6	32
Accel. Chapter 4	Zynq-7000	24 306	10 837	18	32
Accel. Chapter 5	Artix-7	20 697	11 833	13	36.5
Accel. Chapter 7	**Zynq-7000**	**24 235**	**11 863**	**22**	**32**
	UltraScale+	**23 441**	**11 094**	**22**	**32**

Table 7.5: Cycle count for Saber with unified post-quantum and elliptic curve accelerator.

	KeyGen	Encaps	Decaps
Saber on ESP32 with coprocessor [WGY20]	827 050	1 070 073	243 023[a)]
Saber Zynq-7000 SoC [BMTK+20]	2 180 000	2 762 000	2 560 000
Saber baseline (Chapter 4)	2 110 283	2 737 181	2 797 400
Saber (Chapter 4)	760 893	1 000 043	1 201 524
Saber (Chapter 5)	233 452	312 477	351 370
Saber (Chapter 7)	**217 330**	**279 320**	**300 480**

[a)] Only CPA-secure.

the critical path is not within the accelerators, the maximum operating frequency slightly decreased for the design of this chapter. This is most probably the case because a higher resource utilization leads to a higher routing effort. It must be noted that the Keccak and binomial sampling accelerators of Section 4.3 are included in the design costs of the proposed approach.

Cycle count evaluation for Saber. Table 7.5 provides the cycle count measurements of the proposed design. The results show a similar cycle count as the accelerated version in Chapter 5, which is based on the generic NTT with prime lift. This chapter uses a tailored ring splitting algorithm to map the large polynomials to the architecture. The developed approach allows performing the ring reductions directly in hardware. For further details, the corresponding publication of this section can be read [OFP+22].

Cycle count evaluation for Curve25519. The advantage of the generic design approach becomes present when evaluating the performance of elliptic curve cryptography. Table 7.6 summarizes the cycle count for different Curve25519 implementations. The RISC-V baseline implementation of this thesis uses the library of [Sod21]. Compared to the software baseline implementation, a speedup factor of 9.96 was achieved with the proposed accelerator. The achieved cycle count also outperforms existing ARM implementations. The results verify that the unified hardware multiplier significantly accelerates Saber and Curve25519.

Table 7.6: Cycle count for scalar multiplication in Curve25519 with unified post-quantum and elliptic curve accelerator.

	Scalar Mult.	Inv.	Mult.	Add.	Sub.
ARM Cortex-M4 [HL19][a)]	625 347	42 590	222	55	72
ARM Cortex-M4 [FA19][a)]	894 391	64 425	273	86	86
ARM Cortex-A7 [FA19]	825 914	62 648	290	52	52
ARM Cortex-A15 [FA19]	572 910	41 978	225	36	36
RISC-V Hifive1 [vdB20]	4 432 988	–	–	–	–
PULPino baseline	4 103 653	343 334	1 691	42	42
This work	**411 810**	**19 350**	**87**	**68**	**84**

[a)] The works evaluated the performance of different ARM Cortex-M4 microcontrollers. In the table, the results for the STM32F4 microcontroller are reported.

7.5 Summary

This chapter presented the first post-quantum chip that is based on a hardware/software codesign approach with post-quantum instruction set extensions. The chip verifies that the proposed tightly coupled post-quantum accelerators are suitable for real-world applications and ASIC designs. The complete chip has an area of only 3.01 mm^2. Due to its high flexibility, similar concepts are suitable for a variety of applications. The results show that no expensive or powerful processor is required to support PQC.

The chapter further investigated the suitability of PQC for automotive applications. The steadily growing communication effort, the required high level of safety and security, and the long life cycles make automotive cryptography a particularly relevant use case for applied cryptography. AURIX is a widely deployed microcontroller that fulfills the tough requirements of the automotive industry. In order to verify the applicability of PQC for this microcontroller, the NIST finalists Kyber, Saber, and NTRU were implemented and evaluated. While Kyber and Saber seem to be directly applicable for this microcontroller, NTRU seems to require more research effort to achieve good performance results for this microcontroller. Further research would be required to integrate PQC directly into the HSM of the AURIX microcontroller. For this purpose, the hardware/software codesign ideas of this thesis might be applied.

The last part of this chapter presented a hardware architecture that efficiently supports hybrid key encapsulations. Hybrid key encapsulations combine well-known cryptography with new PQC primitives. It fosters migration towards PQC as it increases the confidence in the security of a system. It turned out that the schoolbook multiplication based on a shift register approach suits well for hardware accelerators for the polynomial arithmetic of PQC and the large integer arithmetic of elliptic curve cryptography.

8 Conclusion

This chapter recapitulates the content and approaches presented in this thesis. The thesis investigated the applicability of PQC for embedded devices. Several hardware/software codesign strategies were developed that allow deploying PQC even on small non-commercial microcontrollers. This chapter summarizes these approaches and strategies. Furthermore, it highlights research areas that require more attention in the next years to develop highly efficient, practical, and secure cryptography resistant against quantum attacks.

8.1 Conclusion

In connection with the NIST standardization process, strong cryptographic algorithms have been developed that are resistant to quantum attacks. NIST plans to create the first draft standards of selected remaining candidates between the years 2022 and 2024 [NIS21]. Especially, lattice-based cryptography is well suited for the development of PKE/KEM and digital signature schemes. Most schemes of the NIST competition can be efficiently implemented on modern desktop computers. There is also remarkable progress in assembly optimized implementations for microcontrollers with moderate computing power. However, for countless applications with smaller computational power, hardware acceleration seems to be unavoidable. This is also the case for security applications on embedded devices that require strong protection against implementation attacks (e.g., HSMs or smartcards). Hardware acceleration additionally becomes important in a hybrid key encapsulation setting (PQC + traditional cryptography), where two cryptographic operations must be performed within the same time frame in which previously only one operation was performed.

Flexibility. Monolithic accelerators are tailored hardware realizations of a specific scheme or instance. They achieve an extremely high throughput but have the disadvantages of a large area consumption and low flexibility. For many applications with constrained resources, the extreme speedup of a monolithic accelerator is not necessary, e.g., because the performance is limited by the transmission rate anyway. For this reason, flexibility is often given greater importance. This thesis provided an investigation of hardware/software

codesign solutions for PQC to find an ideal trade-off between performance and flexibility.

Loosely coupled coprocessors. This work presented efficient coprocessors for the bottlenecks of lattice-based cryptography. The developed coprocessor for the ternary polynomial multiplication of NTRU achieves significant speed improvements. This work further explored suitable hardware realizations of the NTT, which is used for some lattice-based schemes to reduce the arithmetic complexity of polynomial multiplications. It was shown that efficient reduction routines could be integrated into NTT algorithms with on-the-fly Twiddle factor computations. Further, it was illustrated how to hide the NTT post-processing cost and that just a few additional gates significantly reduce the power consumption of the NTT circuit. The development of a hardware/software codesign for NewHope with loosely coupled Keccak and NTT coprocessors demonstrated that flexibility can be efficiently combined with performance. Except for hash and NTT computations, the whole algorithm flow was executed in software leading to a high degree of flexibility.

Tightly coupled accelerators. A tight processor coupling turned out to bring many advantages. It can reduce the area overhead as no costly input/output buffers and control circuitry are required. Moreover, existing system resources like memory blocks, registers, and multipliers of the processor can be reused. Tightly coupled accelerators are often designed for small operations, leading to a high flexibility. This thesis illustrated that tightly coupled accelerators and instruction set extensions are well suited for lattice-based cryptography, isogeny-based cryptography, and elliptic curve cryptography. In particular, Keccak and the NTT have proven to be well suited for a tightly coupled approach. In order to improve the performance, SIMD principles and memory access strategies were developed. A tightly coupled hardware/software codesign can be even faster than a loosely coupled approach. Loosely coupled solutions require a complex bus communication, which decreases the performance when copying multiple operands (e.g., polynomials) from the system memory to the accelerator. In order to avoid this overhead, intermediate results must be stored within the accelerator, leading to a higher area consumption of the accelerator. In summary, tightly coupled solutions should be preferred if they are able to meet the performance requirements and if the tasks can be broken down into sufficiently small pieces.

SCA protection. This work investigated DPA protection mechanisms for hardware/software codesigns of lattice-based cryptography. As a use case, the PQC NIST finalists Kyber and Saber were considered. The work presented masked accelerators and instruction set extensions for the linear and non-linear operations of these schemes. In particular, the non-linear operations, such as the Keccak Chi operation or binomial sampling, require great attention. They need to combine multiple shares and are thus often the source of side-channel leakage. Therefore, the non-linear operations were completely performed in hardware to achieve a controlled execution. The proposed approach carefully separates the different shares and deletes sensitive data before processing the next

share. Compared to Saber, Kyber turned out to have a higher masking overhead such that hardware acceleration is required for many embedded applications. Kyber requires sampling several binomially distributed polynomials, which is costly in a masked setting. Moreover, Kyber has a prime modulus, which complicates ciphertext compression and A2B/B2A conversions.

Hardware accelerator overview. The developed hardware accelerators of this work can be mainly categorized into sampling/hash accelerators, arithmetic accelerators, error correction accelerators, and A2B/B2A conversion accelerators. Table 8.1 provides an overview and analysis of accelerators for lattice-based and isogeny-based cryptography.

All sampling/hash accelerators are more suitable for a tight processor coupling. The Keccak and SHA256 accelerators can highly benefit from a parallel access to multiple registers of the processor. This parallel access brings a huge performance improvement and comes with a small cost of several multiplexers. In order to avoid a routing to the execution stage, Keccak and SHA256 accelerators can be directly placed in the instruction decode stage (T-ID coupling), where the register sets GPR and FPR are located. All other sampling accelerators, e.g., Binomial or Gaussian sampler, can be placed in the execution stage (T-EX coupling) as they only require access to a few registers. The Gaussian sampling has not been discussed so far. In [LG20], it was shown that uniformly distributed randomness could be efficiently turned into discrete Gaussian samples using Boolean functions. It turns out that these Boolean functions can be efficiently realized with a small hardware accelerator located in the execution stage. Further investigations have been conducted in a publication of the thesis author [KFS22].

Several accelerators were explored for the arithmetic operations of PQC. The most suitable accelerator depends on the application. If NTT-based schemes shall be supported, the developed tightly coupled NTT and Modular Arithmetic Unit is well suited. If, in addition, also non-NTT-based schemes shall be supported, a loose coupling can be considered (L coupling) because the operand width when using prime lifts might not be suitable for a 32-bit system. For schemes with a ternary polynomial multiplication, a ternary shift-register multiplier might be selected. This multiplier can be extremely fast but must apply a software splitting to achieve a good trade-off between area and performance. The generic shift-register multiplier is well suited for hybrid key encapsulations. It is, however, not suitable for NTT-based schemes. The large field arithmetic accelerator is suitable for SIKE and can be extended to support elliptic curve cryptography or RSA. SIKE is one of the slowest NIST schemes and requires a high amount of hardware resources or a small and fast technology node. Therefore, also standalone accelerator solutions might be considered for this scheme.

The thesis has shown how the error correction can improve the bandwidth and security level. On the other side, the error correction introduces another performance bottleneck. This bottleneck can be mitigated by an error correction accelerator.

Finally, the masked secure adder is relevant for all schemes that apply masking countermeasures and that require A2B/B2A conversions.

Table 8.1: List of accelerators.

Name	Category	Suited Schemes	Type	Notes
Keccak Accelerator	Sampling, hash	All	T-ID	Requires 50×32-bit reg. (reuse GPR/FPR)
Masked Keccak Chi Accelerator	Sampling, hash	All	T-EX	Can be combined with Keccak accelerator
SHA256 Accelerator	Sampling, hash	All	T-ID	Requires 8×32-bit reg. (reuse GPR)
Binomial Sampling Accelerator	Sampling	Kyber, Saber, NewHope	T-EX	–
Masked Binomial Sampling Accelerator	Sampling	Kyber, Saber, NewHope	T-EX	–
Gaussian Sampling Accelerator	Sampling	Frodo, Falcon	T-EX	–
NTT Accelerator I	Ring arithmetic	All NTT schemes	T-EX, T-ID	For modulus $\leq$ 32-bit
NTT Accelerator II	Ring arithmetic	All NTT schemes + Saber, NTRU, LAC	L	For non-NTT schemes with modulus $>$ 32-bit
Shift-Register-Multiplier I (ternary)	Ring arithmetic	NTRU, LAC	T-ID, L	Gets large if no splitting is applied
Shift-Register-Multiplier II (generic)	Ring arithmetic, field arithmetic	All non-NTT schemes, ECC, RSA	T-ID	Supports hybrid key encapsulation
Large field arithmetic accelerator	Field arithmetic	SIKE	T-ID	Can be extended for hybrid key encapsulation
Error Correction Accelerator	Error correction	LAC, ThreeBears	T-EX	For Chien search
Masked Secure Adder	A2B/B2A conversions	All	T-EX	Required in masked setting

NIST finalists from an implementation perspective. Due to its large public key size, Classic McEliece is not suitable for many embedded applications. For small devices, the most relevant PKE/KEM finalists are Kyber, NTRU, and Saber. They all have two main bottlenecks: polynomial sampling and ring arithmetic.

The polynomial sampling requires a huge amount of randomness. All three lattice-based finalists need an efficient PRNG, like Keccak, to create the required randomness. In a non-masked setting, the transformation of a uniform PRNG bitstream into the desired target distribution is relatively cheap. Only in a masked setting, high variances of the target distribution significantly increase the complexity. While Kyber needs to sample secret and error polynomials, the MLWR scheme Saber can completely avoid the sampling of error polynomials. This is a small advantage of Saber. NTRU is generally slower than the other two lattice-based finalists [AASA+20].

The polynomial ring arithmetic is the next important bottleneck. The selection of the modulus q has the largest impact on the performance. A modulus that supports the NTT leads to efficient implementations. Even if the modulus does not directly support NTTs, prime lifts can be considered to apply NTT-based polynomial multiplications. The prime lift, however, leads to increased operand widths. While Kyber is highly optimized for the NTT, Saber and NTRU have a power-of-two modulus and can apply the NTT only with prime lifts. On the other side, the power-of-two modulus facilitates the integration of masking. In addition to the costly polynomial multiplications, NTRU requires time-consuming polynomial inversions within the key generation, which is not required for Kyber and Saber.

All three lattice-based finalists have their advantages and disadvantages. The differences between Kyber and Saber are relatively small. Both schemes can be efficiently implemented. NTRU seems to require more research effort if it gets standardized by NIST. Particularly, the protection against SCA and the efficient implementation of the key generation have received less research attention so far.

8.2 Future Work

In the last years, the focus has been on PKE/KEM schemes—among others—because encrypted messages can be recorded and broken later when a sufficiently large quantum computer is built. Even if such scenarios are less problematic for signature schemes, they should soon receive more research attention due to long migration times. Lattice-based signature schemes require similar hardware accelerators as PKE/KEM schemes. Nevertheless, a thorough analysis of different implementation aspects is missing.

Efficient countermeasures against implementation attacks are also not sufficiently studied yet. Lattice-based cryptography is built upon many different components. This leads to a complicated overall system with various potential vulnerabilities. A comprehensive guideline for the protection against implementation attacks must be developed. In particular, efficient countermeasures against horizontal SCA and fault attacks have not yet been sufficiently investigated. The attacks are becoming more sophisticated every day. Among others, attacks using machine learning have become very powerful. Research

in the direction of secure and efficient PQC implementations but also the migration of various applications towards PQC will take many years to complete.

Acronyms

`BPGM` Blinding Polynomial Generation Method.

`MGF` Mask Generation Function.

A2B Arithmetic to Boolean.

ACP Accelerator Coherency Port.

AES Advanced Encryption Standard.

ALU Arithmetic Logic Unit.

ASIC Application-Specific Integrated Circuit.

ASIL Automotive Safety Integrity Level.

AURIX Automotive Realtime Integrated NeXt Generation Architecture.

AXI Advanced eXtensible Interface.

B2A Boolean to Arithmetic.

BCH Bose–Chaudhuri–Hocquenghem.

BI-AWGNC Binary Input Additive White Gaussian Noise Channel.

BRAM Block RAM.

CCA Chosen-Ciphertext Attacks.

CPA Chosen-Plaintext Attacks.

CRT Chinese Remainder Theorem.

CSR Control and Status Register.

DDR-RAM Double Data Rate Synchronous Dynamic Random-Access Memory.

DIF Decimation-In-Frequency.

DIT Decimation-In-Time.

DMA Direct Memory Access.

DOM Domain-Oriented Masking.

DPA Differential Power Analysis.

DSP Digital Signal Processing.

EDA Electronic Design Automation.

FF Flip-Flop.

FFT Fast Fourier Transform.

FPGA Field-Programmable Gate Array.

FPR Floating-Point Register.

FPU Floating-Point Unit.

GP General-Purpose Port.

GPIO General-Purpose Input/Output.

GPR General-Purpose Register.

HP High Performance Port.

HSM Hardware Security Module.

I2C Inter-Integrated Circuit.

IMLWE Integer Module Learning With Errors.

IoT Internet of Things.

ISA Instruction Set Architecture.

KEM Key Encapsulation Mechanism.

LDPC Low-Density Parity-Check.

LFSR Linear-Feedback Shift Register.

LLR Log-Likelihood Ratio.

LSU Load-Store Unit.

LUT Lookup Table.

LWE Learning With Errors.

LWR Learning With Rounding.

MAC Multiply Accumulate.

MAU Modular Arithmetic Unit.

MLWE Module Learning With Errors.

MLWR Module Learning With Rounding.

NAEP NTRU Asymmetric Encryption Padding.

NIST National Institute of Standards and Technology.

NSA National Security Agency.

NTRU Nth Degree Truncated Polynomial Ring Unit.

NTT Number Theoretic Transform.

PCB Printed Circuit Board.

PKC Public-Key Cryptography.

PKE Public-Key Encryption.

PQC Post-Quantum Cryptography.

PRNG Pseudo Random Number Generator.

PULP Parallel Ultra Low Power.

QC-LDPC Quasi-Cyclic Low-Density Parity-Check.

QC-MDPC Quasi-Cyclic Moderate-Density Parity-Check.

QFN Quad-Flat No-leads.

RAM Random Access Memory.

RCC Rocket Custom Coprocessor.

RISC Reduced Instruction Set Computer.

RLWE Ring Learning With Errors.

RNG Random Number Generator.

RSA Rivest–Shamir–Adleman.

SCA Side-Channel Attacks.

SIMD Single Instruction Multiple Data.

SoC System on a Chip.

SPA Simple Power Analysis.

SPI Serial Peripheral Interface.

SVES Shortest Vector Encryption Scheme.

TCF Toggle Count Format.

TI Threshold Implementation.

TPM Trusted Platform Module.

TRNG True Random Number Generator.

TVLA Test Vector Leakage Assessment.

UART Universal Asynchronous Receiver Transmitter.

XOF eXtendable-Output Function.

Own Publications

[BFM+18] Konstantin Braun, Tim Fritzmann, Georg Maringer, Thomas Schamberger, and Johanna Sepúlveda. Secure and compact full NTRU hardware implementation. In *IFIP/IEEE International Conference on Very Large Scale Integration (VLSI-SoC)*, pages 89–94. IEEE, 2018.

[FPS18] Tim Fritzmann, Thomas Pöppelmann, and Johanna Sepúlveda. Analysis of error-correcting codes for lattice-based key exchange. In *Selected Areas in Cryptography (SAC)*, pages 369–390. Springer, 2018.

[Fri17] Tim Fritzmann. Towards an improved NewHope Simple. Master's thesis, Technical University of Munich, 2017. Submitted on: 25th Oct. 2017.

[FS19] Tim Fritzmann and Johanna Sepúlveda. Efficient and flexible low-power NTT for lattice-based cryptography. In *IEEE International Symposium on Hardware Oriented Security and Trust (HOST)*, pages 141–150. IEEE, 2019.

[FSF+19] Tim Fritzmann, Thomas Schamberger, Christoph Frisch, Konstantin Braun, Georg Maringer, and Johanna Sepúlveda. Efficient hardware/software co-design for NTRU. In Nicola Bombieri, Graziano Pravadelli, Masahiro Fujita, Todd Austin, and Ricardo Reis, editors, *VLSI-SoC: Design and Engineering of Electronics Systems Based on New Computing Paradigms*, pages 257–280. Springer, 2019.

[FSM+19] Tim Fritzmann, Uzair Sharif, Daniel Müller-Gritschneder, Cezar Reinbrecht, Ulf Schlichtmann, and Johanna Sepúlveda. Towards reliable and secure post-quantum co-processors based on RISC-V. In *Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pages 1148–1153. IEEE, 2019.

[FSS20a] Tim Fritzmann, Georg Sigl, and Johanna Sepúlveda. Extending the RISC-V instruction set for hardware acceleration of the post-quantum scheme LAC. In *Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pages 1420–1425. IEEE, 2020.

[FSS20b] Tim Fritzmann, Georg Sigl, and Johanna Sepúlveda. RISQ-V: Tightly coupled RISC-V accelerators for post-quantum cryptography. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, 2020(4):239–280, Aug. 2020.

[FVBBR+21] Tim Fritzmann, Michiel Van Beirendonck, Debapriya Basu Roy, Patrick Karl, Thomas Schamberger, Ingrid Verbauwhede, and Georg Sigl. Masked accelerators and instruction set extensions for post-quantum cryptography. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, 2022(1):414–460, Nov. 2021.

[FVFS21] Tim Fritzmann, Jonas Vith, Daniel Flórez, and Johanna Sepúlveda. Post-quantum cryptography for automotive systems. *Microprocessors and Microsystems*, 87(November 2021):1–8, Nov. 2021.

[FVS19] Tim Fritzmann, Jonas Vith, and Johanna Sepúlveda. Post-quantum key exchange mechanism for safety critical systems. In *17th escar Europe: embedded security in cars*. Ruhr-Universität Bochum, Universitätsbibliothek, 2019.

[FVS20] Tim Fritzmann, Jonas Vith, and Johanna Sepúlveda. Strengthening post-quantum security for automotive systems. In *23rd Euromicro Conference on Digital System Design (DSD)*, pages 570–576. IEEE, 2020.

[KFS22] Patrick Karl, Tim Fritzmann, and Georg Sigl. Hardware accelerated FrodoKEM on RISC-V. In *International Symposium on Design and Diagnostics of Electronic Circuits and Systems (DDECS)*. IEEE, 2022.

[MFS20] Georg Maringer, Tim Fritzmann, and Johanna Sepúlveda. The influence of LWE/RLWE parameters on the stochastic dependence of decryption failures. In *International Conference on Information and Communications Security (ICICS)*, pages 331–349. Springer, 2020.

[OFP+22] Felix Oberhansl, Tim Fritzmann, Thomas Pöppelmann, Debapriya Basu Roy, and Georg Sigl. Uniform instruction set extensions for multiplications in contemporary and post-quantum cryptography. In *Journal of Cryptographic Engineering (JCEN)*. Springer, 2022. Submitted in February 2022 (no acceptance decision yet).

[RFS20] Debapriya Basu Roy, Tim Fritzmann, and Georg Sigl. Efficient hardware/software co-design for post-quantum crypto algorithm SIKE on ARM and RISC-V based microcontrollers. In *IEEE/ACM International Conference On Computer Aided Design (ICCAD)*, pages 1–9. IEEE/ACM, 2020.

Bibliography

[AAB+19a] Erdem Alkim, Roberto Avanzi, Joppe Bos, Léo Ducas, Antonio de la Piedra, Thomas Pöppelmann, Peter Schwabe, and Douglas Stebila. NewHope: Algorithm Specifications and Supporting Documentation (v2), 2019. Accessed December 18, 2021: https://newhopecrypto.org/data/NewHope_2019_07_10.pdf.

[AAB+19b] Frank Arute, Kunal Arya, Ryan Babbush, Dave Bacon, Joseph C Bardin, Rami Barends, Rupak Biswas, Sergio Boixo, Fernando GSL Brandao, David A Buell, et al. Quantum supremacy using a programmable superconducting processor. *Nature*, 574(7779):505–510, 2019.

[AASA+19] Gorjan Alagic, Jacob Alperin-Sheriff, Daniel Apon, David Cooper, Quynh Dang, Yi-Kai Liu, Carl Miller, Dustin Moody, Rene Peralta, et al. Status report on the first round of the NIST post-quantum cryptography standardization process. *US Department of Commerce, NIST*, 2019.

[AASA+20] Gorjan Alagic, Jacob Alperin-Sheriff, Daniel Apon, David Cooper, Quynh Dang, John Kelsey, Yi-Kai Liu, Carl Miller, Dustin Moody, Rene Peralta, et al. Status report on the second round of the NIST post-quantum cryptography standardization process. *US Department of Commerce, NIST*, 2020.

[ABB+20] Nicolas Aragon, Paulo Barreto, Slim Bettaieb, Loic Bidoux, Olivier Blazy, Jean-Christophe Deneuville, Phillipe Gaborit, Santosh Ghosh, Shay Gueron, Tim Guneysu, Carlos Aguilar Melchor, et al. BIKE: Bit flipping key encapsulation (Round 3 submission), 2020. Accessed December 18, 2021: https://csrc.nist.gov/projects/post-quantum-cryptography/round-3-submissions.

[ABCG20] Erdem Alkim, Yusuf Alper Bilgin, Murat Cenk, and François Gérard. Cortex-M4 optimizations for {R,M}LWE schemes. Cryptology ePrint Archive, Report 2020/012, 2020.

[ABD+19] Roberto Avanzi, Joppe Bos, Léo Ducas, Eike Kiltz, Tancrède Lepoint, Vadim Lyubashevsky, John M. Schanck, Peter Schwabe, Gregor Seiler, and Damien Stehlé. CRYSTALS-Kyber: Algorithm specifications and supporting documentation (v2), 2019. Accessed December 18, 2021: https://pq-crystals.org/kyber/data/kyber-specification-round2.pdf.

[ABD+20] Roberto Avanzi, Joppe Bos, Léo Ducas, Eike Kiltz, Tancrède Lepoint, Vadim Lyubashevsky, John M. Schanck, Peter Schwabe, Gregor Seiler, and Damien Stehlé. CRYSTALS-Kyber: Algorithm specifications and supporting documentation (v3), 2020. Accessed December 18, 2021: https://pq-crystals.org/kyber/data/kyber-specification-round3.pdf.

[ABF+08] Ali Can Atici, Lejla Batina, Junfeng Fan, Ingrid Verbauwhede, and S Berna Ors Yalcin. Low-cost implementations of NTRU for pervasive security. In *International Conference on Application-Specific Systems, Architectures and Processors*, pages 79–84. IEEE, 2008.

[ABP+18] Victor Arribas, Begül Bilgin, George Petrides, Svetla Nikova, and Vincent Rijmen. Rhythmic Keccak: SCA security and low latency in HW. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, 2018(1):269–290, Feb. 2018.

[ADPS16a] Erdem Alkim, Léo Ducas, Thomas Pöppelmann, and Peter Schwabe. NewHope without reconciliation. Cryptology ePrint Archive, Report 2016/1157, 2016.

[ADPS16b] Erdem Alkim, Léo Ducas, Thomas Pöppelmann, and Peter Schwabe. Post-quantum key exchange—a new hope. In *USENIX security symposium*, pages 327–343, 2016.

[AEL+20] Erdem Alkim, Hülya Evkan, Norman Lahr, Ruben Niederhagen, and Richard Petri. ISA extensions for finite field arithmetic: Accelerating Kyber and NewHope on RISC-V. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, 2020(3):219–242, Jun. 2020.

[AHH+18] Martin R. Albrecht, Christian Hanser, Andrea Hoeller, Thomas Pöppelmann, Fernando Virdia, and Andreas Wallner. Implementing RLWE-based schemes using an RSA co-processor. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, 2019(1):169–208, Nov. 2018.

[AJS16] Erdem Alkim, Philipp Jakubeit, and Peter Schwabe. NewHope on ARM Cortex-M. In *International Conference on Security, Privacy, and Applied Cryptography Engineering*, pages 332–349. Springer, 2016.

[Ajt96] Miklós Ajtai. Generating hard instances of lattice problems. In *Proceedings of the 28th annual ACM Symposium on Theory of Computing*, pages 99–108. Association for Computing Machinery, 1996.

[APS13] Aydin Aysu, Cameron Patterson, and Patrick Schaumont. Low-cost and area-efficient FPGA implementations of lattice-based cryptography. In *IEEE International Symposium on Hardware-Oriented Security and Trust (HOST)*, pages 81–86. IEEE, 2013.

[Bar86] Paul Barrett. Implementing the Rivest Shamir and Adleman public key encryption algorithm on a standard digital signal processor. In *Conference on the Theory and Application of Cryptographic Techniques*, pages 311–323. Springer, 1986.

[BBC+18] Marco Baldi, Alessandro Barenghi, Franco Chiaraluce, Gerardo Pelosi, and Paolo Santini. LEDAkem: A post-quantum key encapsulation mechanism based on QC-LDPC codes. In *International Conference on Post-Quantum Cryptography*, pages 3–24. Springer, 2018.

[BBCD18] Elaine Barker, Elaine Barker, Lily Chen, and Richard Davis. *Recommendation for key-derivation methods in key-establishment schemes*. US Department of Commerce, National Institute of Standards and Technology, 2018.

[BBE+18] Gilles Barthe, Sonia Belaïd, Thomas Espitau, Pierre-Alain Fouque, Benjamin Grégoire, Mélissa Rossi, and Mehdi Tibouchi. Masking the GLP lattice-based signature scheme at any order. In *Annual International Conference on the Theory and Applications of Cryptographic Techniques*, pages 354–384. Springer, 2018.

[BBF+19] Nina Bindel, Jacqueline Brendel, Marc Fischlin, Brian Goncalves, and Douglas Stebila. Hybrid key encapsulation mechanisms and authenticated key exchange. In *International Conference on Post-Quantum Cryptography*, pages 206–226. Springer, 2019.

[BCE+01] Daniel V. Bailey, Daniel Coffin, Adam Elbirt, Joseph H. Silverman, and Adam D. Woodbury. NTRU in constrained devices. In *International Workshop on Cryptographic Hardware and Embedded Systems*, pages 262–272. Springer, 2001.

[BDGH15] Shivam Bhasin, Jean-Luc Danger, Sylvain Guilley, and Wei He. Exploiting FPGA block memories for protected cryptographic implementations. *ACM Transactions on Reconfigurable Technology and Systems (TRETS)*, 8(3):1–16, 2015.

[BDH+20] Guido Bertoni, Joan Daemen, Seth Hoffert, Michaël Peeters, and Gilles Van Assche. Keccak in VHDL, 2020. Accessed December 18, 2021: https://keccak.team/hardware.html.

[BDK+18] Joppe Bos, Léo Ducas, Eike Kiltz, Tancrède Lepoint, Vadim Lyubashevsky, John M. Schanck, Peter Schwabe, Gregor Seiler, and Damien Stehlé. CRYSTALS-Kyber: a CCA-secure module-lattice-based KEM. In *2018 IEEE European Symposium on Security and Privacy (EuroS&P)*, pages 353–367. IEEE, 2018.

[BDK+21] Michiel Van Beirendonck, Jan-Pieter D'Anvers, Angshuman Karmakar, Josep Balasch, and Ingrid Verbauwhede. A side-channel-resistant implementation of Saber. *Journal on Emerging Technologies in Computing Systems*, 17(2), Apr. 2021.

[BDPVA09] Guido Bertoni, Joan Daemen, Michaël Peeters, and Gilles Van Assche. Keccak sponge function family main document. *Submission to NIST (Round 2)*, 3(30):320–337, 2009.

[BDPVA10] Guido Bertoni, Joan Daemen, Michaël Peeters, and Gilles Van Assche. Building power analysis resistant implementations of Keccak. In *Second SHA-3 Candidate Conference*. Citeseer, 2010.

[Ber66] Elwyn R. Berlekamp. Nonbinary BCH decoding. In *International Symposium on Information Theory*, 1966.

[Ber06] Daniel J. Bernstein. Curve25519: New Diffie-Hellman speed records. In *Public Key Cryptography*, pages 207–228. Springer, 2006.

[BGR+21] Joppe W. Bos, Marc Gourjon, Joost Renes, Tobias Schneider, and Christine van Vredendaal. Masking Kyber: First- and higher-order implementations. Cryptology ePrint Archive, Report 2021/483, 2021.

[BGV93] Antoon Bosselaers, René Govaerts, and Joos Vandewalle. Comparison of three modular reduction functions. In *Annual International Cryptology Conference*, pages 175–186. Springer, 1993.

[BGV14] Zvika Brakerski, Craig Gentry, and Vinod Vaikuntanathan. (Leveled) fully homomorphic encryption without bootstrapping. *ACM Transactions on Computation Theory (TOCT)*, 6(3):1–36, 2014.

[BKS19] Leon Botros, Matthias J. Kannwischer, and Peter Schwabe. Memory-efficient high-speed implementation of Kyber on Cortex-M4. In *International Conference on Cryptology in Africa*, pages 209–228. Springer, 2019.

[BL21] Daniel J. Bernstein and Tanja Lange. eBACS: ECRYPT benchmarking of cryptographic systems, 2021. Accessed December 18, 2021: https://bench.cr.yp.to.

[BMD+20] Andrea Basso, Jose Maria Bermudo Mera, Jan-Pieter D'Anvers, Angshuman Karmakar, Sujoy Sinha Roy, Michiel Van Beirendonck, and Frederik Vercauteren. SABER: Mod-LWR based KEM (v3), 2020. Accessed December 18, 2021: https://csrc.nist.gov/projects/post-quantum-cryptography/round-3-submissions.

[BMKV20] Jose Maria Bermudo Mera, Angshuman Karmakar, and Ingrid Verbauwhede. Time-memory trade-off in Toom-Cook multiplication: an application to module-lattice based cryptography. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, 2020(2):222–244, Mar. 2020.

[BMTK+20] Jose Maria Bermudo Mera, Furkan Turan, Angshuman Karmakar, Sujoy Sinha Roy, and Ingrid Verbauwhede. Compact domain-specific co-processor for accelerating module lattice-based KEM. In *57th ACM/IEEE Design Automation Conference (DAC)*, pages 1–6. ACM/IEEE, 2020.

[BMVT78] Elwyn Berlekamp, Robert McEliece, and Henk Van Tilborg. On the inherent intractability of certain coding problems. *IEEE Transactions on Information Theory*, 24(3):384–386, 1978.

[BNN+12] Begül Bilgin, Svetla Nikova, Ventzislav Nikov, Vincent Rijmen, and Georg Stütz. Threshold implementations of all 3×3 and 4×4 S-boxes. In *International Workshop on Cryptographic Hardware and Embedded Systems*, pages 76–91. Springer, 2012.

[BPR12] Abhishek Banerjee, Chris Peikert, and Alon Rosen. Pseudorandom functions and lattices. In *Annual International Conference on the Theory and Applications of Cryptographic Techniques*, pages 719–737. Springer, 2012.

[BR21] Andrea Basso and Sujoy Sinha Roy. Optimized polynomial multiplier architectures for post-quantum KEM Saber. In *58th ACM/IEEE Design Automation Conference (DAC)*, pages 1285–1290. ACM/IEEE, 2021.

[Bra16] Matt Braithwaite. Experimenting with post-quantum cryptography, 2016. Accessed December 18, 2021: https://security.googleblog.com/2016/07/experimenting-with-post-quantum.html.

[BRvV22] Joppe W. Bos, Joost Renes, and Christine van Vredendaal. Post-quantum cryptography with contemporary co-processors: Beyond Kronecker, Schönhage-Strassen & Nussbaumer. In *31st USENIX Security Symposium*. USENIX Association, 2022.

[BUC19] Utsav Banerjee, Tenzin Ukyab, and Anantha Chandrakasan. Sapphire: A configurable crypto-processor for post-quantum lattice-based protocols. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, 2019(4):17–61, Aug. 2019.

[CFG+10] Christophe Clavier, Benoit Feix, Georges Gagnerot, Mylène Roussellet, and Vincent Verneuil. Horizontal correlation analysis on exponentiation. In *International Conference on Information and Communications Security (ICICS)*, pages 46–61. Springer, 2010.

[CGV14] Jean-Sébastien Coron, Johann Großschädl, and Praveen Kumar Vadnala. Secure conversion between Boolean and arithmetic masking of any order. In *International Workshop on Cryptographic Hardware and Embedded Systems*, pages 188–205. Springer, 2014.

[Chi64] Robert Chien. Cyclic decoding procedures for Bose-Chaudhuri-Hocquenghem codes. *IEEE Transactions on Information Theory*, 10(4):357–363, 1964.

[CHK+21] Chi-Ming Marvin Chung, Vincent Hwang, Matthias J. Kannwischer, Gregor Seiler, Cheng-Jhih Shih, and Bo-Yin 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.

[CJL+16] Lily Chen, Stephen Jordan, Yi-Kai Liu, Dustin Moody, Rene Peralta, Ray Perlner, and Daniel Smith-Tone. *Report on post-quantum cryptography*, volume 12. US Department of Commerce, National Institute of Standards and Technology, 2016.

[CJRR99] Suresh Chari, Charanjit S. Jutla, Josyula R. Rao, and Pankaj Rohatgi. Towards sound approaches to counteract power-analysis attacks. In *Annual International Cryptology Conference*, pages 398–412. Springer, 1999.

[CT65] James W. Cooley and John W. Tukey. An algorithm for the machine calculation of complex Fourier series. *Mathematics of Computation*, 19(90):297–301, 1965.

[CT03] Jean-Sébastien Coron and Alexei Tchulkine. A new algorithm for switching from arithmetic to Boolean masking. In *International Workshop on Cryptographic Hardware and Embedded Systems*, pages 89–97. Springer, 2003.

[Deb12] Blandine Debraize. Efficient and provably secure methods for switching from arithmetic to Boolean masking. In *International Workshop on Cryptographic Hardware and Embedded Systems*, pages 107–121. Springer, 2012.

[DI 10] DI Management Services. The Chinese Remainder Theorem, 2010. Accessed December 18, 2021: https://www.di-mgt.com.au/crt.html.

[DKRV18] Jan-Pieter D'Anvers, Angshuman Karmakar, Sujoy Sinha Roy, and Frederik Vercauteren. Saber: Module-LWR based key exchange, CPA-secure encryption and CCA-secure KEM. In *International Conference on Cryptology in Africa*, pages 282–305. Springer, 2018.

[DKRV20] Jan-Pieter D'Anvers, Angshuman Karmakar, Sujoy Sinha Roy, and Frederik Vercauteren. SABER: Mod-LWR based KEM (v2), 2020. Accessed December 18, 2021: https://csrc.nist.gov/projects/post-quantum-cryptography/round-2-submissions.

[DTVV19] Jan-Pieter D'Anvers, Marcel Tiepelt, Frederik Vercauteren, and Ingrid Verbauwhede. Timing attacks on error correcting codes in post-quantum schemes. In *Proceedings of ACM Workshop on Theory of Implementation Security Workshop*, pages 2–9. Association for Computing Machinery, 2019.

[DVV19] Jan-Pieter D'Anvers, Frederik Vercauteren, and Ingrid Verbauwhede. The impact of error dependencies on Ring/Mod-LWE/LWR based schemes. In *Post-Quantum Cryptography*, pages 103–115. Springer, 2019.

[DY09] Jintai Ding and Bo-Yin Yang. Multivariate public key cryptography. In *Post-Quantum Cryptography*, pages 193–241. Springer, 2009.

[FA19] Hayato Fujii and Diego F. Aranha. Curve25519 for the Cortex-M4 and beyond. In *International Conference on Cryptology and Information Security in Latin America (LATINCRYPT)*, pages 109–127. Springer, 2019.

[Fan12] John Fan. *Constrained Coding and Soft Iterative Decoding*. The Springer International Series in Engineering and Computer Science. Springer, 2012.

[FDNG19] Farnoud Farahmand, Viet B. Dang, Duc Tri Nguyen, and Kris Gaj. Evaluating the potential for hardware acceleration of four NTRU-based key encapsulation mechanisms using software/hardware codesign. In *PQCrypto*, pages 23–43. Springer, 2019.

[Flu16] Scott Fluhrer. Cryptanalysis of ring-LWE based key exchange with key share reuse. Cryptology ePrint Archive, Report 2016/085, 2016.

[FND+19] Farnoud Farahmand, Duc Tri Nguyen, Viet B. Dang, Ahmed Ferozpuri, and Kris Gaj. Software/hardware codesign of the post quantum cryptography algorithm NTRUEncrypt using high-level synthesis and register-transfer level design methodologies. In *29th International Conference on Field Programmable Logic and Applications (FPL)*, 2019.

[FO99] Eiichiro Fujisaki and Tatsuaki Okamoto. Secure integration of asymmetric and symmetric encryption schemes. In *Annual International Cryptology Conference*, pages 537–554. Springer, 1999.

[Gal62] Robert Gallager. Low-density parity-check codes. *IRE Transactions on Information Theory*, 8(1):21–28, 1962.

[Gam20] Jay Gambetta. IBM's roadmap for scaling quantum technology, 2020. Accessed December 18, 2021: https://research.ibm.com/blog/ibm-quantum-roadmap.

[GE21] Craig Gidney and Martin Ekerå. How to factor 2048 bit RSA integers in 8 hours using 20 million noisy qubits. *Quantum*, 5:433, 2021.

[GFS+12] Norman Göttert, Thomas Feller, Michael Schneider, Johannes Buchmann, and Sorin Huss. On the design of hardware building blocks for modern lattice-based encryption schemes. In *International Workshop on Cryptographic Hardware and Embedded Systems*, pages 512–529. Springer, 2012.

[GHW12] Richard Gitlin, Jeremiah Hayes, and Stephen Weinstein. *Data Communications Principles*. Applications of Communications Theory. Springer, 2012.

[GJJR11] Gilbert Goodwill, Benjamin Jun, Josh Jaffe, and Pankaj Rohatgi. A testing methodology for side-channel resistance validation. In *NIST non-invasive attack testing workshop*, pages 115–136, 2011.

[Glo11] Neal Glover. Fast software BCH encoder and decoder, 2011. Accessed December 18, 2021: http://www.channelscience.com/files/NealGloverFastSoftwareBCHEndecR400r2.pdf.

[GMK16] Hannes Gross, Stefan Mangard, and Thomas Korak. Domain-Oriented Masking: Compact masked hardware implementations with arbitrary protection order. In *ACM Workshop on Theory of Implementation Security*, page 3. Association for Computing Machinery, 2016.

[GOPS13] Tim Güneysu, Tobias Oder, Thomas Pöppelmann, and Peter Schwabe. Software speed records for lattice-based signatures. In *International Workshop on Post-Quantum Cryptography*, pages 67–82. Springer, 2013.

[Gou01] Louis Goubin. A sound method for switching between Boolean and arithmetic masking. In *International Workshop on Cryptographic Hardware and Embedded Systems*, pages 3–15. Springer, 2001.

[GPM+17] Oscar M. Guillen, Thomas Pöppelmann, Jose M. Bermudo Mera, Elena Fuentes Bongenaar, Georg Sigl, and Johanna Sepúlveda. Towards post-quantum security for IoT endpoints with NTRU. In *Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pages 698–703. IEEE, 2017.

[GR19] François Gérard and Mélissa Rossi. An efficient and provable masked implementation of qTESLA. In *International Conference on Smart Card Research and Advanced Applications*, pages 74–91. Springer, 2019.

[Gre20] Denisa Greconici. Kyber on RISC-V, 2020. Accessed December 18, 2021: https://www.ru.nl/publish/pages/769526/denisa_greconici.pdf.

[Gro96] Lov K. Grover. A fast quantum mechanical algorithm for database search. In *28th annual ACM Symposium on Theory of Computing*, pages 212–219. Association for Computing Machinery, 1996.

[GS66] W. Morven Gentleman and Gordon Sande. Fast Fourier transforms: for fun and profit. In *Proceedings of the November 7-10, 1966, Fall Joint Computer Conference*, pages 563–578, 1966.

[GSM17] Hannes Groß, David Schaffenrath, and Stefan Mangard. Higher-order side-channel protected implementations of Keccak. In *Euromicro Conference on Digital System Design (DSD)*, pages 205–212. IEEE, 2017.

[Gu19] Chunsheng Gu. Integer version of ring-LWE and its applications. In *International Symposium on Security and Privacy in Social Networks and Big Data*, pages 110–122. Springer, 2019.

[Ham17] Mike Hamburg. Post-quantum cryptography proposal: ThreeBears (Round 1), 2017. Accessed December 18, 2021: https://csrc.nist.gov/projects/post-quantum-cryptography/round-1-submissions.

[Ham19] Mike Hamburg. Post-quantum cryptography proposal: ThreeBears (Round 2), 2019. Accessed December 18, 2021: https://csrc.nist.gov/projects/post-quantum-cryptography/round-2-submissions.

[HEAD01] Xiao-Yu Hu, Evangelos Eleftheriou, D-M Arnold, and Ajay Dholakia. Efficient implementations of the sum-product algorithm for decoding LDPC codes. In *IEEE Global Telecommunications Conference*, pages 1–6. IEEE, 2001.

[HGSSW03] Nick Howgrave-Graham, Joseph H. Silverman, Ari Singer, and William Whyte. NAEP: Provable security in the presence of decryption failures. Cryptology ePrint Archive, Report 2003/172, 2003.

[HHLW20] Yiming Huang, Miaoqing Huang, Zhongkui Lei, and Jiaxuan Wu. A pure hardware implementation of CRYSTALS-KYBER PQC algorithm through resource reuse. *IEICE Electronics Express*, pages 1–6, 2020.

[HL19] Björn Haase and Benoît Labrique. AuCPace: Efficient verifier-based PAKE protocol tailored for the IIoT. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, 2019(2):1–48, Feb. 2019.

[HOKG18] James Howe, Tobias Oder, Markus Krausz, and Tim Güneysu. Standard lattice-based key encapsulation on embedded devices. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, 2018(3):372–393, Aug. 2018.

[HP21] Daniel Heinz and Thomas Pöppelmann. Combined fault and DPA protection for lattice-based cryptography. Cryptology ePrint Archive, Report 2021/101, 2021.

[HPS98] Jeffrey Hoffstein, Jill Pipher, and Joseph H Silverman. NTRU: A ring-based public key cryptosystem. In *International Algorithmic Number Theory Symposium*, pages 267–288. Springer, 1998.

[HPS+20] Julius Hermelink, Thomas Pöppelmann, Marc Stöttinger, Yi Wang, and Yong Wan. Quantum safe authenticated key exchange protocol for automotive application. In *18th escar Europe: embedded security in cars*. Ruhr-Universität Bochum, Universitätsbibliothek, 2020.

[Hsi12] Homer Hsing. OpenCores. SHA3 (KECCAK), 2012. Accessed October 01, 2018: https://opencores.org/projects/sha3.

[IEE16] IEEE. IEEE standard for wireless access in vehicular environments–security services for applications and management messages. *IEEE Std 1609.2-2016*, pages 1–240, 2016.

[J+20] David Jao et al. Supersingular isogeny key encapsulation, 2020. Accessed December 18, 2021: https://sike.org/files/SIDH-spec.pdf.

[JDF11] David Jao and Luca De Feo. Towards quantum-resistant cryptosystems from supersingular elliptic curve isogenies. In *Post-Quantum Cryptography*, pages 19–34. Springer, 2011.

[Jon01] Douglas W. Jones. Modulus without division, a tutorial, 2001. Accessed December 18, 2021: http://homepage.cs.uiowa.edu/~jones/bcd/mod.shtml.

[KG16] Petter Källström and Oscar Gustafsson. Fast and area efficient adder for wide data in recent Xilinx FPGAs. In *26th International Conference on Field Programmable Logic and Applications (FPL)*, pages 1–4. IEEE, 2016.

[KJJ99] Paul Kocher, Joshua Jaffe, and Benjamin Jun. Differential power analysis. In *Annual International Cryptology Conference*, pages 388–397. Springer, 1999.

[KLC+17] Po-Chun Kuo, Wen-Ding Li, Yu-Wei Chen, Yuan-Che Hsu, Bo-Yuan Peng, Chen-Mou Cheng, and Bo-Yin Yang. High performance post-quantum key exchange on FPGAs. Cryptology ePrint Archive, Report 2017/690, 2017.

[KO62] Anatolii Alekseevich Karatsuba and Yu P. Ofman. Multiplication of many-digital numbers by automatic computers. In *Doklady Akademii Nauk*, pages 293–294. Russian Academy of Sciences, 1962.

[Koc96] Paul C. Kocher. Timing attacks on implementations of Diffie–Hellman, RSA, DSS, and other systems. In *Annual International Cryptology Conference*, pages 104–113. Springer, 1996.

[Koc08] Cetin Kaya Koc. *Cryptographic Engineering*. Springer, 2008.

[Kro82] Leopold Kronecker. Grundzüge einer arithmetischen Theorie der algebraischen Größen, 1882.

[KRSS19] Matthias J. Kannwischer, Joost Rijneveld, Peter Schwabe, and Ko Stoffelen. pqm4: Testing and benchmarking NIST PQC on ARM Cortex-M4. Cryptology ePrint Archive, Report 2019/844, 2019.

[KY09] Abdel Alim Kamal and Amr M. Youssef. An FPGA implementation of the NTRUEncrypt cryptosystem. In *International Conference on Microelectronics*, pages 209–212. IEEE, 2009.

[L+01] Daniel Lieman et al. Standard specification for public-key cryptographic techniques based on hard problems over lattices. *IEEE P1363*, 1:D2, 2001.

[LC04] Shu Lin and Daniel J. Costello. *Error Control Coding, Second Edition.* Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 2004.

[LG20] Michael X. Lyons and Kris Gaj. Sampling from discrete distributions in combinational hardware with application to post-quantum cryptography. In *Design, Automation & Test in Europe Conference & Exhibition (DATE)*. IEEE, 2020.

[LLJ+17] Xianhui Lu, Yamin Liu, Dingding Jia, Haiyang Xue, Jingnan He, and Zhenfei Zhang. Supporting documentation: LAC (Round 1 submission), 2017. Accessed December 18, 2021: https://csrc.nist.gov/Projects/Post-Quantum-Cryptography/Round-1-Submissions.

[LPR10] Vadim Lyubashevsky, Chris Peikert, and Oded Regev. On ideal lattices and learning with errors over rings. In *Annual International Conference on the Theory and Applications of Cryptographic Techniques*, pages 1–23. Springer, 2010.

[LS15] Adeline Langlois and Damien Stehlé. Worst-case to average-case reductions for module lattices. *Designs, Codes and Cryptography*, 75(3):565–599, 2015.

[LS19] Vadim Lyubashevsky and Gregor Seiler. NTTRU: Truly fast NTRU using NTT. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, 2019(3):180–201, May 2019.

[LW15] Bingxin Liu and Huapeng Wu. Efficient architecture and implementation for NTRUEncrypt system. In *IEEE 58th International Midwest Symposium on Circuits and Systems (MWSCAS)*, pages 1–4. IEEE, 2015.

[LW16] Bingxin Liu and Huapeng Wu. Efficient multiplication architecture over truncated polynomial ring for NTRUEncrypt system. In *IEEE International Symposium on Circuits and Systems (ISCAS)*, pages 1174–1177. IEEE, 2016.

[Mar20] Adrian Marotzke. A constant time full hardware implementation of streamlined NTRU Prime. In *International Conference on Smart Card Research and Advanced Applications*, pages 3–17. Springer, 2020.

[Mas69] James Massey. Shift-register synthesis and BCH decoding. *IEEE Transactions on Information Theory*, 15(1):122–127, 1969.

[MBC+08] Enrico Macii, Leticia Bolzani, Andrea Calimera, Alberto Macii, and Massimo Poncino. Integrating clock gating and power gating for combined dynamic and leakage power optimization in digital CMOS circuits. In *11th EUROMICRO Conference on Digital System Design Architectures, Methods and Tools*, pages 298–303. IEEE, 2008.

[MGTF19] Vincent Migliore, Benoît Gérard, Mehdi Tibouchi, and Pierre-Alain Fouque. Masking Dilithium: Efficient implementation and side-channel evaluation. In *Applied Cryptography and Network Security*, pages 344–362. Springer, 2019.

[MKÖ+20] Ahmet Can Mert, Emre Karabulut, Erdinç Öztürk, Erkay Savaş, Michela Becchi, and Aydin Aysu. A flexible and scalable NTT hardware: applications from homomorphically encrypted deep learning to post-quantum cryptography. In *Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pages 346–351. IEEE, 2020.

[MNP+20] Ben Marshall, G. Richard Newell, Dan Page, Markku-Juhani O. Saarinen, and Claire Wolf. The design of scalar AES instruction set extensions for RISC-V. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, 2021(1):109–136, Dec. 2020.

[Mon85] Peter Montgomery. Modular multiplication without trial division. *Mathematics of Computation*, 44(170):519–521, 1985.

[Mos18] Michele Mosca. Cybersecurity in an era with quantum computers: will we be ready? *IEEE Security & Privacy*, 16(5):38–41, 2018.

[MPP21] Ben Marshall, Daniel Page, and Thinh Hung Pham. A lightweight ISE for ChaCha on RISC-V. In *IEEE 32nd International Conference on Application-specific Systems, Architectures and Processors (ASAP)*, pages 25–32. IEEE, 2021.

[MPWZ21] Georg Maringer, Sven Puchinger, and Antonia Wachter-Zeh. Higher rates and information-theoretic analysis for the RLWE channel. In *IEEE Information Theory Workshop (ITW)*, pages 1–5. IEEE, 2021.

[Nat16a] National Institute of Standards and Technology (NIST). Announcing request for nominations for public-key post-quantum cryptographic algorithms. Federal Register, 81, 92787-92788, 2016. Accessed December 18, 2021: https://csrc.nist.gov/news/2016/public-key-post-quantum-cryptographic-algorithms.

[Nat16b] National Security Agency. Commercial national security algorithm suite, 2016. Accessed December 18, 2021: https://apps.nsa.gov/iaarchive/programs/iad-initiatives/cnsa-suite.cfm.

[NIS21] NIST. Post-quantum cryptography - workshops and timeline, 2021. Accessed December 18, 2021: https://csrc.nist.gov/Projects/post-quantum-cryptography/workshops-and-timeline.

[NRR06] Svetla Nikova, Christian Rechberger, and Vincent Rijmen. Threshold implementations against side-channel attacks and glitches. In *International Conference on Information and Communications Security*, pages 529–545. Springer, 2006.

[NRS11] Svetla Nikova, Vincent Rijmen, and Martin Schläffer. Secure hardware implementation of nonlinear functions in the presence of glitches. *Journal of Cryptology*, 24(2):292–321, 2011.

[OG17] Tobias Oder and Tim Güneysu. Implementing the NewHope-Simple key exchange on low-cost FPGAs. In *International Conference on Cryptology and Information Security in Latin America (LATINCRYPT)*, pages 128–142. Springer, 2017.

[OM20] Jeffrey Osier-Mixon. RISC-V: An open approach to system security, 2020. Accessed December 18, 2021: https://riscv.org/blog/2020/03/risc-v-an-open-approach-to-system-security/.

[OSPG18] Tobias Oder, Tobias Schneider, Thomas Pöppelmann, and Tim Güneysu. Practical CCA2-secure and masked Ring-LWE implementation. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, 2018(1):142–174, Feb. 2018.

[PG12] Thomas Pöppelmann and Tim Güneysu. Towards efficient arithmetic for lattice-based cryptography on reconfigurable hardware. In *International Conference on Cryptology and Information Security in Latin America (LATINCRYPT)*, pages 139–158. Springer, 2012.

[PGZMG21] Sabine Pircher, Johannes Geier, Alexander Zeh, and Daniel Müller-Gritschneder. Exploring the RISC-V vector extension for the Classic McEliece post-quantum cryptosystem. In *22nd International Symposium on Quality Electronic Design (ISQED)*, pages 401–407. IEEE, 2021.

[PNPM15] Thomas Pöppelmann, Michael Naehrig, Andrew Putnam, and Adrian Macias. Accelerating homomorphic evaluation on reconfigurable hardware (extended version). In *International Workshop on Cryptographic Hardware and Embedded Systems*, pages 143–163. Springer, 2015.

[POG15] Thomas Pöppelmann, Tobias Oder, and Tim Güneysu. High-performance ideal lattice-based cryptography on 8-bit ATxmega microcontrollers. In *International Conference on Cryptology and Information Security in Latin America (LATINCRYPT)*, pages 346–365. Springer, 2015.

[PT19] Thales Bandiera Paiva and Routo Terada. A timing attack on the HQC encryption scheme. In *International Conference on Selected Areas in Cryptography (SAC)*, pages 551–573. Springer, 2019.

[QLW13] Chen Qian, Weilong Lei, and Zhaocheng Wang. Low complexity LDPC decoder with modified Sum-Product algorithm. *Tsinghua Science and Technology*, 18(1):57–61, 2013.

[RBN+15] Oscar Reparaz, Begül Bilgin, Svetla Nikova, Benedikt Gierlichs, and Ingrid Verbauwhede. Consolidating masking schemes. In *Annual Cryptology Conference*, pages 764–783. Springer, 2015.

[RdCR+16] Oscar Reparaz, Ruan de Clercq, Sujoy Sinha Roy, Frederik Vercauteren, and Ingrid Verbauwhede. Additively homomorphic ring-LWE masking. In *Post-Quantum Cryptography*, pages 233–244. Springer, 2016.

[Reg09] Oded Regev. On lattices, learning with errors, random linear codes, and cryptography. *Journal of the ACM (JACM)*, 56(6):1–40, 2009.

[Ric03] Tom Richardson. Error floors of LDPC codes. In *Annual Allerton Conference on Communication Control and Computing*, pages 1426–1435, 2003.

[RIS21] RISC-V. History of RISC-V, 2021. Accessed December 18, 2021: https://riscv.org/about/history/.

[RM19] Debapriya Basu Roy and Debdeep Mukhopadhyay. Post quantum ECC on FPGA platform. Cryptology ePrint Archive, Report 2019/568, 2019.

[RMGS20] Andrew H. Reinders, Rafael Misoczki, Santosh Ghosh, and Manoj R. Sastry. Efficient BIKE hardware design with constant-time decoder. In *IEEE International Conference on Quantum Computing and Engineering (QCE)*, pages 197–204. IEEE, 2020.

[RRVV15] Oscar Reparaz, Sujoy Sinha Roy, Frederik Vercauteren, and Ingrid Verbauwhede. A masked ring-LWE implementation. In *International Workshop on Cryptographic Hardware and Embedded Systems*, pages 683–702. Springer, 2015.

[RSRCB20] Prasanna Ravi, Sujoy Sinha Roy, Anupam Chattopadhyay, and Shivam Bhasin. Generic side-channel attacks on CCA-secure lattice-based PKE and KEMs. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, 2020(3):307–335, Jun. 2020.

[RVM+14] Sujoy Sinha Roy, Frederik Vercauteren, Nele Mentens, Donald Donglong Chen, and Ingrid Verbauwhede. Compact Ring-LWE cryptoprocessor. In *International Workshop on Cryptographic Hardware and Embedded Systems*, pages 371–391. Springer, 2014.

[Saa17] Markku-Juhani O. Saarinen. Supporting documentation: HILA5 (Round 1 submission), 2017. Accessed December 18, 2021: https://csrc.nist.gov/Projects/Post-Quantum-Cryptography/Round-1-Submissions.

[Saa18] Markku-Juhani O. Saarinen. Arithmetic coding and blinding countermeasures for lattice signatures. *Journal of Cryptographic Engineering*, 8(1):71–84, 2018.

[SAJA21] Hwajeong Seo, Mila Anastasova, Amir Jalali, and Reza Azarderakhsh. Supersingular isogeny key encapsulation (SIKE) Round 2 on ARM Cortex-M4. *IEEE Transactions on Computers*, 70(10):1705–1718, 2021.

[Sch13] Manfred Schroeder. *Number Theory in Science and Communication: With Applications in Cryptography, Physics, Digital Information, Computing, and Self-Similarity*. Springer Series in Information Sciences. Springer, 2013.

[Sho94] Peter W. Shor. Algorithms for quantum computation: discrete logarithms and factoring. In *35th annual Symposium on Foundations of Computer Science*, pages 124–134. IEEE, 1994.

[SMG15] Tobias Schneider, Amir Moradi, and Tim Güneysu. Arithmetic addition over Boolean masking. In *International Conference on Applied Cryptography and Network Security*, pages 559–578. Springer, 2015.

[Sod21] Sodium. libsodium, 2021. Accessed December 18, 2021: https://github.com/jedisct1/libsodium.

[SPOG19] Tobias Schneider, Clara Paglialonga, Tobias Oder, and Tim Güneysu. Efficiently masking binomial sampling at arbitrary orders for lattice-based crypto. In *IACR International Workshop on Public Key Cryptography*, pages 534–564. Springer, 2019.

[SR15] Hermann Seuschek and Stefan Rass. Side-channel leakage models for RISC instruction set architectures from empirical data. In *Euromicro Conference on Digital System Design*, pages 423–430. IEEE, 2015.

[SRB20] Sujoy Sinha Roy and Andrea Basso. High-speed instruction-set coprocessor for lattice-based key encapsulation mechanism: Saber in hardware. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, 2020(4):443–466, Aug. 2020.

[SRP+18] Pasquale Davide Schiavone, Davide Rossi, Antonio Pullini, Alfio Di Mauro, Francesco Conti, and Luca Benini. Quentin: an ultra-low-power PULPissimo SoC in 22nm FDX. In *IEEE SOI-3D-Subthreshold Microelectronics Technology Unified Conference (S3S)*, pages 1–3, 2018.

[SRSWZ20] Thomas Schamberger, Julian Renner, Georg Sigl, and Antonia Wachter-Zeh. A power side-channel attack on the CCA2-secure HQC KEM. In

International Conference on Smart Card Research and Advanced Applications, pages 119–134. Springer, 2020.

[SSJA20] Hwajeong Seo, Pakize Sanal, Amir Jalali, and Reza Azarderakhsh. Optimized implementation of SIKE Round 2 on 64-bit ARM Cortex-A processors. *IEEE Transactions on Circuits and Systems I: Regular Papers*, 67(8):2659–2671, 2020.

[TGS] Andreas Traber, Michael Gautschi, and Pasquale Davide Schiavone. PULP RI5CY: User manual. Apr. 2019.

[Too63] Andrei L. Toom. The complexity of a scheme of functional elements realizing the multiplication of integers. In *Soviet Mathematics Doklady*, pages 714–716. Russian Academy of Sciences, 1963.

[TU16] Ehsan Ebrahimi Targhi and Dominique Unruh. Post-quantum security of the Fujisaki-Okamoto and OAEP transforms. In *Theory of Cryptography Conference*, pages 192–216. Springer, 2016.

[TZS+16] Andreas Traber, Florian Zaruba, Sven Stucki, Antonio Pullini, Germain Haugou, Eric Flamand, Frank K. Gurkaynak, and Luca Benini. PULPino: A small single-core RISC-V SoC. In *3rd RISCV Workshop*, 2016.

[Uni21] Pennsylvania State University. Pearson correlation coefficient r, 2021. Accessed December 18, 2021: https://online.stat.psu.edu/stat462/node/96/.

[vdB20] Stefan van den Berg. RISC-V implementation of the NaCl-library, 2020. Accessed December 18, 2021: https://pure.tue.nl/ws/portalfiles/portal/169647601/Berg_S._ES_CSE.pdf.

[WAE19] Andrew Waterman and Krste Asanović (Editors). The RISC-V instruction set manual, Volume I: User-Level ISA, Document Version 20191213, 2019. RISC-V Foundation. Accessed December 18, 2021: https://riscv.org/wp-content/uploads/2019/12/riscv-spec-20191213.pdf.

[Wel47] Bernard L. Welch. The generalization of student's problem when several different population variances are involved. *Biometrika*, 34(1-2):28–35, 1947.

[WGY20] Bin Wang, Xiaozhuo Gu, and Yingshan Yang. Saber on ESP32. In *International Conference on Applied Cryptography and Network Security*, pages 421–440. Springer, 2020.

[Why17] William Whyte. NTRU Open Source Project, 2017. Accessed December 20, 2018: https://github.com/NTRUOpenSourceProject/NTRUEncrypt.

[WR20] Matthew Walters and Sujoy Sinha Roy. Constant-time BCH error-correcting code. In *IEEE International Symposium on Circuits and Systems (ISCAS)*, pages 1–5. IEEE, 2020.

[WTBB+20] Guillaume Wafo-Tapa, Slim Bettaieb, Loïc Bidoux, Philippe Gaborit, and Etienne Marcatel. A practicable timing attack against HQC and its countermeasure. *Advances in Mathematics of Communications*, 2020.

[WTJ+20] Wen Wang, Shanquan Tian, Bernhard Jungk, Nina Bindel, Patrick Longa, and Jakub Szefer. Parameterized hardware accelerators for lattice-based cryptography and their application to the HW/SW co-design of qTESLA. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, 2020(3):269–306, Jun. 2020.

[XHY+20] Guozhu Xin, Jun Han, Tianyu Yin, Yuchao Zhou, Jianwei Yang, Xu Cheng, and Xiaoyang 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*, 2020.

[ZGF20] Davide Zoni, Andrea Galimberti, and William Fornaciari. Efficient and scalable FPGA-oriented design of QC-LDPC bit-flipping decoders for post-quantum cryptography. *IEEE Access*, 8:163419–163433, 2020.

[ZJGS17] Yunlei Zhao, Zhengzhong Jin, Boru Gong, and Guangye Sui. Supporting documentation: KCL (Round 1 submission), 2017. Accessed December 18, 2021: https://csrc.nist.gov/Projects/Post-Quantum-Cryptography/Round-1-Submissions.

[ZYC+20] Neng Zhang, Bohan Yang, Chen Chen, Shouyi Yin, Shaojun Wei, and Leibo 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.

[ZZY+20] Yihong Zhu, Min Zhu, Bohan Yang, Wenping Zhu, Chenchen Deng, Chen Chen, Shaojun Wei, and Leibo Liu. A high-performance hardware implementation of Saber based on Karatsuba algorithm. Cryptology ePrint Archive, Report 2020/1037, 2020.