VIETNAM NATIONAL UNIVERISTY - HO CHI MINH CITY
HO CHI MINH CITY UNIVERSITY OF TECHNOLOGY
FACULTY OF COMPUTER SCIENCE AND ENGINEERING

GRADUATION THESIS

ACCELERATING THE CRYSTALS-KYBER ALGORITHM ON THE SOC-FPGA PLATFORM

Major: COMPUTER ENGINEERING

THESIS COMMITTEE: COMPUTER ENGINEERING 01

SUPERVISOR: ASSOC. PROF. DR. PHAM QUOC CUONG

REVIEWER: MSC. PHAM KIEU NHAT ANH

—o0o—

STUDENT 1: NGUYEN PHUC ANH – 2153166

STUDENT 2: NGUYEN THANH THAO NHI – 2152840

STUDENT 3: TRAN MINH TUAN – 2152336

HO CHI MINH CITY, JUNE 2025

HĐ1 CLC KTMT – 159

ĐẠI HỌC QUỐC GIA TP.HCM
TRƯỜNG ĐẠI HỌC BÁCH KHOA

CỘNG HÒA XÃ HỘI CHỦ NGHĨA VIỆT NAM
Độc lập - Tự do - Hạnh phúc

KHOA:KH & KT MÁY TÍNH
BỘ MÔN: KỸ THUẬT MÁY TÍNH

NHIỆM VỤ LUẬN VĂN/ ĐỒ ÁN TỐT NGHIỆP

Chú ý: Sinh viên phải dán tờ này vào trang nhất của bản thuyết trình

HỌ VÀ TÊN: Nguyễn Phúc Anh — MSSV: 2153166
HỌ VÀ TÊN: Nguyễn Thanh Thảo Nhi — MSSV: 2152840
HỌ VÀ TÊN: Trần Minh Tuấn — MSSV: 2152336
NGÀNH: Kỹ thuật Máy tính — LỚP: MT21KTTN

1. Đầu đề luận văn/ đồ án tốt nghiệp:
Tăng tốc thuật toán CRYSTALS-Kyber trên nền tảng SoC-FPGA *(Accelerating the CRYSTALS-Kyber algorithm on the SoC-FPGA platform)*

2. Nhiệm vụ (yêu cầu về nội dung và số liệu ban đầu):

- Research suitable data security algorithms (CRYSTALS-Kyber).
- Investigate the FPGA-based SoC hardware acceleration approach.
- Propose the system architecture and hardware core for the selected security model.
- Implement the proposed system
- Choose a test application and build the system.

3. Ngày giao nhiệm vụ: 06/01/2025

4. Ngày hoàn thành nhiệm vụ: 22/5/2025

5. Họ tên giảng viên hướng dẫn: **Phần hướng dẫn:**

1) Phạm Quốc Cường

Nội dung và yêu cầu LVTN/ ĐATN đã được thông qua Bộ môn.

Ngày 06 háng 01 năm 2025

CHỦ NHIỆM BỘ MÔN
(Ký và ghi rõ họ tên)

Phạm Quốc Cường

GIẢNG VIÊN HƯỚNG DẪN CHÍNH
(Ký và ghi rõ họ tên)

Phạm Quốc Cường

PHẦN DÀNH CHO KHOA, BỘ MÔN:
Người duyệt (chấm sơ bộ):
Đơn vị:
Ngày bảo vệ:
Điểm tổng kết:
Nơi lưu trữ LVTN/ĐATN:

HB1 CLC KTMT - 159

TRƯỜNG ĐẠI HỌC BÁCH KHOA
KHOA KH & KT MÁY TÍNH

CỘNG HÒA XÃ HỘI CHỦ NGHĨA VIỆT NAM
Độc lập - Tự do - Hạnh phúc

Ngày 15 tháng 5 năm 2025

PHIẾU ĐÁNH GIÁ LUẬN VĂN/ ĐỒ ÁN TỐT NGHIỆP

(Dành cho người hướng dẫn)

1. HỌ VÀ TÊN: Nguyễn Phúc Anh — MSSV: 2153166
Nguyễn Thanh Thảo Nhi — MSSV: 2152840
Trần Minh Tuấn — MSSV: 2152336
NGÀNH: Kỹ thuật Máy tính — LỚP: MT21KTM

2. Đề tài:
Tăng tốc thuật toán CRYSTALS-Kyber trên nền tảng SoC-FPGA *(Accelerating the CRYSTALS-Kyber algorithm on the SoC-FPGA platform)*

3. Họ tên người hướng dẫn: Phạm Quốc Cường

4. Tổng quát về bản thuyết minh:
Số trang: — Số chương:
Số bảng số liệu — Số hình vẽ:
Số tài liệu tham khảo: — Phần mềm tính toán:
Hiện vật (sản phẩm)

5. Những ưu điểm chính của LV/ ĐATN:
- Sinh viên hoàn thành tốt nhiệm vụ đặt ra
- Hệ thống hoạt động ổn định
- Sinh viên làm việc nghiêm túc, báo cáo đạt yêu cầu
- Sinh viên đã tham dự cuộc thi thiết kế vi mạch thông minh lần thứ 2 do Khu Công nghệ cao TPHCM tổ chức và được vào top 25

6. Những thiếu sót chính của LV/ĐATN:
- Chưa đạt được hiệu suất tính toán như kỳ vọng

7. Đề nghị: Được bảo vệ ☒ Bổ sung thêm để bảo vệ ☐ Không được bảo vệ ☐
8. Các câu hỏi SV phải trả lời trước Hội đồng:
a. Đâu là rào cản cho việc gia tăng hiệu suất của hệ thống?

9. Đánh giá chung (bằng chữ: Xuất sắc, Giỏi, Khá, TB): Điểm: 9.3/10

Ký tên (ghi rõ họ tên)

Phạm Quốc Cường

THĐ1 CLC – KTMT – 159

TRƯỜNG ĐẠI HỌC BÁCH KHOA
KHOA KH & KT MÁY TÍNH

CỘNG HÒA XÃ HỘI CHỦ NGHĨA VIỆT NAM
Độc lập - Tự do - Hạnh phúc

Ngày 9 tháng 5 năm 2025

PHIẾU ĐÁNH GIÁ LUẬN VĂN/ ĐỒ ÁN TỐT NGHIỆP

(Dành cho người ~~hướng dẫn~~/phản biện)

1. Họ và tên SV: Nguyễn Phúc Anh, Nguyễn Thanh Thảo Nhi, Trần Minh Tuấn
   MSSV: 2153166, 2152840,2152336 Ngành (chuyên ngành): Kỹ thuật Máy tính
2. Đề tài: Accelerating the CRYSTALS-Kyber Algorithm on the SoC-FPGA

3. Họ tên người ~~hướng dẫn~~/phản biện: Phạm Kiều Nhật Anh

4. Tổng quát về bản thuyết minh:
   Số trang: 89 Số chương: 6
   Số bảng số liệu: 11 Số hình vẽ: 35
   Số tài liệu tham khảo: 41 Phần mềm tính toán: 1
   Hiện vật (sản phẩm)
5. Những ưu điểm chính của LV/ ĐATN:
   - Students present all of the results of the project clearly
   - Students do demonstration well
   - Students have strong knowledge on background of the project

6. Những thiếu sót chính của LV/ĐATN:
   - Students need to do more in evaluation
   - More graphs must be draw to compare with other data size
   - Conclusion must be more specific

7. Đề nghị: Được bảo vệ ☑ Bổ sung thêm để bảo vệ ☐ Không được bảo vệ ☐
8. Các câu hỏi SV phải trả lời trước Hội đồng:
   a. What are the advantages and disadvantages of CRYSTALS-Kyber when this algorithm is implemented on FPGA?

   b. Can this architecture operate well when the input data has a larger size?

   c. Compare the results between demonstration and implementation on hardware?

9. Đánh giá chung (bằng chữ: Xuất sắc, Giỏi, Khá, TB): Điểm : 8.2 /10

Ký tên (ghi rõ họ tên)

PHẠM KIỀU NHẬT ANH

This thesis is dedicated for our parents and our instructors at HCMUT.

CONTENTS

LIST OF FIGURES

LIST OF TABLES

COMMITMENT

This project was initially inspired by our supervisor's concept. We conducted all the implementation ourselves over the course of the semester. We consulted various open-source projects, articles, and data sources, and you can find references to these resources at the end of the thesis.

We take full responsibility for any potential copyright infringements within the thesis. The thesis adheres to the guidelines and requirements established by the Faculty of Computer Science and Engineering at Ho Chi Minh University of Technology and has been completed accordingly.

Sincerely,
Group of Authors

Nguyen Phuc Anh
Nguyen Thanh Thao Nhi
Tran Minh Tuan

ACKNOWLEDGMENT

First and foremost, we would like to express our sincere gratitude to our supervisor, Assoc. Prof. Dr. Pham Quoc Cuong, for his unwavering support and guidance throughout the course of this project. His mentorship has been instrumental in shaping our academic journey, offering invaluable insights, continuous encouragement, and expert advice. Without his dedicated involvement and support at every stage, this thesis would not have been possible.

We wish to extend our heartfelt thanks to Mr. Huynh Phuc Nghi, Mr. Ton Huynh Long, and Mr. Nguyen Thanh Loc from the CE Laboratory for generously sharing their time and expertise. Their willingness to exchange knowledge and provide technical discussions has greatly enriched our understanding and contributed significantly to the successful completion of this work.

We also extend our sincere appreciation to the esteemed educators of the Faculty of Computer Science and Engineering, and to the Ho Chi Minh City University of Technology as a whole. Over the past four years, their unwavering commitment to imparting knowledge has been an enduring source of learning. Their support, encouragement, and insightful ideas have significantly contributed to the successful completion of the project.

Last but certainly not least, we cannot overlook the significance of acknowledging our friends and family. The unwavering love and blessings from our late parents, the care and companionship of friends and acquaintances who kept our spirits high, have all played a pivotal role in guiding us to this significant stage in our lives. We are grateful for their unwavering support during challenging moments, as they motivated us to overcome obstacles and pursue our aspirations.

In closing, we extend our heartfelt wishes for your continued well-being and success in all your noble endeavors.

ABSTRACT

This research project focuses on the design and implementation of a hardware accelerator for the CRYSTALS-Kyber algorithm, a post-quantum cryptography (PQC) scheme, on a Field-Programmable Gate Array (FPGA) platform. The urgency for this work stems from the looming threat posed by quantum computers to conventional cryptographic systems like RSA and ECC, which rely on the difficulty of factoring large numbers or solving discrete logarithms, problems readily solvable by quantum algorithms. CRYSTALS-Kyber, a lattice-based key encapsulation mechanism (KEM), has emerged as a promising candidate for standardization by NIST due to its robustness against both classical and quantum attacks.

The project leverages FPGA technology to accelerate Kyber's computationally demanding operations, aiming to achieve significant performance improvements compared to software-based implementations. Critical functions within the Kyber algorithm, such as the Number Theoretic Transform (NTT) for fast polynomial multiplication and the SHA3 core for secure cryptographic hashing, are implemented and optimized for hardware.

The implemented hardware design is rigorously tested, benchmarked, and compared against software implementations of Kyber. Metrics such as execution time (measured in clock cycles), resource utilization (logic elements, memory, DSP blocks), and functional equivalence with software models are used to evaluate the effectiveness of the hardware acceleration.

The final design is ported and demonstrated on the ZCU106 board. This board, featuring a Zynq UltraScale+ MPSoC, integrates a high-performance processing system and configurable logic, making it well-suited for advanced cryptographic applications.

1

INTRODUCTION

1.1. INTRODUCTION

The rapid development of quantum computing introduces unprecedented challenges to current cryptographic systems, such as RSA and ECC, as these are susceptible to attacks facilitated by quantum algorithms like Shor's algorithm. This evolution threatens secure communication in critical industries, including IoT, healthcare, telecommunications, and finance, where data security is paramount.

To mitigate this risk, the field of post-quantum cryptography (PQC) has emerged, focusing on cryptographic algorithms resistant to quantum-based attacks. NIST's Post-Quantum Cryptography Standardization project has identified promising candidates for quantum-resistant encryption. Among these, CRYSTALS-Kyber—a lattice-based key encapsulation mechanism (KEM)—stands out for its robust security guarantees, computational efficiency, and suitability for real-world applications.

In parallel, the growing demand for AI-driven services and intelligent edge devices has led to significant challenges in data processing architectures. Traditionally, data collected by edge devices is transmitted to centralized servers for processing and then sent back to the edge. However, this approach introduces critical drawbacks: increased latency, potential vulnerabilities in data transmission security, and the generation of massive volumes of data that strain communication infrastructure. To address these challenges, a paradigm shift is required, promoting the processing of data directly at the point of collection—on edge devices themselves. This approach not only enhances response times and reduces dependency on network stability but also improves data privacy and security. In this context, implementing cryptographic operations such as en-

cryption and decryption locally at the edge becomes essential to ensuring secure, efficient, and real-time data handling.

Performing cryptographic operations at the edge demands solutions that are both high-performance and energy-efficient. FPGA platforms are particularly well-suited for this task due to their inherent parallelism, reconfigurability, and ability to deliver low-power, high-throughput performance. Demonstrating efficient Kyber encryption on FPGA not only validates its applicability for post-quantum security but also ensures the design is practical for deployment in edge devices, where computational resources and energy budgets are constrained.

Consequently, this project draws its motivation from both the urgent need for quantum-resistant encryption methods and the architectural shifts in edge computing. By implementing the Kyber algorithm on an FPGA platform, this work aims to combine the computational efficiency of hardware acceleration with the robust, quantum-resistant properties of Kyber. The outcome will contribute to practical, high-performance encryption solutions tailored to critical industries requiring long-term data security and operational viability in edge computing environments.

1.2. RESEARCH OBJECTIVE

The study of CRYSTALS-Kyber's theoretical aspects, alongside our research into current trends in Post-Quantum Cryptography (PQC) algorithms, has formed the foundation for the objectives of this project. Building on the progress made during the Computer Engineering Project phase, we have established six key goals for this initiative:

1. **Achieving a comprehensive understanding** of the theory behind Kyber Public Key Encryption (PKE), ensuring a solid theoretical foundation for subsequent work.

2. **Examining existing hardware implementations** of Kyber PKE, critically assessing their strengths and weaknesses, and using this analysis to propose an optimized solution.

3. **Implementing logic design models** for the CRYSTALS-Kyber Accelerator cores, followed by functional simulations to verify the accuracy and reliability of the design.

4. **Optimizing key functions**, including:

    - The Number-Theoretic Transform (NTT) for efficient polynomial multiplication.

- The Hash core for secure cryptographic hashing.

5. **Deploying the design on the Xilinx ZCU106 board**, utilizing AXI4 protocols and the ZYNQ framework to leverage established techniques in logic implementation and enhance the performance of the system.

6. **Developing a demonstration process** that captures live images and demonstrates the encryption-decryption process, interfacing with the cores on the development board.

7. **Conducting a comprehensive performance evaluation**, analyzing the resource usage and performance metrics of various design alternatives to determine the most efficient approach.

The first three objectives have been successfully completed during the Computer Engineering Project phase, while The remaining objectives are addressed and finalized during the Capstone Project phase.

1.3. RESEARCH SCOPE

The scope of this project is focused on the CRYSTALS-Kyber post-quantum cryptography algorithm. The encryption scheme is structured into two primary components: Kyber.CPAPKE for public-key encryption and Kyber.CCAKEM for key encapsulation. There are three variants of this algorithm: Kyber-512, Kyber-768, and Kyber-1024, with increasing security levels. Given the available knowledge, capabilities, and time constraints, this project will focus on the hardware implementation of Kyber.CPAPKE for Kyber-512.

This project specifically excludes the implementation of the algorithm on ASICs, due to the increased complexity and costs associated with ASIC design. Instead, the focus is on utilizing FPGA resources to develop a high-performance and resource-efficient solution for the Kyber-512 PKE algorithm.

1.4. RESEARCH SUBJECT

The research subject of this project is:

- The Number-Theoretic Transform, Hashing and Kyber-512 PKE algorithm.
- The AXI4 protocol.
- The Xilinx ZCU106 platform.
- Architecture of CRYSTALS-Kyber Accelerator Core on Xilinx Zynq Ultrascale+ PL.

1

1.5. OUTLINE

- **Chapter 1 - Introduction**
  This chapter provides an overview of the research problem, highlighting the urgent need for post-quantum cryptography and significance of optimizing the CRYSTALS-Kyber algorithm. It outlines the challenges in achieving efficient implementations on resource-constrained platforms, defines the objectives of the thesis, and offers a concise summary of the project scope and contributions.

- **Chapter 2 - Background and Related work**
  This chapter presents the foundational knowledge relevant to the research. It includes a detailed explanation of the CRYSTALS-Kyber algorithm, its role in lattice-based cryptography, and the theoretical underpinnings of post-quantum security. Additionally, it discusses about relevant study and existing implementation.

- **Chapter 3 - Proposed Architecture**
  This chapter introduces the proposed hardware-software co-design architecture for accelerating CRYSTALS-Kyber on the SoC-FPGA platform. It provides a high-level overview of the system design, supported by block diagrams illustrating the interaction between the FPGA fabric and the processing system, along with the data flow for cryptographic operations.

- **Chapter 4 - Implementation**
  This chapter elaborates on the implementation details of the proposed architecture. It covers the hardware acceleration of key CRYSTALS-Kyber operations on the FPGA and the software components running on the SoC's ARM processor. The chapter also addresses the integration and optimization strategies employed to enhance performance.

- **Chapter 5 - Evaluation**
  This chapter evaluates the performance of the implemented system through experimental testing. It presents the methodology for benchmarking, including test scenarios and metrics such as timing, throughput, and resource utilization. The results are compared with existing implementations to demonstrate the improvements achieved by the proposed approach.

- **Chapter 6 - Conclusion**
  This chapter summarizes the findings of the research, drawing conclusions based on the performance evaluation. It discusses the advantages and limitations of the proposed implementation, provides insights into

the practical implications of the work, and suggests directions for future research to further optimize CRYSTALS-Kyber on SoC-FPGA platforms.

2

BACKGROUND AND RELATED WORK

2.1. OVERVIEW OF POST-QUANTUM CRYPTOGRAPHY AND CRYSTALS-KYBER

Quantum computing presents a significant threat to traditional encryption systems, as it can efficiently solve problems such as integer factorization and discrete logarithms, which are the foundational assumptions of RSA and ECC. Algorithms like Shor's can break these classical cryptographic systems, rendering them insecure in a quantum-enabled future. This risk has led to the emergence of Post-Quantum Cryptography (PQC), which aims to develop algorithms resistant to attacks from both classical and quantum computers.

PQC algorithms are based on mathematical problems that are believed to be hard even for quantum computers. Among the most promising approaches are lattice-based, code-based, multivariate, and hash-based cryptographic systems. Among these, lattice-based cryptography has gained considerable attention for its strong theoretical foundations, security guarantees, and practical efficiency. One of the leading lattice-based algorithms is CRYSTALS-Kyber, a key encapsulation mechanism (KEM) selected by NIST for post-quantum cryptography standardization.

Lattice-based cryptography relies on the difficulty of solving problems in high-dimensional lattices, which remains hard even for quantum computers. CRYSTALS-Kyber is an asymmetric encryption scheme whose security is based on the hardness of the Module Learning With Errors (MLWE) problem, an extension of the Learning With Errors (LWE) problem. The LWE problem, in particular, involves solving a system of noisy linear equations, and its hardness

guarantees the security of the Kyber encryption scheme against both classical and quantum attacks. CRYSTALS-Kyber is particularly focused on providing secure and efficient public-key encryption and key exchange mechanisms, suitable for a wide range of applications, from secure communication to long-term data protection in a quantum-enabled future.

The theory of CRYSTALS-Kyber is primarily presented by Avanzi et al. in their study [1] and on the official CRYSTALS-Kyber website, which provides updates on the latest versions. The encryption scheme is structured into two primary components: Kyber.CPAPKE for public-key encryption and Kyber.CCAKEM for key encapsulation. Both processes involve heavy polynomial multiplications within finite fields, requiring efficient algorithms to perform these operations.

2.2. CRYSTALS-KYBER ALGORITHM

2.2.1. KEY FEATURES OF CRYSTALS-KYBER

CRYSTALS-Kyber is designed to meet the requirements of post-quantum cryptography, including:

- Security: Resists both classical and quantum attacks, providing strong guarantees based on MLWE.
- Efficiency: Offers fast encryption and decryption operations, suitable for resource-constrained environments.
- Simplicity: Features a modular design that is easy to implement in hardware and software.

Kyber is parameterized into three different levels of security, each with varying levels of efficiency. Kyber512 provides 128 bits of security, Kyber768 provides 192 bits, and Kyber1024 delivers 256 bits. Table 2.1 summarizes the core parameters used in the CRYSTALS-Kyber Key Encapsulation Mechanism (KEM) across its three security levels. These include polynomial dimensions, modulus values, noise parameters, compression levels, and failure probabilities, all of which influence the algorithm's security and efficiency trade-offs.

Version	n	k	q	η_1 , η_2	du , dv	Failure Probability (δ)
Kyber-512	256	2	3329	3 , 2	10 , 4	2^{-139}
Kyber-768	256	3	3329	2 , 2	10 , 4	2^{-164}
Kyber-1024	256	4	3329	2 , 2	11 , 5	2^{-174}

Table 2.1: Parameters of CRYSTALS-Kyber KEM

2.2.2. CORE COMPONENTS OF CRYSTALS-KYBER

1. **Kyber.CPAPKE**:

   Kyber's Chosen-Plaintext Attack-secure Public-Key Encryption (IND-CPA) scheme serves as the foundation for its key encapsulation mechanism. Its operations include:

   - Key Generation: Produces a public-private key pair based on MLWE. The keys are matrices or polynomials sampled from specific distributions in the ring
   - Encryption: Encodes a plaintext into a ciphertext using the public key, adding controlled noise to ensure security.
   - Decryption: Recovers the plaintext from the ciphertext using the private key. The correctness of decryption depends on the noise being small enough to avoid errors.

   This scheme is designed to resist quantum adversaries by relying on the hardness of MLWE, coupled with small random errors.

2. **Kyber.CCAKEM**:

   To enhance security, Kyber employs the Fujisaki-Okamoto transform to convert the CPA-secure Kyber.CPAPKE scheme into a Chosen-Ciphertext Attack-secure Key Encapsulation Mechanism (IND-CCA2). This ensures robust security even in scenarios where attackers have access to ciphertext decryption mechanisms.

2.2.3. FOUNDATIONAL CONCEPTS AND NOTATION IN CRYSTALS-KYBER

To understand the inner workings of the Kyber algorithm, it is essential to establish a common language and define certain mathematical constructs. The following notation and concepts are fundamental:

1. **Bytes and Byte Arrays**

   - B: Represents the set of all possible byte values (8-bit unsigned integers), i.e., $\{0, 1, \ldots, 255\}$.
   - B^k: Denotes the set of all byte arrays having a length of k.
   - B^*: Represents the set of byte arrays with arbitrary lengths (or byte streams).
   - $\|$: This symbol is used to denote the concatenation of two byte arrays.

- $a+k$: For a byte array a, this denotes the sub-array starting from the k-th byte (zero-based indexing).

2. **Polynomial Rings**

- R: Denotes the polynomial ring $\mathbb{Z}[X]/(X^n+1)$, where $\mathbb{Z}$ is the set of integers.
- R_q: Represents the polynomial ring $\mathbb{Z}_q[X]/(X^n+1)$, where $\mathbb{Z}_q$ is the set of integers modulo q. The sources consistently use $n = 256$ and $q = 3329$.

3. **Vectors and Matrices**

- Lowercase bold letters (e.g., $\mathbf{v}$) symbolize vectors with coefficients in R or R_q.
- Uppercase bold letters (e.g., $\mathbf{A}$) represent matrices.
- $\mathbf{v}^T$ (or $\mathbf{A}^T$) denotes the transpose of a vector $\mathbf{v}$ (or a matrix $\mathbf{A}$).
- $\mathbf{v}[i]$: Accesses the i-th element of the vector $\mathbf{v}$ (zero-based indexing).
- $\mathbf{A}[i][j]$: Refers to the element at the i-th row and j-th column of the matrix $\mathbf{A}$.

4. **Norms**

- For $w \in \mathbb{Z}_q$: $\|w\|_\infty = |w \mod \pm q|$.
- For a polynomial $w = w_0 + w_1 X + \cdots + w_{n-1} X^{n-1} \in R$:

$$\|w\|_\infty = \max_i \|w_i\|_\infty,$$
$$\|w\| = \sqrt{\|w_0\|_\infty^2 + \cdots + \|w_{n-1}\|_\infty^2}.$$

- These norms are extended similarly to vectors in R^k and R_q^k.

5. **Sets and Distributions**

- $s \leftarrow S$: Indicates that s is sampled uniformly at random from the set S.
- $s \leftarrow D$: Denotes that s is sampled according to the probability distribution D.

6. **Uniform Sampling in R_q**

Kyber uses a deterministic approach to sample elements in R_q that are statistically close to a uniformly random distribution. This is achieved

using a function Parse : $B^* \to R_q$, which converts a byte stream $B = \{b_0, b_1, b_2, \ldots\} \in B^*$ into the NTT-representation $\hat{a} \in R_q$ of $a \in R_q$. It works as follows:

1. Initialize $i \leftarrow 0$, $j \leftarrow 0$.

2. While $j < n$:

$$d_1 = b_i + 256 \cdot (b_{i+1} \mod 16)$$

$$d_2 = \lfloor b_{i+1}/16 \rfloor + 16 \cdot b_{i+2}$$

If $d_1 < q$, set:

$$\hat{a}_j = d_1, \quad j \leftarrow j + 1$$

If $d_2 < q$ and $j < n$, set:

$$\hat{a}_j = d_2, \quad j \leftarrow j + 1$$

Update $i \leftarrow i + 3$.

3. Finally, return:

$$\hat{a}(X) = \hat{a}_0 + \hat{a}_1 X + \hat{a}_2 X^2 + \cdots + \hat{a}_{n-1} X^{n-1}.$$

The intuition behind the function Parse is that if the input byte array is statistically close to a uniformly random byte array, then the output polynomial is statistically close to a uniformly random element of R_q.

7. **Sampling from a Centered Binomial Distribution**

The noise in Kyber is sampled from a centered binomial distribution B_η with parameter $\eta = 2$ or $\eta = 3$.

- Sample $(a_1, a_2, \ldots, a_\eta, b_1, b_2, \ldots, b_\eta)$ uniformly from $\{0,1\}^{2\eta}$.
- The output is:

$$X = \sum_{i=1}^{\eta} (a_i - b_i).$$

When a polynomial $f \in R_q$ (a ring of polynomials modulo $X^n + 1$) is sampled from B_η, it means each coefficient of f is sampled independently from B_η.

The polynomial f sampled from B_η can be represented as:

$$f(X) = \sum_{i=0}^{255} f_i X^i,$$

where each coefficient f_i is defined as:

$$f_i = \sum_{j=0}^{\eta-1} \beta_{2i\eta+j} - \sum_{j=0}^{\eta-1} \beta_{2i\eta+\eta+j}.$$

Here:

- $(\beta_0, \beta_1, \ldots, \beta_{512\eta-1})$ are the bits derived from the input byte array B.
- $f(X)$ is a polynomial with $n = 256$ coefficients in R_q.

8. **Compression and Decompression Functions**

The functions Compress_q and Decompress_q are defined to facilitate the reduction of ciphertext size by discarding low-order bits without significantly impacting the accuracy of the representation.

Given an integer $x \in \mathbb{Z}_q$, where q is a prime modulus, these functions output an integer in the range $\{0, \ldots, 2^d - 1\}$ with $d < \lceil \log_2(q) \rceil$.

The formal definitions of Compress_q and Decompress_q are as follows:

$$\text{Compress}_q(x, d) = \left\lfloor \frac{2^d}{q} \cdot x \right\rfloor \bmod 2^d$$

$$\text{Decompress}_q(x, d) = \left\lfloor \frac{q}{2^d} \cdot x \right\rfloor$$

These functions are designed to satisfy the following property:

$$x' = \text{Decompress}_q\Big(\text{Compress}_q(x, d), d\Big)$$

where x' is an element close to x. Specifically,

$$|x' - x \bmod \pm q| \leq B_q := \left\lfloor \frac{q}{2^{d+1}} \right\rfloor$$

Here, B_q represents a bound on the error introduced by the compression and decompression process. This error tolerance allows for a balance between compression efficiency and accuracy.

Apart from compression, these functions are also used to perform the standard Learning With Errors (LWE) error correction during encryption and decryption.

9. **Symmetric Primitives (Hash functions)**

   - **PRF**: A pseudorandom function, PRF : $B^{32} \times B \to B^*$, used for key derivation and noise generation.
   - **XOF**: An extendable output function, XOF : $B^* \times B \times B \to B^*$, used for expanding the public matrix **A** from a seed.
   - **H**: A hash function, H : $B^* \to B^{32}$.
   - **G**: A hash function, G : $B^* \to B^{32} \times B^{32}$.
   - **KDF**: A key derivation function, KDF : $B^* \to B^*$.

10. **NTT (Number Theoretic Transform)**

    A very efficient way to perform multiplications in R_q is via the so-called number-theoretic transform (NTT). For the prime $q = 3329$ with $q - 1 = 2^8 \cdot 13$, the base field $\mathbb{Z}_q$ contains primitive 256-th roots of unity but not primitive 512-th roots. Therefore, the defining polynomial $X^{256} + 1$ of R factors into 128 polynomials of degree 2 modulo q, and the NTT of a polynomial $f \in R_q$ is a vector of 128 polynomials of degree 1.

    Simple in-place implementations of the NTT without reordering output these polynomials in bit-reversed order. We define the NTT in this way. Concretely, let $\zeta = 17$ be the first primitive 256-th root of unity modulo q, and $\{\zeta, \zeta^3, \zeta^5, \ldots, \zeta^{255}\}$ the set of all 256-th roots of unity. The polynomial $X^{256} + 1$ can therefore be written as

$$X^{256} + 1 = \prod_{i=0}^{127} (X^2 - \zeta^{2i+1}) = \prod_{i=0}^{127} (X^2 - \zeta^{2\mathrm{br}_7(i)+1}),$$

    where $\mathrm{br}_7(i)$ for $i = 0, \ldots, 127$ is the bit-reversal of the unsigned 7-bit integer i. This latter ordering of the factors is useful for compatibility with the idiosyncrasies of AVX instructions. Then the NTT of $f \in R_q$ is given by

$$(f \mod X^2 - \zeta^{2\mathrm{br}_7(0)+1}, \ldots, f \mod X^2 - \zeta^{2\mathrm{br}_7(127)+1}).$$

    This vector of linear polynomials is serialized to a vector in $\mathbb{Z}_q^{256}$ in the canonical way. Moreover, in order to facilitate in-place implementations of the NTT, we define

$$\mathrm{NTT}(f) = \hat{f} = \hat{f}_0 + \hat{f}_1 X + \cdots + \hat{f}_{255} X^{255},$$

where

$$\hat{f}_{2i} = \sum_{j=0}^{127} f_{2j}\zeta^{(2\mathrm{br}_7(i)+1)j},$$
$$\hat{f}_{2i+1} = \sum_{j=0}^{127} f_{2j+1}\zeta^{(2\mathrm{br}_7(i)+1)j}.$$

Although $\hat{f}$ is written as a polynomial in R_q, it has no algebraic meaning as such. The natural algebraic representation of $\mathrm{NTT}(f) = \hat{f}$ is as 128 polynomials of degree 1 as in the vector form above.

Using NTT and its inverse NTT^{-1}, the product $f \cdot g$ of two elements $f, g \in R_q$ can be computed efficiently as

$$\mathrm{NTT}^{-1}(\mathrm{NTT}(f) \circ \mathrm{NTT}(g)),$$

where $\circ$ denotes the component-wise multiplication consisting of 128 basecase multiplications:

$$\hat{h}_{2i} + \hat{h}_{2i+1}X = (\hat{f}_{2i} + \hat{f}_{2i+1}X)(\hat{g}_{2i} + \hat{g}_{2i+1}X) \mod X^2 - \zeta^{2\mathrm{br}_7(i)+1}.$$

11. **Encoding and Decoding**

Kyber requires two data types to be serialized into byte arrays: byte arrays themselves and (vectors of) polynomials. The function Decode_ℓ deserializes a byte array of 32ℓ bytes into a polynomial

$$f = f_0 + f_1 X + \cdots + f_{255}X^{255},$$

where each coefficient $f_i \in \{0, \ldots, 2^\ell - 1\}$. The function Encode_ℓ is defined as the inverse of Decode_ℓ.

2.2.4. CRYSTALS-KYBER OPERATIONS

Figure 2.1 illustrates the key generation, encryption, and decryption mechanisms of Kyber in a client-server model, assuming Alice and Bob perform the encryption communication.

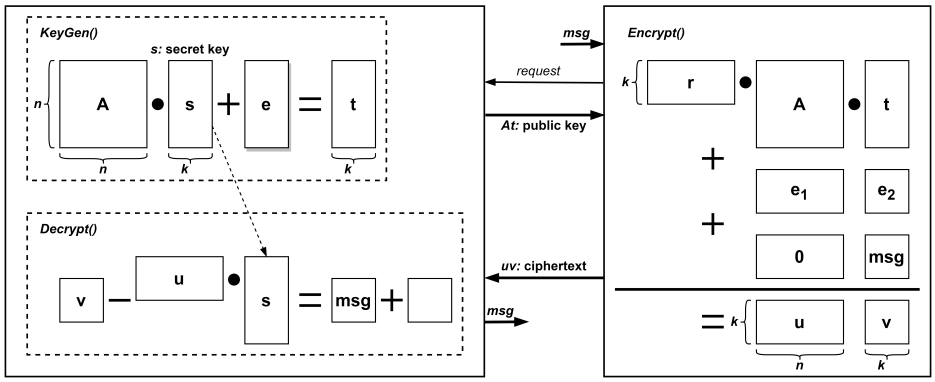

Figure 2.1: Kyber cryptography process

In a typical client-server communication scenario utilizing CRYSTALS-Kyber, Bob (the client) initiates the process by sending a request to Alice (the server). In response, Alice generates a cryptographic key pair consisting of a public key and a secret key. The public key is then shared with Bob, while Alice securely retains the secret key. Bob uses the received public key to encrypt a plaintext message, resulting in a ciphertext, which he sends back to Alice. Upon receiving the ciphertext, Alice employs her secret key to decrypt it, successfully recovering the original plaintext message. This secure workflow underscores the robust encryption and decryption capabilities of the Kyber algorithm, showcasing the essential roles of public and private keys in facilitating confidentiality and data integrity.

The core operations of CRYSTALS-Kyber are structured into three main phases:

KEY GENERATION (KEYGEN)

A fundamental aspect of the process involves generating a matrix A through uniform sampling in the field $\mathbb{Z}_q$. The noise matrix e and matrix s are sampled from a binomial distribution in the field $\mathbb{Z}_q$. It is easy to see that when given the matrix A and matrix t, finding the matrix s is straightforward by multiplying the inverse of A with t: $s = A^{-1} \circ t$. However, with the Learning With Errors (LWE) problem, adding the noise matrix e makes it very difficult to precisely find the matrix s when only A and t are known. Here, A and t can be understood as the public key, while s is the secret key.

Kyber's CPA-secure encryption scheme heavily relies on the computation of the term $As + e$, where the most resource-intensive operation is the polynomial multiplication involved. To optimize this, Kyber employs the Number-Theoretic Transform (NTT) and its inverse (INTT), which significantly accelerate polynomial operations while reducing computational overhead.

Algorithm 1 shows the Kyber.CCAKEM.KeyGen() algorithm, where the algorithm Kyber.CPAPKE.KeyGen() is called to generate a key pair that includes a public key pk and a secret key sk.

Algorithm 1 Kyber.CCAKEM.KeyGen()

Output: Public key $pk \in B^{12 \cdot k \cdot n/8+32}$
Output: Secret key $sk \in B^{24 \cdot k \cdot n/8+96}$
1: $z \leftarrow B^{32}$
2: $(pk, sk_0) :=$ `Kyber.CPAPKE.KeyGen()`
3: $sk \leftarrow (sk_0 \| pk \| kH(pk) \| z)$
4: **Return** (pk, sk)

The final public key pk is the one generated from Kyber.CPAPKE.KeyGen(), while the final secret key is composed of the secret key sk from Kyber.CPAPKE.KeyGen(), concatenated with the public key pk and the result of $H(pk)$, and ending with z, which is generated randomly. The function H, corresponds to the SHA3-256 function.

The Kyber.CPAPKE.KeyGen() algorithm is describe in the algorithm 2

Algorithm 2 Kyber.CPAPKE.KeyGen(): Key Generation

Output: Secret key $sk \in B^{12 \cdot k \cdot n/8}$
Output: Public key $pk \in B^{12 \cdot k \cdot n/8+32}$
$d \leftarrow B^{32}$
$(\rho, \sigma) \leftarrow G(d)$
$N \leftarrow 0$
for each i from 0 to $k-1$ **do** ▷ Generate matrix $\hat{A} \in R_q^{k \times k}$ in NTT domain
  for each j from 0 to $k-1$ **do**
    $\hat{A}[i][j] \leftarrow \text{Parse}(\text{XOF}(\rho, j, i))$
  end for
end for
for each i from 0 to $k-1$ **do** ▷ Sample $s \in R_q^k$ from B_{η_1}
  $s[i] \leftarrow \text{CBD}_{\eta_1}(\text{PRF}(\sigma, N))$
  $N \leftarrow N+1$
end for
for each i from 0 to $k-1$ **do** ▷ Sample $e \in R_q^k$ from B_{η_1}
  $e[i] \leftarrow \text{CBD}_{\eta_1}(\text{PRF}(\sigma, N))$
  $N \leftarrow N+1$
end for
$\hat{s} \leftarrow \text{NTT}(s)$
$\hat{e} \leftarrow \text{NTT}(e)$
$\hat{t} \leftarrow \hat{A} \circ \hat{s} + \hat{e}$
$pk \leftarrow (\text{Encode}_{12}(\hat{t} \mod +q) \| \rho)$ ▷ $pk := A \circ s + e$
$sk \leftarrow \text{Encode}_{12}(\hat{s} \mod +q)$ ▷ $sk := s$
Return (pk, sk)

The process begins by generating the matrix A directly in the NTT domain through uniform sampling (Uniform Sampling-Parse). The vectors s and e are generated using binomial distribution sampling. This is achieved by the function CBD_η (for "centered binomial distribution").

Once generated, these vectors are moved to the NTT domain, where the vector s, after being transformed into the NTT domain, becomes the secret key sk. The public key pk is generated by computing the polynomial $\hat{t}$, concatenated with a randomly generated bit string ρ, where $\hat{t} = \hat{A} \cdot \hat{s} + \hat{e}$, and the symbols with hats represent variables in the NTT domain.

In Kyber, an effective way to handle polynomial multiplication is through the Number Theoretic Transform (NTT). This method is used to accelerate polynomial multiplication and is analogous to the Fast Fourier Transform (FFT), but

operates in modular arithmetic.

ENCRYPTION (ENCRYPT)

The algorithm 3 illustrates the Kyber.CCAKEM.Enc(pk) algorithm, which is the encryption process in Kyber, uses the public key pk to generate the ciphertext c and the shared key K.

Algorithm 3 Kyber.CCAKEM.Enc(pk)

Input: Public key $pk \in B^{12 \cdot k \cdot n/8+32}$
Output: Ciphertext $c \in B^{du \cdot k \cdot n/8+dv \cdot n/8}$
Output: Shared key $K \in B^*$
1: $m \leftarrow B^{32}$
2: $m \leftarrow H(m)$ ▷ Do not send output of system RNG
3: $(K, \bar{r}) := G(m \| kH(pk))$
4: $c := \texttt{Kyber.CPAPKE.Enc}(pk, m, r)$
5: $K := \texttt{KDF}(\bar{K} \| H(c))$
6: **Return** (c, K)

The encryption process (as depicted in the algorithm 3) begins by generating a random message m, and then hashing m to obtain its hash. The result of $H(pk)$ is input into the hash function G along with the message m to generate $\bar{K}$ and r for the encryption process, through the Kyber.CPAPKE.Enc(pk, m, r) as presented in the algorithm 4, resulting in the ciphertext c. Finally, the shared key K is computed from the result of the Key Derivation Function (KDF) function, which corresponding to the SHAKE256 hash function.

Algorithm 4 Kyber.CPAPKE.Enc(pk, m, r): Encryption

Input: Public key $pk \in B^{12 \cdot k \cdot n/8+32}$
Input: Message $m \in B^{32}$
Input: Random coins $r \in B^{32}$
Output: Ciphertext $c \in B^{d_u \cdot k \cdot n/8 + d_v \cdot n/8}$
$N \leftarrow 0$
$\hat{t} \leftarrow \text{Decode}_{12}(pk)$
$\rho \leftarrow pk + 12 \cdot k \cdot n/8$
for each i from 0 to $k-1$ **do** ▷ Generate matrix $\hat{A} \in R_q^{k \times k}$ in NTT domain
    for each j from 0 to $k-1$ **do**
        $\hat{A}^T[i][j] \leftarrow \text{Parse}(\text{XOF}(\rho, i, j))$
    end for
end for
for each i from 0 to $k-1$ **do** ▷ Sample $r \in R_q^k$ from B_{η_1}
    $r[i] \leftarrow \text{CBD}_{\eta_1}(\text{PRF}(r, N))$
    $N \leftarrow N+1$
end for
for each i from 0 to $k-1$ **do** ▷ Sample $e_1 \in R_q^k$ from B_{η_2}
    $e_1[i] \leftarrow \text{CBD}_{\eta_2}(\text{PRF}(r, N))$
    $N \leftarrow N+1$
end for
$e_2 \leftarrow \text{CBD}_{\eta_2}(\text{PRF}(r, N))$ ▷ Sample $e_2 \in R_q$ from B_{η_2}
$\hat{r} \leftarrow \text{NTT}(r)$
$u \leftarrow \text{NTT}^{-1}(\hat{A}^T \circ \hat{r}) + e_1$
$v \leftarrow \text{NTT}^{-1}(\hat{t}^T \circ \hat{r}) + e_2 + \text{Decompress}_q(\text{Decode}_1(m), 1)$
$c_1 \leftarrow \text{Encode}_{d_u}(\text{Compress}_q(u, d_u))$
$c_2 \leftarrow \text{Encode}_{d_v}(\text{Compress}_q(v, d_v))$
Return $c = (c_1 \| c_2)$ ▷ $c := (\text{Compress}_q(u, d_u), \text{Compress}_q(v, d_v))$

By using theKyber.CPAPKE.Enc(pk, m, r), the matrix $\hat{A}^T$ is generated in the NTT domain through uniform sampling. The vectors r, e_1, and e_2 are generated using the centered binomial distribution. Then, the vector r and e_1 are passed into the NTT domain to compute the vectors u and v:

$$u := A^T \circ r + e_1$$

$$v := t^T \circ r + e_2 + \text{Decompress}_q(m, 1)$$

DECRYPTION (DECRYPT)

Algorithm 5 explains the Kyber.CCAKEM.Dec(c, sk) algorithm, which performs decryption and key sharing based on the ciphertext c and the secret key sk.

Algorithm 5 Kyber.CCAKEM.Dec(c, sk)

Input: Ciphertext $c \in B^{du \cdot k \cdot n/8 + dv \cdot n/8}$
Input: Secret key $sk \in B^{24 \cdot k \cdot n/8 + 96}$
Output: Shared key $K \in B^*$
$pk := sk[12 \cdot k \cdot n/8]$
$h := sk[24 \cdot k \cdot n/8 + 32] \in B^{32}$
$z := sk[24 \cdot k \cdot n/8 + 64]$
$m_0 := \texttt{Kyber.CPAPKE.Dec}(sk, c)$
$(\bar{K}_0, r_0) := G(m_0 \| h)$ ▷ Concatenate using $\|$
$c_0 := \texttt{Kyber.CPAPKE.Enc}(pk, m_0, r_0)$
if $c = c_0$ **then**
    Return $K := \texttt{KDF}(\bar{K}_0 \| H(c))$
else
    Return $K := \texttt{KDF}(z \| H(c))$
end if

First, Kyber extracts components from the secret key sk to obtain the public key pk, h, and z. Then, Kyber decrypts using the algorithm Kyber.CPAPKE.Dec(s, (u, v)), where the secret key s excludes the public key pk, h, and z, and the ciphertext $c = (u, v)$ is used to recover the message m_0. After recovering the message m_0, Kyber re-encrypts m_0 to obtain a new ciphertext c_0, which ensures that the message m_0 is correct, as the encryption process has been outlined earlier.

The Kyber.CPAPKE.Dec(s, (u, v)) algorithm 6 is presented as below:

Algorithm 6 Kyber.CPAPKE.Dec(sk, c): Decryption

Input: Secret key $sk \in B^{12 \cdot k \cdot n/8}$
Input: Ciphertext $c \in B^{d_u \cdot k \cdot n/8 + d_v \cdot n/8}$
Output: Message $m \in B^{32}$
$u \leftarrow \text{Decompress}_q(\text{Decode}_{d_u}(c), d_u)$
$v \leftarrow \text{Decompress}_q(\text{Decode}_{d_v}(c + d_u \cdot k \cdot n/8), d_v)$
$\hat{s} \leftarrow \text{Decode}_{12}(sk)$
$m \leftarrow \text{Encode}_1(\text{Compress}_q(v - \text{NTT}^{-1}(\hat{s}^T \circ \text{NTT}(u)), 1))$
Return m

In this step, Kyber decompresses the components from the ciphertext c to recover vectors u and v. Then, the secret key s is decoded, and the result is assigned to vector $\hat{s}$, where $\hat{s} = \text{Decode}_{12}(s)$. Finally, the message m' is computed as $m_0 = \text{Encode}_1(\text{Compress}_q(v - \text{NTT}^{-1}(\hat{s}^T \cdot \text{NTT}(u)), 1))$, and the message m_0 is returned. Note that $m := \text{Compress}_q(v - s^T \circ u, 1)$

2.3. NUMBER THEORETIC TRANSFORM (NTT)

Kyber's security level is based on the hardness of solving the Learning-With-Errors (LWE) problem in lattice-based cryptography. While this enhances security, Kyber faces challenges due to the inherent complexity of polynomial multiplication, which has a time complexity of $O(n^2)$ with n coefficients. To address this, the use of the Number Theoretic Transform (NTT), an effective method for computing polynomial multiplication, has been proposed. NTT is a generalization of the Fast Fourier Transform (FFT) in the finite field and reduces the computation time and complexity to $O(n \log n)$, much faster than the schoolbook multiplication method mentioned. While NTT significantly improves the efficiency of polynomial multiplication, implementing an efficient NTT hardware architecture poses challenges, as it requires a careful balance between execution time and hardware area.

2.3.1. CLASSICAL NTT/INTT

To represent polynomials in computing, it is common to use the coefficient representation, where the coefficients of each polynomial term are stored as an array. For example, a polynomial of the form:

$$C = 2 + 3x + 4x^2 \rightarrow C = [2, 3, 4]$$

Another representation is the value representation, in which a polynomial of degree d is represented by $d+1$ points sampled from the polynomial's curve. To enable fast polynomial multiplication and transformations, the Fast Fourier Transform (FFT) and its inverse (IFFT) are often used [17].

The FFT and IFFT have a complexity of $O(N \log N)$, which is a significant improvement over the typical $O(N^2)$ complexity of the Discrete Fourier Transform (DFT) [1; 18].

In modular arithmetic, specifically with rings like the ones used in Kyber over Z_q, the Number Theoretic Transform (NTT) and its Inverse (INTT or NTT^{-1}) are applied to switch between coefficient and value representations of polynomials [1].

The relationship between NTT and INTT for polynomial multiplication

$h = f \cdot g$ can be expressed as:

$$h = \text{NTT}^{-1}(\text{NTT}(f) \cdot \text{NTT}(g))$$

2

The forward NTT can be defined as follows [12]: The forward NTT can be defined as follows [12]:

$$\hat{a}_i = \text{NTT}_q(a_i) = \sum_{j=0}^{n-1} a_j \omega_n^{ij} \mod q$$

$$\text{for } i = 0, 1, \ldots, n-1.$$

Similarly, the inverse NTT (INTT) is defined as [12]:

$$a_i = \text{INTT}_q(\hat{a}_i) = n^{-1} \sum_{j=0}^{n-1} \hat{a}_j \omega_n^{-ij} \mod q$$

$$\text{for } i = 0, 1, \ldots, n-1.$$

where ω_n is a primitive root of unity.
To implement NTT in hardware, **Butterfly Units (BU)**, a computation method within FFT, are used. The NTT is executed using the **Cooley-Tukey (CT) Butterfly Unit** while the INTT uses the **Gentleman-Sande (GS) Butterfly Unit**. These units are efficient for designing NTT hardware.

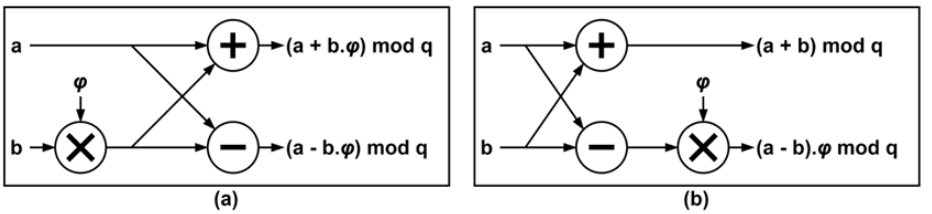

Figure 2.2: Two butterfly units in NTT/INTT: (a) Cooley-Tuckey (CT) butterfly. (b) Gentleman-Sande (GS) butterfly

2.3.2. NEGATIVE WRAPPED CONVOLUTION

Kyber operates with $n = 256$. This requires computing the NTT for polynomials of degree up to 256 in the $n = 256$ ring or 512 in the $n = 512$ ring. To facilitate this, the Negative Wrapped Convolution (NWC) approach is used [12].

According to convolution theory [12], polynomial multiplication can be performed by:

$$\text{INTT}_{2n}(\text{NTT}_{2n}(\text{zeropadding}(a)) \odot \text{NTT}_{2n}(\text{zeropadding}(b)))$$

where $\odot$ represents point-wise multiplication, and the function zeropadding(a) expands the length of a from n to $2n$ by appending zeros. Consequently, the multiplication within the ring $\mathbb{Z}_q[x]/\langle f(x)\rangle$ can be executed with three $2n$-point NTT/INTT operations followed by reduction using the modular polynomial $f(x)$.

When $f(x)$ is defined as the polynomial $x^n - 1$, polynomial multiplication over $\mathbb{Z}_q[x]/\langle f(x)\rangle$ can be more efficiently implemented using positive wrapped convolution. This approach requires only three n-point NTT/INTT operations without increasing the size of the transforms to $2n$. Reduction with $f(x)$ is automatically handled, as shown below:

$$c = \text{INTT}_n(\text{NTT}_n(a) \odot \text{NTT}_n(b))$$

When $f(x)$ is adjusted to $x^n + 1$, Negative Wrapped Convolution (NWC) enables multiplication over $\mathbb{Z}_q[x]/\langle x^n + 1\rangle$ with three n-point NTT/INTT calculations. This variant does not necessitate explicit modular reduction but requires the prime q to satisfy $q \equiv 1 \pmod{2n}$, allowing for the existence of ω_n and its square root γ_{2n}. However, NWC introduces pre-processing before NTT and post-processing after INTT, defined as follows: let $\bar{a}_i = a_i\gamma_{2n}^i$, $\bar{b}_i = b_i\gamma_{2n}^i$, and $\bar{c}_i = c_i\gamma_{2n}^{-i}$. The resulting product $c = a \cdot b$ over $\mathbb{Z}_q[x]/\langle x^n + 1\rangle$ using NWC is computed as:

$$\bar{c} = \text{INTT}_n(\text{NTT}_n(\bar{a}) \odot \text{NTT}_n(\bar{b}))$$

In this scheme, the classic n-point NTT is applied to the scaled vectors $\bar{a}$ and $\bar{b}$. After the classic n-point INTT, the scaled vector $\bar{c}$ is obtained, from which the final result c is reconstructed by computing $c_i = \bar{c}_i\gamma_{2n}^{-i}$. This method avoids both size doubling and explicit reduction, albeit requiring coefficient-wise scaling by γ_{2n} before NTT and by γ_{2n}^{-1} after INTT.

In this paper, **pre-processing** denotes the coefficient-wise multiplications of a_i and b_i by γ_{2n}^i before NTT, while **post-processing** involves scaling by γ_{2n}^{-i} after INTT. Note that INTT typically uses FFT, including a final scaling by n^{-1}. Therefore, post-processing in this case refers to the final scaling by n^{-1} in the classic INTT, along with coefficient-wise scaling by γ_{2n}^{-i}.

2.3.3. PROPERTIES OF THE TWIDDLE FACTOR

If n is a power of two, the twiddle factors of NTT exhibit three significant properties that are similarly found in FFT twiddle factors:

$$\begin{aligned}
&\text{symmetry property:} \quad \omega_n^{k+N/2} = -\omega_n^k,\\
&\text{periodicity property:} \quad \omega_n^{k+N} = \omega_n^k,\\
&\text{complex conjugate symmetry:} \quad \omega_n^{n-k} = \omega_n^{-k},
\end{aligned}$$

where ω_n represents the n-th primitive root of unity. If n is a power of two, then $\gamma_{2n} = \sqrt{\omega_n}$, making it straightforward to verify these properties.

2.3.4. OPTIMIZED NTT/INTT VERSION

For Kyber, the integer q does not satisfy $q \equiv 1 \mod 2n$, and thus Kyber operates within the ring R_q as $Z_q[X]/(X^n+1)$ with $q = 3329$ and $n = 256$.

However, the NTT/INTT results of Kyber can still be computed using the NWC method. This involves decomposing a degree-n polynomial $f \in Z_q[X]/(X^n+1)$ using a coefficient representation of degree $n/2$ as $f_{\text{even}}(x^2)$ and $f_{\text{odd}}(x^2)$, where $f_{\text{even}}, f_{\text{odd}} \in Z_q[x^2]/((x^2)^{n/2}+1)$. Thus, $f(x)$ can be recovered by rearranging the coefficients as follows:

$$f(x) = f_{\text{even}}(x^2) + x \cdot f_{\text{odd}}(x^2)$$

In this way, the NTT/INTT can be calculated with NWC using two functions, f_{even} and f_{odd}, with a smaller system $n' = n/2 = 256/2 = 128$. Hence, the new definition is $n = 128$. The NTT for Kyber can be computed as:

$$\text{NTT}(f(x)) = (\text{NTT}(f_{\text{even}}), \text{NTT}(f_{\text{odd}}))$$

and the INTT for Kyber as:

$$\text{INTT}(f(x)) = (\text{INTT}(f_{\text{even}}), \text{INTT}(f_{\text{odd}}))$$

When q satisfies $q \equiv 1 \mod 2n$, then Kyber operates within the ring R_q as $Z_q[X]/(X^n+1)$ with $q = 3329; n = 128$.

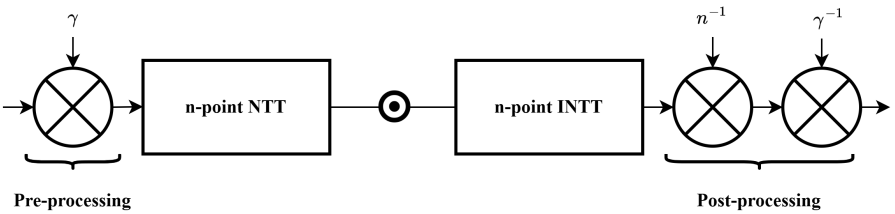

Figure 2.3: Negative Wrapped Convolution of NTT of INTT with Pre-processing and Post-processing

Figure 2.3 illustrates the Negative Wrapped Convolution (NWC) using Number Theoretic Transform (NTT) and its inverse (INTT), including the necessary pre-processing and post-processing steps. This structure enables efficient polynomial multiplication in Kyber through modular arithmetic in the NTT domain.

In Kyber's design, the following equations define the NWC NTT:

$$\hat{a}_i = \mathrm{NTT}_n(a_i) = \sum_{j=0}^{n-1} a_j \omega_n^{ij} \gamma_{2n}^{j} \mod q$$

and for INTT:

$$a_i = \mathrm{INTT}_n(\hat{a}_i) = n^{-1} \gamma_{2n}^{-j} \sum_{j=0}^{N-1} \hat{a}_j \omega_n^{-ij} \gamma_{2n}^{j} \mod q$$

$$\text{for } i = 0, 1, \ldots, n-1.$$

A further optimization method for the NTT algorithm is the low complexity NTT/INTT approach [12]. The complexity of the NTT using NWC (Negative Wrapped Convolution) is $O(\frac{N}{2}\log N + N)$. The $\frac{N}{2}\log N$ factor denotes the complexity of computing the NTT, where $\gamma_{2n} = \sqrt{\omega_n}$ is utilized.

Applying the equation 2.1 in [12], We have:

$$\omega_m^j \gamma_{2m} \equiv \gamma_{2m}^{2j+1} \equiv \gamma_{2N}^{\frac{(2j+1)n}{m}} \tag{2.1}$$

$$m = 2^1, 2^2, \ldots, n; \quad j = 0, 1, \ldots, \frac{m}{2} - 1$$

Applying this approach reduces both the pre-processing and the complexity for the NTT to $O(\frac{N}{2}\log N)$.

Similarly, we want to remove γ^{-1} from INTT calculation. This also leads to the following transformation equation:

$$\omega_m^j \gamma_{2m}^{-1} \equiv \gamma_{2m}^{-(2j+1)} \equiv \gamma_{2N}^{-\frac{(2j+1)N}{m}} \tag{2.2}$$

$$m = 2^1, 2^2, \ldots, N; \quad j = 0, 1, \ldots, \frac{m}{2} - 1$$

The result from 2.2 reduces the complexity of the INTT operation to $O(\frac{N}{2}\log N + N)$.

Furthermore, to eliminate the post-processing step of multiplying by $\frac{1}{n}$, study [12] also proposed a method to halve the results from the BU Gentleman-Sande. This approach reduces the complexity of the INTT to $O(N \log N)$ when configured with the Gentleman-Sande BU structure.

$$a_{2i} = \left(\frac{n}{2}\right)^{-1} \gamma_{2N}^{-i} \sum_{j=0}^{n/2-1} \hat{b}_j^{(0)} \omega_{n/2}^{-ij} \mod q \tag{2.3}$$

$$a_{2i+1} = \left(\frac{n}{2}\right)^{-1} \gamma_{2n}^{-i} \sum_{j=0}^{n/2-1} \hat{b}_j^{(1)} \omega_{n/2}^{-ij} \mod q \tag{2.4}$$

$$\text{With} \quad \hat{b}_j^{(0)} = \frac{\hat{a}_j + \hat{a}_{(j+n/2)}}{2} \mod q, \quad \hat{b}_j^{(1)} = \frac{\hat{a}_j - \hat{a}_{(j+n/2)}}{2} \omega_n^{-j} \gamma_{2n}^{-1}.$$

Equations 2.3 and 2.4 show that this is a structure similar to the NWC INTT above, but with a reduction to $n/2$ points instead of n points. Instead of performing the $1/n$ scaling at the end, we apply a division by 2 to each result of the Gentleman-Sande butterfly unit (BU GS) at each stage.

Algorithm 7 and Algorithm 8 implement low-complexity forward and inverse NTT operations, respectively, by applying Negative Wrapped Convolution (NWC) techniques.

Algorithm 7 Low complexity NTT operation with Cooley–Tukey butterfly

Input: polynomial $a = (a_0, a_1, \ldots, a_{n-1})$ as $a(x) \in \mathbb{Z}_q[X]/(X^n + 1)$, $\omega_n \in \mathbb{Z}_q$ is the n-th primitive root of unity, $n = 2^t$ and $\gamma_{2n} = \sqrt{\omega_n}$
Output: $\hat{a} = \text{NTT}(a)$

1: $\hat{a} :=$ bit-reverse(a)
2: **for** $i := 1 \rightarrow t$ **do**
3: $\quad m := 2^i$
4: $\quad$ **for** $j := 0 \rightarrow m/2 - 1$ **do**
5: $\quad\quad \omega := \gamma_{2n}^{(2j+1)N/m}$
6: $\quad\quad$ **for** $k := 0 \rightarrow N/m - 1$ **do**
7: $\quad\quad\quad u := a[km + j]$
8: $\quad\quad\quad t := \omega \cdot a[km + j + m/2] \mod q$
9: $\quad\quad\quad a[km + j] := u + t \mod q$
10: $\quad\quad\quad a[km + j + m/2] := u - t \mod q$
11: $\quad\quad$ **end for**
12: $\quad$ **end for**
13: **end for**
14: **return** $\hat{a}$

Algorithm 8 Low complexity INTT operation with Gentleman–Sande butterfly

Input: polynomial $\hat{a} = (\hat{a}_0, \hat{a}_1, \ldots, \hat{a}_{n-1})$ as $\hat{a}(x) \in \mathbb{Z}_q[X]/(X^n+1)$, $\omega_n \in \mathbb{Z}_q$ is the N-th primitive root of unity, $n = 2^t$ and $\gamma_{2n} = \sqrt{\omega_n}$
Output: $a = \text{INTT}(\hat{a})$

1: **for** $i := 1 \rightarrow t$ **do**
2: $m := 2^{t-i}$
3: **for** $j := 0 \rightarrow m/2 - i$ **do**
4: $\omega := \gamma_{2n}^{-(2j+1)N/m}$
5: **for** $k := 0 \rightarrow N/m - 1$ **do**
6: $u := a[km + j]$
7: $t := a[km + j + m/2] \mod q$
8: $a[km + j] := \frac{u+t}{2} \mod q$
9: $a[km + j + m/2] := \frac{u-t}{2}\omega \mod q$
10: **end for**
11: **end for**
12: **end for**
13: $\hat{a} :=$ bit-reverse$(\hat{a})$
14: **return** $\hat{a}$

2.4. MODULAR ARITHMETIC

NTT arithmetic requires the execution of numerous modular addition, subtraction, and multiplication operations. To achieve efficient modular arithmetic operations, we employ hardware-friendly constant-time modular arithmetic algorithms. Throughout this section, we will assume a K-bit modulus q. It is important to note that our modular arithmetic operations produce numbers within the range $[0, 2^K - 1]$, rather than $[0, q-1]$.

2.4.1. MODULAR ADDITION

Algorithm 9 illustrates a hardware-friendly, constant-time partial modular addition operation. Since the modulus $q = 3329$ is 12-bit, the input A and B should also be 12-bit integers and less than $2^{12} - 1$. The result of $A + B$ is a 13-bit integer with a maximum value that may exceed $2q$. This implies that at most two subtraction operations are required to reduce the result of the addition operation back to K bits. Consequently, in Algorithm 9, the result C is guaranteed to be a K-bit number. As demonstrated, Algorithm 9 is designed to operate at a constant time from a hardware perspective.

Algorithm 9 Modular Addition Algorithm

Input: A, B, q (K-bit positive integers)
Output: $C \equiv A + B \pmod{q}$ (K-bit positive integer)

```
1: T1 = A + B
2: T2 = T1 − q
3: T3 = T1 − 2 · q
4: if T2 < 0 then
5:     C = T1
6: else if T3 < 0 then
7:     C = T2
8: else
9:     C = T3
10: end if
11: return C
```

2.4.2. MODULAR SUBTRACTION

To improve efficiency, we utilize partial modular subtraction operations, rather than full modular subtraction. Algorithm 10 illustrates our proposed algorithm. Given 12-bit numbers A and B that are less than $2^{12} - 1$, the minimum value of $A - B$ will not be less than $-(2^{12} - 1)$, and the maximum value will not exceed $2^{12} - 1$. In the case of taking the minimum value, an addition of $2q$ is required. This implies that at most two addition operations are required to guarantee a positive result. Consequently, in Algorithm 10, the result C is a positive K-bit number. As demonstrated, Algorithm 10 is designed to operate at a constant time from a hardware perspective.

Algorithm 10 Modular Subtraction Algorithm

Input: A, B, q (K-bit positive integers)
Output: $C \equiv A - B \pmod{q}$ (K-bit positive integer)

1: $T1 = A - B$
2: $T2 = T1 + q$
3: $T3 = T1 + 2 \cdot q$
4: **if** $T2 < 0$ **then**
5: $\quad C = T3$
6: **else if** $T1 < 0$ **then**
7: $\quad C = T2$
8: **else**
9: $\quad C = T1$
10: **end if**
11: **return** C

2.4.3. MODULAR MULTIPLICATION

Within the entire NTT architecture, the most critical and time-consuming module is modular multiplication. Conventional techniques for modular reduction, such as Montgomery and traditional Barrett reductions, are widely adopted because they can be implemented using addition, bit-shift, and constant-coefficient multiplication operations, without involving costly division operations.

In this work, we adopt a customized low-complexity Barrett Reduction specifically optimized for CRYSTALS-Kyber, with modulus $q = 3329$. This approach enhances computational efficiency by breaking down the reduction process into simple shift and addition operations, making it highly suitable for hardware-accelerated and lightweight implementations.

The reduction method used minimizes the bit-width of intermediate values and avoids the need for iterative correction steps. The detailed reduction procedure is described in Algorithm 11.

Algorithm 11 The Low-Complexity Barrett Reduction for Kyber

Input: $c = a \cdot b \in [0, 2^{24}]$
Input: The truncation parameters ρ_1, ρ_2, σ
Output: $r \equiv c \mod 3329$
$c_1 \leftarrow c[23:8], \quad c_0 \leftarrow c[7:0]$
Segment 1. Calculate $Q_1' \approx \lfloor c_1/13 \rfloor$:
$d_1 \leftarrow \lfloor c_1 \cdot 2^{-\rho_1} \rfloor + \lfloor c_1 \cdot 2^{-(\rho_1+2)} \rfloor$
$d_2 \leftarrow \lfloor d_1 \cdot 2^{-\rho_2} \rfloor - \lfloor d_1 \cdot 2^{-(\rho_2+6)} \rfloor$
$Q_1' \leftarrow \lfloor d_2/2^{4-(\rho_1+\rho_2)} \rfloor$
Segment 2. Calculate $r_1' \approx c_1 \mod 13$:
$s_1 \leftarrow c_1[\sigma-1:0] - Q_1'[\sigma-4:0] \cdot 2^3$
$s_2 \leftarrow Q_1'[\sigma-3:0] \cdot 2^2 + Q_1'[\sigma-1:0]$
$r_1' \leftarrow s_1[\sigma-1:0] - s_2[\sigma-1:0]$
Segment 3. Obtain the raw remainder R and reduce it to $\mathbb{Z}_{3329}$:
$R \leftarrow \{r_1', c_0\} - Q_1'$
$r \leftarrow \mathrm{MOD}(R, 3329)$
return $r \in \mathbb{Z}_{3329}$

2.5. SHA3 HASH FUNCTION

2.5.1. OVERVIEW OF KECCAK-1600 AND SHA3

Keccak-1600 is the sponge-based cryptographic hash function that serves as the core algorithm for the SHA-3 family of hash functions. It was selected as the winner of the NIST hash function competition in 2012 and standardized as FIPS 202. Keccak operates on a 1600-bit state and utilizes a permutation-based design that supports various output sizes, making it highly versatile for diverse cryptographic applications.

The SHA-3 hash function, built on Keccak, provides a robust and flexible hashing mechanism. It includes variations such as SHA3-224, SHA3-256, SHA3-384, and SHA3-512, defined by their respective output lengths. Unlike SHA-2, which is block-based, SHA-3's sponge construction enables it to absorb input data of arbitrary length and produce hash outputs of configurable size, offering resistance against length-extension attacks.

In the Kyber algorithm, KECCAK is used as a **hashing function** and **pseudo-random generator**. KECCAK is the family of sponge functions with the KECCAK-p$[b, 12+2\ell]$ permutation with pad10^*1 as the padding rule. The family is parameterized by any choices of the rate r and the capacity c such that $r+c$ is in {25, 50, 100, 200, 400, 800, 1600}, i.e..When restricted to the case $b = 1600$, the KECCAK family is denoted by KECCAK[c]; in this case, r is determined by the choice of c.

$$\text{KECCAK}[c] = \text{SPONGE}[\text{KECCAK-p}[1600,24], \text{pad10}^*1, 1600-c].$$

Thus, given an input bit string N and an output length d,

$$\text{KECCAK}[c](N,d) = \text{SPONGE}[\text{KECCAK-p}[1600,24], \text{pad10}^*1, 1600-c](N,d).$$

2.5.2. KECCAK-1600 SPONGE CONSTRUCTION

The Figure 2.4 illustrate the architecture of Keccak-1600. The core of the algorithm is the sponge construction, described in Algorithm 12, which consists of two phases:

- **Absorption Phase**: KECCAK processes the input message by dividing it into blocks and absorbing each block into its state.
- **Squeezing Phase**: After processing all blocks, the sponge function outputs the desired number of bits, which Kyber uses for generating keys or seeds for polynomial coefficients.

The KECCAK permutations are specified, with two parameters:

- The fixed length of the strings that are permuted, called the width of the permutation. In Keccak-1600, the value of Keccak-1600 is 1600 bits.
- The number of iterations of an internal transformation, called a round. In Keccak-1600, the value of Keccak-1600 is 24 rounds.

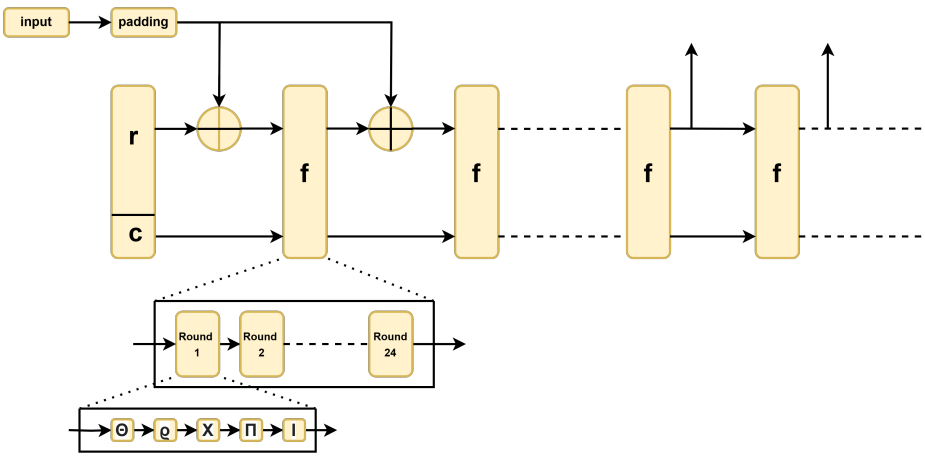

Figure 2.4: The architecture of Keccak-1600 algorithm.

The Keccak algorithm performs five core operations during its permutation process:

1. Theta algorithm (Θ):

$$C[x] = \bigoplus_{y=0}^{4} A[x,y] \quad \text{for } x \in \{0,\ldots,4\} \tag{2.5}$$

$$D[x] = C[(x-1) \bmod 5] \oplus \mathrm{ROT}(C[(x+1) \bmod 5], 1) \tag{2.6}$$

$$A'[x,y] = A[x,y] \oplus D[x] \quad \text{for all } (x,y) \tag{2.7}$$

2. Rho algorithm (ρ):

$$A'[x,y] = \mathrm{ROT}(A[x,y], \rho(x,y)) \tag{2.8}$$

3. Pi algorithm (Π):

$$A'[x,y] = A[(x+3y) \bmod 5, x] \tag{2.9}$$

4. Chi algorithm (χ):

$$A'[x,y] = A[x,y] \oplus \left(\neg A[(x+1) \bmod 5, y] \wedge A[(x+2) \bmod 5, y]\right) \tag{2.10}$$

5. Iota algorithm (ι):

$$A'[0,0] = A[0,0] \oplus \mathrm{RC}[r] \tag{2.11}$$

B[x,y], C[x] and D[x] are intermediate variables. ROT(W,r) the bitwise cyclic shift operation, moving the bit at position i into position i+r (modulo 64). The constants RC[i] and r[x,y] are cyclic shift offset and round constant respectively.

Algorithm 12 SPONGE[f, pad, r](N, d)

Input: String N, Nonnegative integer d.
Output: String Z such that len(Z) = d.

$P \leftarrow N \,||\, \text{pad}(r, \text{len}(N))$ ▷ Pad N to match r's size
$n \leftarrow \text{len}(P)/r$ ▷ Pad N to align with rate r.
$c \leftarrow b - r$ ▷ Compute the capacity.
Divide P into $P_0, P_1, \ldots, P_{n-1}$ where $P = P_0 \,||\, P_1 \,||\, \ldots \,||\, P_{n-1}$
$S \leftarrow 0^b$ ▷ Initialize the state to all zeros
for $i \leftarrow 0$ to $n-1$ **do**
    $S \leftarrow f(S \oplus (P_i \,||\, 0^c))$ ▷ Absorb the i-th block
end for
$Z \leftarrow \emptyset$ ▷ Initialize the output string.
repeat
    $Z \leftarrow Z \,||\, \text{Trunc}_r(S)$ ▷ Append r bits of S to Z
    if $d \leq |Z|$ **then**
        return $\text{Trunc}_d(Z)$ ▷ Return the required d-bit output
    end if
    $S \leftarrow f(S)$ ▷ Permute the state for further squeezing
until $d \leq |Z|$

2.5.3. SHA3 HASH FUNCTION IN CRYSTAL-KYBER

Given a message M, the four SHA-3 hash functions are defined based on appending a two-bit suffix to M and specifying the length of the output. The definitions are as follows:

$$\text{SHA3-224}(M) = \text{KECCAK}[448](M \,||\, 01, 224);$$
$$\text{SHA3-256}(M) = \text{KECCAK}[512](M \,||\, 01, 256);$$
$$\text{SHA3-384}(M) = \text{KECCAK}[768](M \,||\, 01, 384);$$
$$\text{SHA3-512}(M) = \text{KECCAK}[1024](M \,||\, 01, 512).$$

In each of these cases, the **capacity** is set to be double the digest length, that is, $c = 2d$. As a result, the input N to KECCAK[c] is formed by appending the suffix to the original message:

$$N = M \,||\, 01.$$

The purpose of appending this suffix is to support **domain separation**; that is, it ensures that inputs to KECCAK[c] originating from the SHA-3 hash functions can be distinguished from inputs originating from the SHA-3 extendable-output functions (XOFs) as defined in Section 6.2, as well as from other domains that might be defined in the future.

The two SHA-3 XOFs, SHAKE128 and SHAKE256, are defined based on the KECCAK[c] function that a four-bit suffix is appended to M for any specified output length d. The definitions are as follows:

$$\text{SHAKE128}(M, d) = \text{KECCAK[256]}(M \,||\, 1111, d);$$
$$\text{SHAKE256}(M, d) = \text{KECCAK[512]}(M \,||\, 1111, d).$$

This four-bit suffix `1111` enables domain separation by differentiating the inputs for the SHAKE functions from other domains, ensuring clear separation between the usage of SHAKE and other cryptographic applications.

Kyber defines the following instantiations for its cryptographic operations:

- **eXtendable-Output Function XOF** is instantiated with **SHAKE-128**, which are required in various stages of the Kyber protocol for producing random coefficients and other values as needed.
- **Hash Function** H is instantiated with **SHA3-256**, utilized for hashing messages and commitments within the Kyber protocol.
- **Hash Function** G is instantiated with **SHA3-512**, which is used in key generation and encapsulation steps.
- **Pseudorandom Function PRF**(s, b) is instantiated with **SHAKE-256** and takes the concatenation of a seed s and a bitstring b as its input, denoted by $\text{PRF}(s, b) = \text{SHAKE-256}(s \,||\, b)$.
- **Key Derivation Function KDF** is instantiated with **SHAKE-256**, which securely derives keys by generating high-entropy values from initial inputs.

2.5.4. AXI4 PROTOCOLS

The Advanced eXtensible Interface (AXI), along with APB and AHB, constitutes a set of communication protocols specified within ARM's Advanced Microcontroller Bus Architecture (AMBA) standard. These protocols are designed for high-speed, on-chip data transfers. AXI4, a revision of the original AXI protocol, is employed in this project as well as in the Kria SoC platform to interface with the accelerator core.

In AXI4, communication is organized into five distinct channels, illustrated by Figure 2.5:

- **Read Address Channel**: Enables the master to send a read request along with relevant information such as the address and burst type.

- **Read Data Channel**: Allows the slave to return the requested read data along with a response corresponding to the previous read address.
- **Write Address Channel**: Enables the master to send a write request and associated information such as the address and burst type.
- **Write Data Channel**: Permits the master to send the actual write data along with byte-enable strobes.
- **Write Response Channel**: Allows the slave to send a response confirming the completion of a write request.

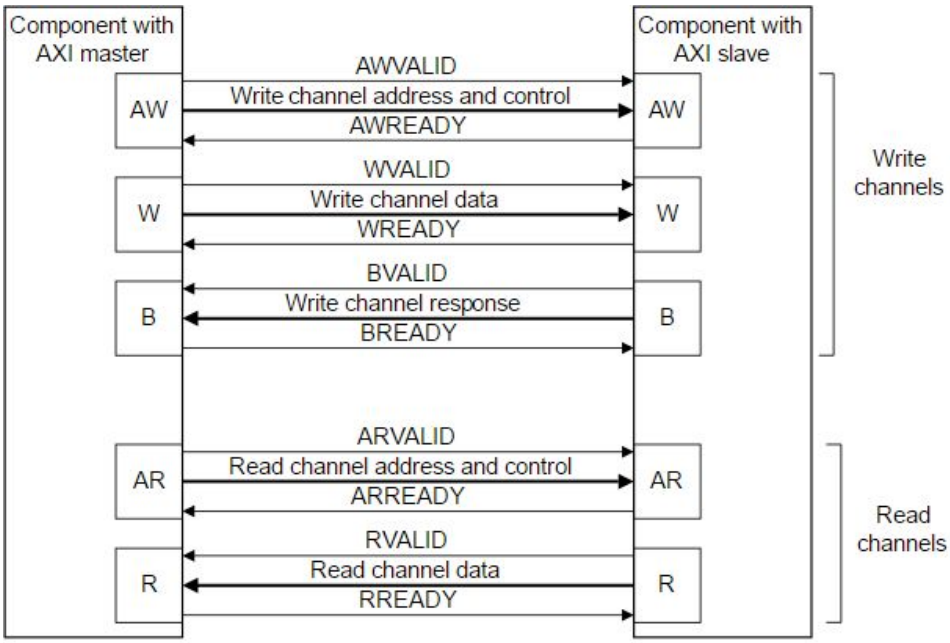

Figure 2.5: AXI Interface's Channels [13]

Although there are certain specifications and restrictions defined within the AXI4 standard, these channels operate relatively independently. For example, the master is not required to wait for the read data to return before issuing subsequent read requests if the slave supports pipelining. This separation of transaction channels enables efficient utilization of bandwidth, reduction of latency, and ultimately facilitates high-performance on-chip communication, making AXI interfaces widely adopted in the semiconductor industry.

READ TRANSACTION

Figure 2.6 illustrates the AXI4 read transaction processes. To initiate a read operation, the master transmits the target address and burst information over the read address channel. Once the slave acknowledges the transaction (indicated by the READY signal being asserted), the associated data will subsequently be transmitted over the read data channel.

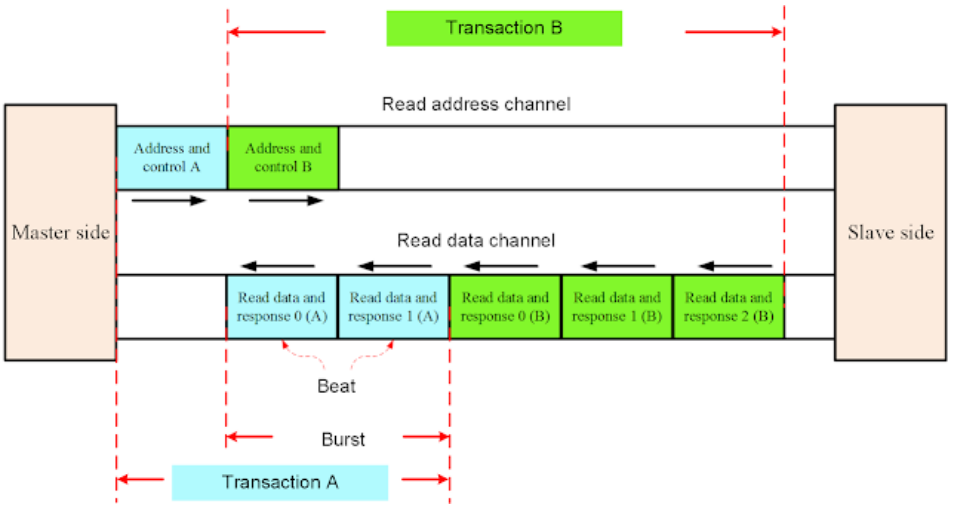

Figure 2.6: Read Transaction [14]

WRITE TRANSACTION

To perform a write operation, the master sends the target address and burst information over the write address channel. Simultaneously, the master prepares the data to be transmitted through the write data channel. These two actions are independent; for instance, the master can begin sending write data before the address transaction is completed. In such cases, the slave may buffer the incoming data until the address information is available. Finally, upon the completion of the write transaction, the slave issues a corresponding write response to inform the master that the operation has been successfully completed. Figure 2.7 descibe how the AXI4 protocol operates write transaction.

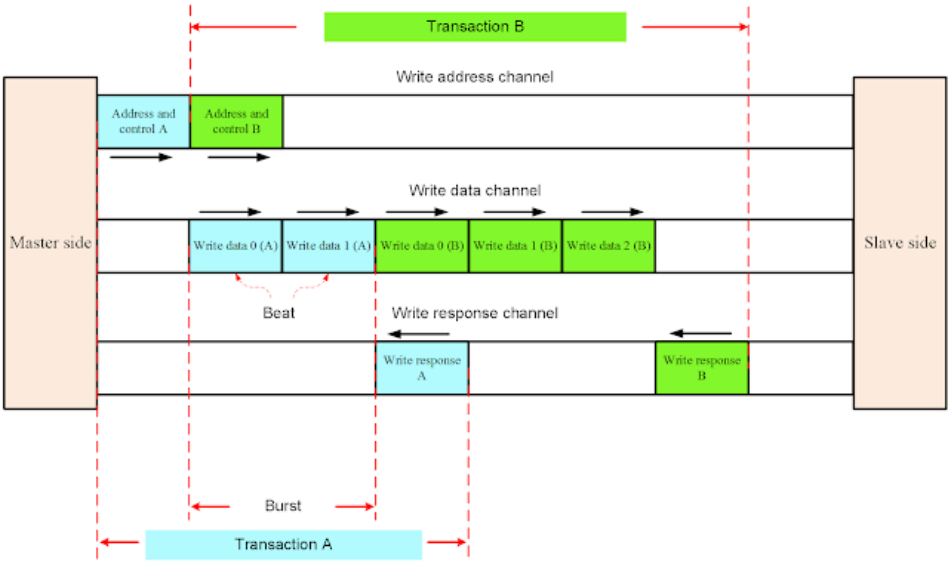

Figure 2.7: Write Transaction [14]

2.5.5. ZYNQ ULTRASCALE+ MPSOC ZCU106 EVALUATION BOARD

The ZCU106 evaluation board serves as a versatile platform for rapid prototyping, leveraging the capabilities of the Zynq UltraScale+ MPSoC ZU7EV device. The ZU7EV integrates a Processing System (PS) comprising a quad-core ARM Cortex-A53 application processor unit (APU) and a dual-core ARM Cortex-R5F real-time processor unit (RPU). This heterogeneous architecture enables unprecedented multi-threaded processing capabilities, catering to a wide range of computational demands.

In addition to the PS, the ZCU106 incorporates a Programmable Logic (PL) layer with a Field-Programmable Gate Array (FPGA), facilitating custom hardware designs. The board is equipped with high-speed DDR4 memory, a Flexible Mezzanine Card (FMC) expansion connector, gigabit-capable serial interfaces, a Video Codec Unit (VCU), and multiple peripheral interfaces. These features collectively provide a robust prototyping environment for advanced embedded and signal processing applications [15].

Key features and specifications of the ZCU106 board (illustrated in Figure 2.8) include [16]:

- Optimized for prototyping applications utilizing the Zynq UltraScale+ MPSoC.
- Integrated video encoding and decoding support for H.264 and H.265 standards.

- HDMI interface for video input and output.
- Support for PCIe® Gen3 Endpoint, USB 3.0, DisplayPort, and SATA interfaces.
- DDR4 SODIMM (72-bit) integrated with the Processing System.
- DDR4 Component (64-bit) integrated with the Programmable Logic.
- Dual Small Form-Factor Pluggable Plus (SFP+) interfaces.
- Dual FPGA Mezzanine Card (FMC) interfaces for I/O expansion.

The ZCU106 board's technical specifications are summarized in Table 2.2:

Parameter	Value
System Logic Cells (K)	504
Memory	38 Mb
DSP Slices	1,728
Video Codec Unit (VCU)	1
Maximum I/O Pins	464

Table 2.2: Technical Specifications of the ZCU106 Evaluation Board

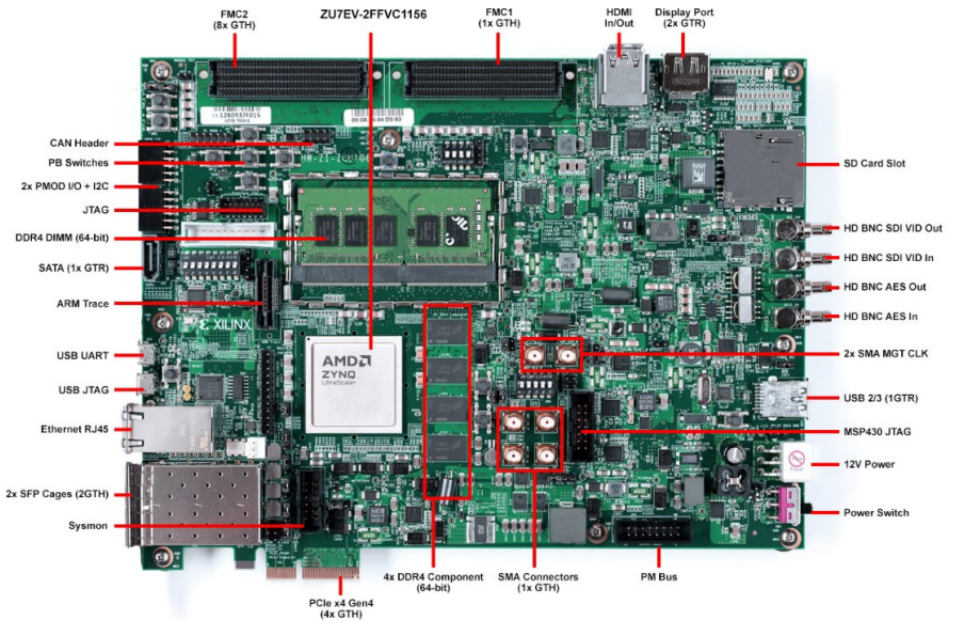

Figure 2.8: Features of ZCU106 Evaluation Board

2.5.6. PYNQ FRAMEWORK

The PYNQ (Python Productivity for Zynq) framework is an open-source initiative that provides a high-level programming environment for Zynq-based platforms, leveraging the Python programming language and the Jupyter Notebook interface. It abstracts low-level hardware complexities, thereby enabling rapid development and prototyping, particularly for artificial intelligence (AI) and embedded applications.

At the core of the PYNQ framework lies the concept of an *Overlay*, which encapsulates a pre-designed FPGA bitstream alongside a corresponding software application programming interface (API). The Overlay acts as a bridge between the software and hardware domains, managing the configuration of the Programmable Logic (PL) layer and facilitating data transfer between the Processing System (PS) and PL layers. A well-crafted Overlay abstracts the intricate hardware details, presenting a simplified interface to software developers and thereby significantly enhancing development productivity.

Traditionally, embedded systems development has relied on languages such as C or C++ to achieve performance-critical requirements. In contrast, PYNQ prioritizes ease of use and rapid development by employing Python. However, this higher level of abstraction may introduce performance trade-offs, potentially limiting the realization of the hardware's full computational capabilities compared to lower-level implementations.

Deploying PYNQ on a development board typically requires a microSD card preloaded with a compatible Linux image. Upon initialization, the PS layer's CPU boots into the Linux operating system, enabling access to the board through Ethernet or network connectivity. This configuration supports seamless interaction with the PYNQ environment, facilitating both local and remote development workflows.

Overall, PYNQ represents a paradigm shift in FPGA-based system design, democratizing access to complex hardware resources and enabling a broader range of developers, including those with limited hardware design experience, to innovate and accelerate application development.

2.6. RELATED WORK

As quantum computing advances, efficient hardware implementations of CRYSTALS-Kyber have gained significant attention, particularly on FPGAs, which balance performance and flexibility. This section highlights notable works in optimizing Kyber's core operations, such as modular reduction, NTT, and SHA-3 integration.

The lightweight hardware implementation of CRYSTALS-Kyber [4] proposes a resource-efficient hardware architecture for the Kyber KEM protocol, focusing on lightweight applications, illustrated in figure 2.9. Key innovations include a novel modular multiplication unit (MMU) without DSPs, optimized school-book polynomial multiplication (SPM) instead of the NTT core, and an improved hash module that reduces flip-flop usage by 48% through register reuse. The implementation, on a Kintex-7 FPGA at 244 MHz, achieves key generation, encapsulation, and decapsulation in 278 μs, 416 μs, and 552 μs, respectively, using 4,777/4,993 LUTs, 2,661/2,765 FFs, and 2.5 BRAMs for client/server sides. Compared to existing works, this design minimizes hardware resources and is well-suited for resource-constrained devices like IoT and mobile systems.

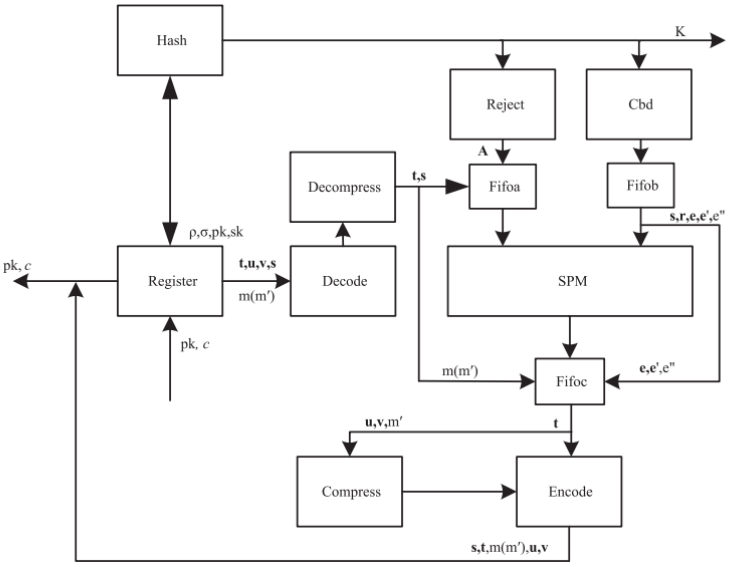

Figure 2.9: Overall lightweight hardware structure of CRYSTALS-Kyber [4]

The architecture shown in Figure 2.10 achieves a remarkable speedup of up to 129x compared to Cortex-M4 implementations. By reusing key modules such as the NTT, inverse NTT, and hash functions, and incorporating pipelining

and parallel execution, the design (presented in [5]) maximizes both resource utilization and performance. Operating at 155 MHz on Artix-7 and 192 MHz on Virtex-7 FPGAs, the architecture supports three security levels while significantly reducing clock cycles and efficiently utilizing BRAM for intermediate data storage. However, the design comes with trade-offs, including increased hardware complexity and a focus on optimizing encryption and decryption at the expense of key generation capabilities.

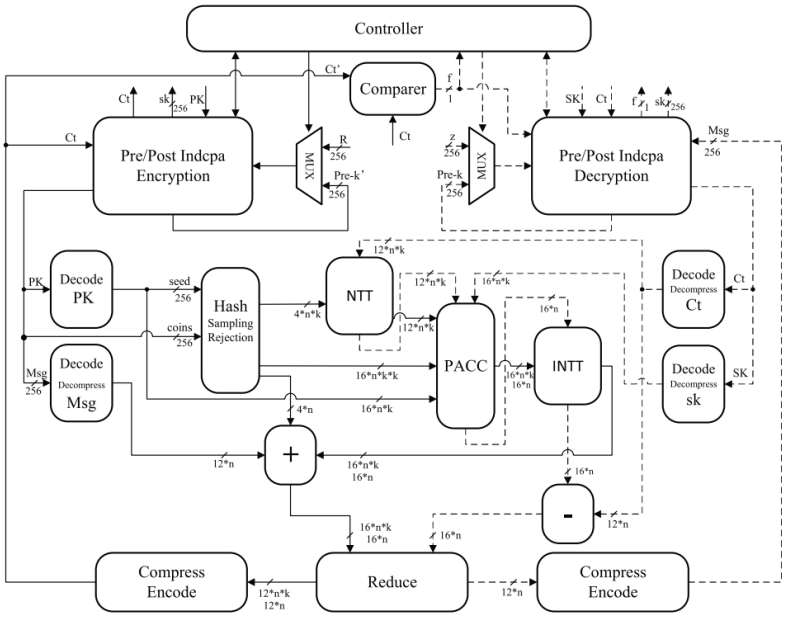

Figure 2.10: Overall pure Kyber encapsulation and decapsulation architecture [5]

The HPKA accelerator proposed in [8] optimizes CRYSTALS-Kyber, a quantum-resistant Key Encapsulation Mechanism (KEM), using inter- and intra-module pipelining and resource-sharing strategies. Implemented on Artix-7 and Zynq UltraScale+ FPGAs, it delivers 25–51% speedup and 50–75% DSP reduction compared to prior designs, achieving high Area-Time (AT) efficiency. Notable features include efficient NTT/INTT modules requiring only 128 cycles per computation and FIFO-based buffering to minimize resource use. While highly efficient, its focus on performance excludes countermeasures against side-channel attacks, leaving potential security concerns. The figure 2.11 below depicts its architecture.

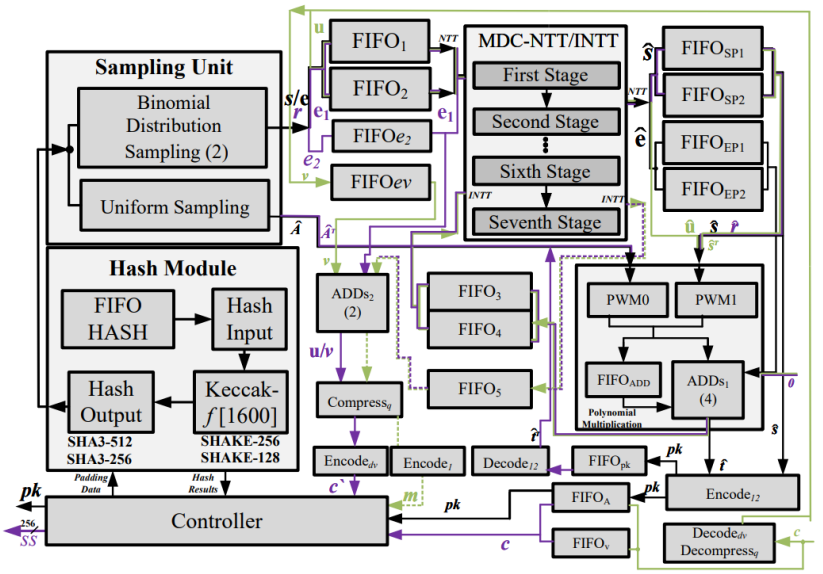

Figure 2.11: The overall server-side HPKA Kyber768 architecture [8]

In [7], the authors propose a compact FPGA-based hardware design for CRYSTALS-KYBER, operating at 159 MHz on a Xilinx Artix-7. Key innovations include an optimized modular reduction for $q = 3329$, a reconfigurable data path for NTT/INTT and PWM, and a pipelined modular adder integrating $\times\frac{1}{2}$ operations. The design as shown in figure 2.12 uses 2,253 slices, 7.9k LUTs, 3.6k FFs, 4 DSPs, and 16 BRAMs, achieving execution times of 49.1 μs (KeyGen), 52.8 μs (KENC), and 66.0 μs (KDEC). The architecture of this design is illustrated in Figure 2.12.

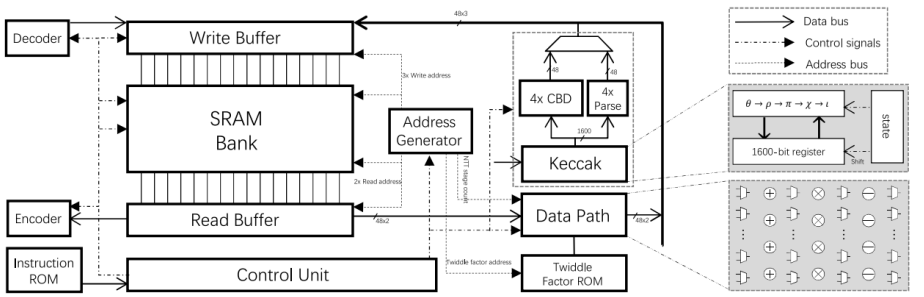

Figure 2.12: The area-time efficient hardware architecture for CRYSTALS-Kyber [7]

3

PROPOSED ARCHITECTURE

3.1. SYSTEM ARCHITECTURE

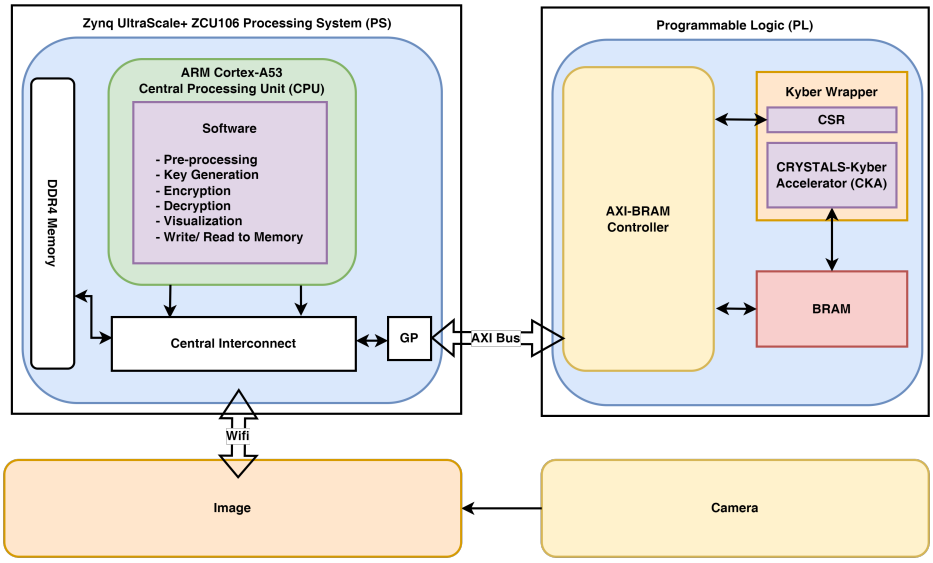

Figure 3.1: System Architecture

Figure 3.1 presents the overall architecture of the proposed system. In this system, we divided the implementation into two major domains:

1. **Processing System (PS)**: Includes ARM Cortex-A53 CPU, DDR4 memory, AXI interfaces, and the central interconnect.

- **ARM Cortex-A53 CPU**: The CPU runs embedded Linux and user-level applications written in Python. It performs the following operations:
  - **Pre-processing**: Load, resize and convert the images into hex arrays.
  - **Key Generation, Encryption, and Decryption**: In collaboration with the Kyber engine.
  - **Visualisation**: Read, convert and plot encrypted/decrypted results.
  - **Read Write to Memory**: Perform write/read operations.
- **DDR4 Memory**: Stores raw image data, intermediate buffers, and encryption results.
- **Central Interconnect**: Every read/write made by an AXI master (like the Cortex-A53) goes through the Central Interconnect. It serves as the backbone for AXI-based communication between the Processing System (PS) and Programmable Logic (PL). It facilitates data routing, arbitration, and protocol conversion among various AXI master and slave interfaces.
- **AXI-GP Port**: AXI General Purpose port is used for register-level communication (MMIO) from the PS to control the Kyber Wrapper in PL. This is low-bandwidth but suitable for small data/control transfers.

2. **Programmable Logic (PL)**: Implements the Kyber Wrapper, cryptographic accelerator, control logic, and BRAM memory.

   - **Kyber Wrapper**: A top-level hardware module that orchestrates key generation, encryption and decryption operations. It includes:
     - **CSR (Control and Status Register)**: The collection of register bank and logic that triggers cryptographic functions.
     - **CRYSTALS-Kyber Accelerator (CKA)**: Implements core Kyber operations.
   - **BRAM**: Block RAM is used as temporary local storage for cryptographic data:
     - Public and private keys
     - Ciphertexts and messages
     - Random coin value
   - **AXI-BRAM Controller**: Bridges the AXI interface and BRAM blocks. It allows the PS to write data to BRAM before triggering the Kyber operation and read the result after finishing.

In this system, the operation begins with a camera capturing an image, which is then wirelessly transmitted to the Processing System (PS) for pre-processing. The processed image is stored in DDR4 memory. The PS sends control signals and relevant data to the Programmable Logic (PL) through the AXI General Purpose (AXI-GP) port, activating the Kyber Wrapper module. Based on the control signals, the Kyber Wrapper triggers the Cryptographic Kernel Accelerator (CKA) to perform one of three operations: key generation, encryption, or decryption. Once the cryptographic operation is completed, the result is stored BRAM and returned to the PS through the MMIO interface. The PS then reads the result, processes and displays the processed image.

3.2. KYBER WRAPPER ARCHITECTURE

3.2.1. INTERFACE

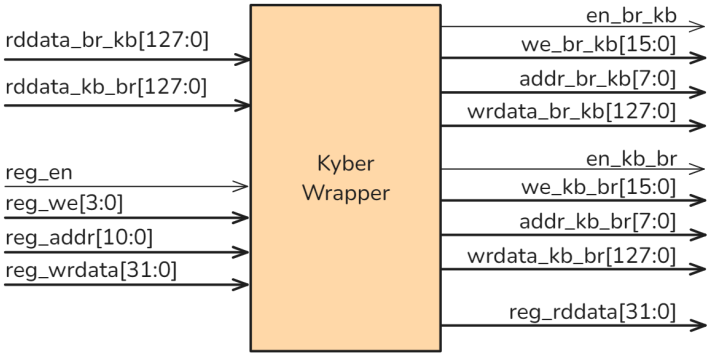

Figure 3.2: Kyber Wrapper Interface

Figure 3.2 illustrates the register-level interface of the Kyber wrapper module, highlighting the connection between the processor system (PS) and BRAM. The interface includes control registers that allow the PS to configure and trigger operations, with signals for address selection, data writing, and reading, as well as enabling or disabling register access. The interface also connects to BRAM (Block RAM) for storing and retrieving data such as keys, ciphertext, and decrypted messages during the cryptographic processes.

3.2.2. INTERFACE SIGNALS DESCRIPTION

Port Name	Direction	Width	Description
reg_addr	Input	11	Address for control/status registers
reg_clk	Input	1	Clock signal
reg_en	Input	1	Register access enable
reg_rddata	Output	32	Read data from registers
reg_rst	Input	1	Synchronous reset
reg_we	Input	4	Byte-enable for write
reg_wrdata	Input	32	Data to be written
addr_br_kb	Output	8	Address to BRAM (read)
en_br_kb	Output	1	Read enable for BRAM
rddata_br_kb	Input	128	Data read from BRAM
we_br_kb	Output	16	Write enable mask for BRAM
wrdata_br_kb	Output	128	Data to be written to BRAM
addr_kb_br	Output	8	Address to BRAM (write)
en_kb_br	Output	1	Write enable for BRAM
rddata_kb_br	Input	128	Data read by Kyber core
we_kb_br	Output	16	Write mask for BRAM
wrdata_kb_br	Output	128	Data written by Kyber core

Table 3.1: Kyber Wrapper Module Ports

Table 3.1 lists the input and output signals of the `kyber_wrapper` module, including register interface and BRAM access signals for data movement. The `kyber_wrapper` module interfaces with dual-port BRAM to facilitate the reading of input data and writing of output data during Kyber cryptographic processing. In the reading phase, the Kyber core reads data from BRAM (called **BRAM_IN**) using the `*_br_kb` signals. This interface ensures that the Kyber core can retrieve necessary input data, such as public keys or messages, from BRAM. The `we_br_kb` signal, when active, allows the Kyber core to modify the data in BRAM as required during computation. This bi-directional communication ensures smooth data flow throughout the cryptographic processes in the Kyber core. In the writing phase, the Kyber core writes data to BRAM (called **BRAM_OUT**) using the `*_kb_br` signals.

3.2.3. BLOCK DIAGRAM

The overall internal architecture of the `kyber_wrapper` module is shown in Figure 3.3, which integrates control logic, FSM, and BRAM access for Kyber processing. The `kyber_wrapper` module serves as a top-level integration wrapper for a Kyber post-quantum cryptographic processor. It coordinates data move-

ment between a control/status register interface, a finite state machine (FSM), and external dual-port BRAMs for cryptographic input and output.

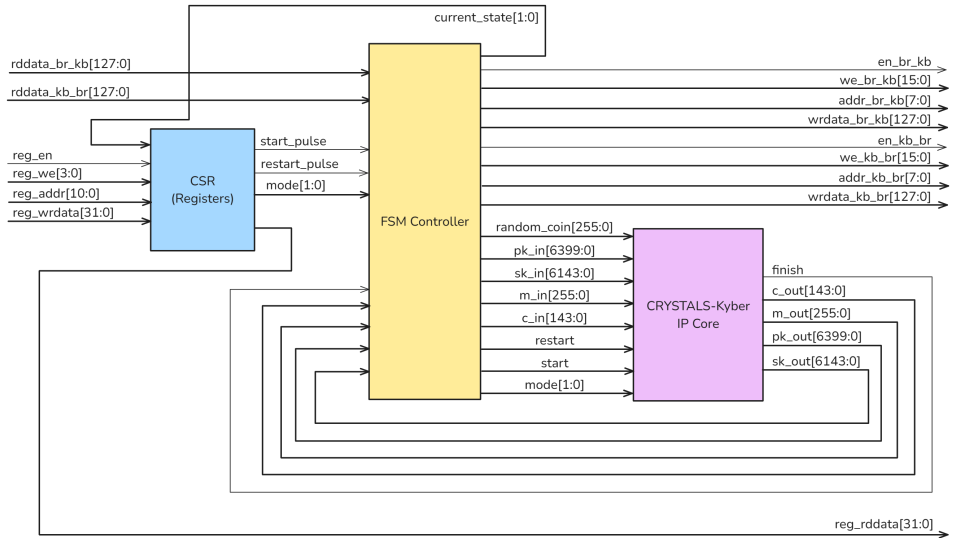

Figure 3.3: Kyber Wrapper Architecture

3.3. CRYSTALS-KYBER ACCELERATOR ARCHITECTURE

3.3.1. INTERFACE

The diagram below represents a hardware accelerator interface for the Kyber cryptographic scheme, which implements several core operations in hardware for efficiency:

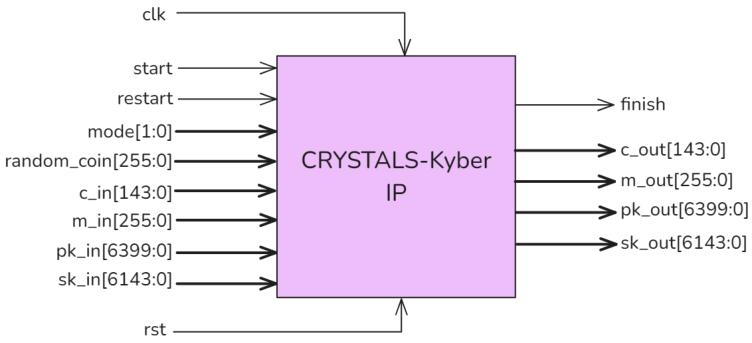

Figure 3.4: CRYSTALS-Kyber interface design

Figure 3.4 shows the interface design of the Kyber hardware accelerator, de-

tailing the key control and data signals required for cryptographic operations. The `kyber` module provides the necessary interface to perform post-quantum cryptographic operations based on the Kyber algorithm, including key generation, encryption, and decryption. It operates under the control of a clock (`clk`) and a synchronous reset (`rst`). The process is initiated with the `start` signal, and the internal finite state machine (FSM) can be reset using the `restart` signal. The operational mode is determined by the 2-bit `mode` input, where `00` corresponds to key generation, `01` to encryption, and `10` to decryption.

3.3.2. Interface Signals Description

Signal Name	Direction	Width	Description
`clk`	Input	1	Clock signal to drive the module
`rst`	Input	1	Synchronous reset signal
`start`	Input	1	Signal to start the cryptographic operation
`restart`	Input	1	Signal to reset internal FSM for a new operation
`mode`	Input	2	Selects operation mode: 00 = keygen, 01 = encrypt, 10 = decrypt
`random_coin`	Input	256	Random bits used for cryptographic randomness
`m_in`	Input	256	Plaintext message to be encrypted
`pk_in`	Input	6400	Public key used for encryption
`sk_in`	Input	6144	Secret key used for decryption
`c_in`	Input	6144	Ciphertext input to be decrypted
`state`	Output	5	Current state of the internal FSM
`m_out`	Output	256	Decrypted message result
`pk_out`	Output	6400	Generated public key (during keygen)
`sk_out`	Output	6144	Generated secret key (during keygen)
`c_out`	Output	6144	Ciphertext result from encryption
`finish`	Output	1	High when operation is completed

Table 3.2: Kyber Module Signal Descriptions

Table 3.2 details the signal-level interface of the Kyber hardware accelerator, including mode control, cryptographic data inputs/outputs, and operational status outputs. To support secure operations, the module takes in a 256-bit random seed (`random_coin`). It also accepts a 256-bit message input (`m_in`), a 6400-bit public key input (`pk_in`), a 6144-bit secret key input (`sk_in`), and a

6144-bit ciphertext input (`c_in`), depending on the operation mode.

The outputs of the module include the 256-bit decrypted message (`m_out`), the 6400-bit public key (`pk_out`), the 6144-bit secret key (`sk_out`), and the 6144-bit ciphertext (`c_out`). The current state of the FSM is exposed through the 5-bit `state` output, and the `finish` signal indicates the completion of the current operation. This interface enables seamless integration into a secure hardware design for efficient cryptographic processing.

3.3.3. BLOCK DIAGRAM

The Figure 3.5 below shows the block diagram the Kyber cryptographic scheme. The clock (clk) and reset (rst_n) signals were omitted from the diagram for clarity and to avoid visual clutter, as including them would make the wiring appear too complicated and harder to follow.

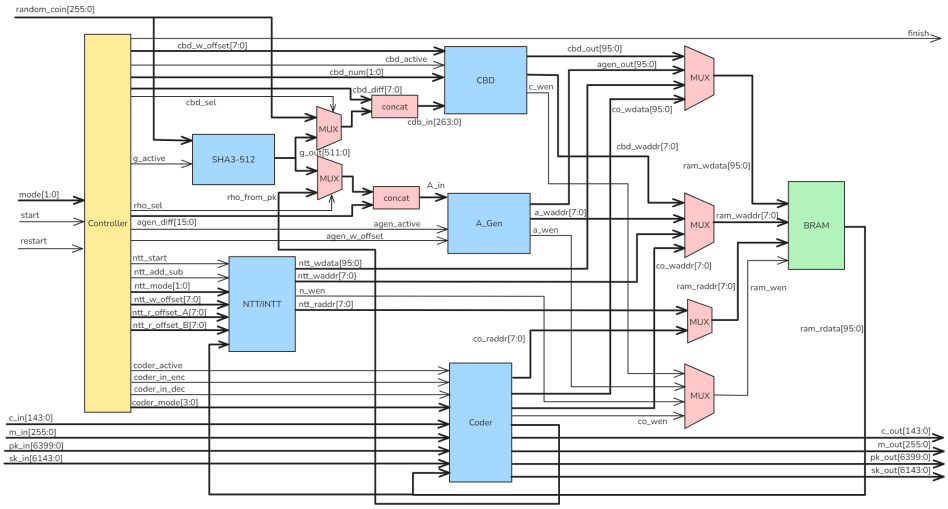

Figure 3.5: CRYSTALS-Kyber block diagram design

The Kyber cryptographic processor operates in three distinct modes—key generation, encryption, and decryption—with each mode activating a specific set of submodules and data paths.

1. **Mode 0 – Key Generation:** In key generation mode, the Kyber processor starts with a 256-bit `random_coin` input, which is fed into the `SHA3_512` module (`G`) to produce two cryptographic seeds: `rho` and `sigma`. The `rho` value is passed to the `A_generator` to generate the public matrix A, while `sigma` is routed to the `CBD` (small polynomial generator) to create secret polynomials. These secret values are transformed into the NTT (Number

Theoretic Transform) domain using the ntt_processor, enabling efficient polynomial operations. The resulting data, including matrix A and secrets, is combined and encoded by the coder module, which produces the output public key (pk_out) and secret key (sk_out). Intermediate values are stored in BRAM via the RAM_WRAPPER, with precise coordination from the controller.

2. **Mode 1 – Encryption:** In encryption mode, the processor again starts by hashing the random_coin through SHA3_512 to obtain fresh rho and sigma values. The rho is used by the A_generator to reconstruct matrix A, while sigma or the original random_coin feeds into CBD to generate ephemeral polynomials. These are transformed via ntt_processor and used with the provided public key (pk_in) and plaintext message (m_in) to generate a ciphertext. The coder module is responsible for loading all required inputs and orchestrating the polynomial arithmetic and encoding steps. The resulting ciphertext is output as c_out. All submodules communicate via shared BRAM, controlled by the central controller, ensuring correct data sequencing and flow.

3. **Mode 2 – Decryption:** In decryption mode, the operation is streamlined as only the coder module is active. It takes in the secret key (sk_in) and the ciphertext (c_in) as inputs. Using these, it performs decryption operations internally—such as inverse NTT and polynomial subtraction—to recover the original message. This message is then output as m_out. The shared BRAM is used for temporary storage and data fetching, with address and control signals managed to avoid conflicts. No new polynomials or matrices are generated in this mode, making it more computationally lightweight compared to the other modes.

The Accelerator follows a modular design, where each functional block is responsible for a specific task in the key generation, encryption, or decryption processes. This clear separation of functionality enhances flexibility, scalability, and ease of verification and performance optimization.

- **Controller:** Central state machine that orchestrates the operation of the entire design by issuing control signals, managing module activation, and sequencing memory accesses based on the current mode.

- **SHA3-512:** Implements the cryptographic hash function used to generate pseudo-random outputs (rho, sigma) used in key and ciphertext construction.

- **CBD (Centered Binomial Distribution):** Generates secret or error polynomials s, e, r, e_1, and e_2 with coefficients sampled from a centered binomial distribution, essential for both key generation and encryption randomness.
- **A_Gen (Matrix A Generator):** Generates the public matrix A (or A^T) deterministically from `rho` using SHAKE-based uniform sampling, storing output coefficients into BRAM.
- **NTT/INTT Block:** Performs NTT, INTT, and pointwise multiplication on polynomials, accelerating polynomial arithmetic over the ring $\mathbb{Z}_q[x]/(x^n+1)$.
- **Coder Module:** Performs serialization and deserialization of public keys, secret keys, messages, and ciphertexts. Also performs compression and decompression operations.
- **BRAM (Block RAM):** Serves as shared memory for data exchange among submodules, accessed through address lines and write-enable signals routed via MUXes.
- **Multiplexers (MUX):** Dynamically route memory addresses and data based on control signals, ensuring each submodule can access BRAM when needed without conflict.
- **Concat Blocks:** Join hash outputs and control parameters into single input streams for downstream modules like CBD and A_Gen.

The architectural design and implementation details of each submodule mentioned above will be thoroughly described in the following section.

4

IMPLEMENTATION

4.1. SYSTEM IMPLEMENTATION

The overall hardware architecture is illustrated in Figure 4.1, showing the connection between the Zynq Processing System and the cryptographic accelerator modules.

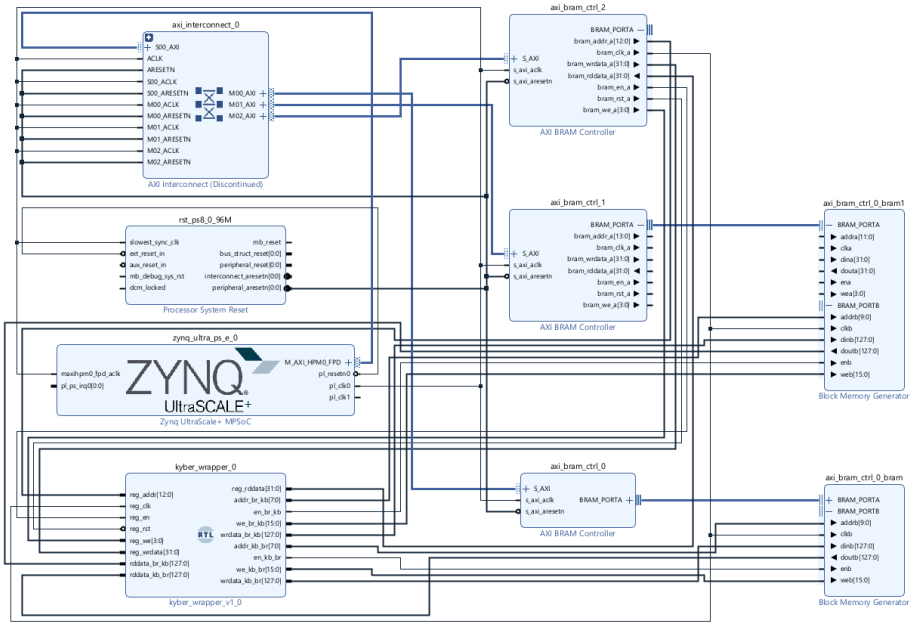

Figure 4.1: System Implementation on Xilinx SoC

This block design implements a cryptographic accelerator using the Kyber post-quantum encryption scheme on a Xilinx Zynq UltraScale+ MPSoC. The design integrates a Zynq Processing System, Kyber wrapper hardware accelerator, and multiple AXI BRAM Controllers for memory-mapped communication and storage:

- **Zynq UltraScale+ MPSoC (zynq_ultra_ps_e_0)**
  - Acts as the central processor, provides the AXI master interface for communication with programmable logic (PL).
- **AXI Interconnect (axi_interconnect_0)**
  - Routes traffic between the Zynq MPSoC master and three AXI slave peripherals:
    - `axi_bram_ctrl_0`
    - `axi_bram_ctrl_1`
    - `axi_bram_ctrl_2`
  - Interfaces:
    - Address width: 32 bits
    - Data width: 32 bits
- **AXI BRAM Controllers:**
  - `axi_bram_ctrl_0` connects to `axi_bram_ctrl_0_bram`
  - `axi_bram_ctrl_1` connects to `axi_bram_ctrl_1_bram`
  - `axi_bram_ctrl_2` directly connects to the CSR interface of the Kyber Wrapper module.
- **Block Memory Generators (BRAM)**
  - Dual-port configuration for simultaneous access from PS and logic.
    - Port A: 32-bit data width
    - Port B: 128-bit data width
    - Size: 128 bits × 1024 = 16 KB
- **Kyber Wrapper (kyber_wrapper_0)**
  - BRAM interfaces 1 (`BRAM_IN`): The Zynq PS write input data to port A of BRAM 1 (`axi_bram_ctrl_1_bram`), and the accelerator accesses port B to collect data.

– BRAM interfaces 2 (BRAM_OUT): The Kyber wrapper module output the result to port B of this BRAM (axi_bram_ctrl_0_bram), and then the Zynq PS can read the data.
– Control status registers interface: designed to send and receive control and status signals such as start, mode, finish,...

4.2. KYBER WRAPPER

The control logic of the Kyber wrapper is governed by a finite state machine (FSM) with four states, as shown in Figure 4.2:

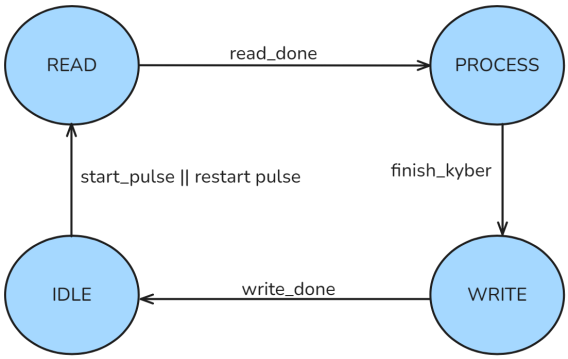

Figure 4.2: Kyber wrapper FSM

- **IDLE**: The FSM starts in the IDLE state, where it waits for a pulse from external signals such as `start_pulse` or `restart_pulse`. Once one of these pulses is received, the FSM transitions to the READ state. If no pulse is received, the FSM remains in the IDLE state.
- **READ**: In this state, the FSM reads data from the BRAM interface (BRAM_IN). The data corresponds to various inputs such as the random coin, public key (`pk_in`), secret key (`sk_in`), and message input (`m_in`). Once sufficient data has been read, indicated by the `read_done` flag, the FSM transitions to the PROCESS state.
- **PROCESS**: After the READ state, the FSM enters the PROCESS state, where the Kyber algorithm is executed. The algorithm processes the input data and outputs the results. The `finish_kyber` signal is used to determine when processing is complete. Once the processing is finished, the FSM transitions to the WRITE state.
- **WRITE**: During the WRITE state, the processed data is written back to the BRAM interface (BRAM_OUT). This includes data such as the public

key, secret key, ciphertext, or message output. The FSM writes the data in chunks, and the `write_done` flag is set when the writing operation is complete. Once all data is written, the FSM returns to the `IDLE` state, awaiting the next operation.

4.3. CONTROLLER

Figure 4.3 presents the interface of the controller module that coordinates all operations inside the Kyber hardware.

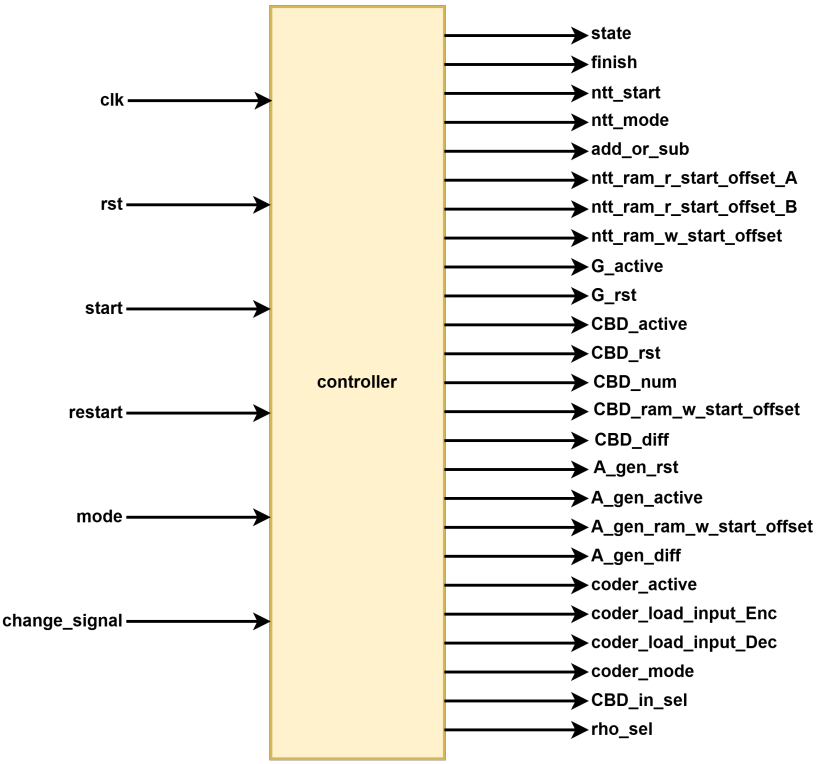

Figure 4.3: Controller

Objective: This function manages the operation of the Kyber hardware. It determines the address values and start signals for the input ADDSUB, NTT/INTT, RAM, HASH, and Coder modules. Additionally, it controls the mode of operation for Kyber (encryption, decryption, or key generation). The function also manages the read and write addresses of the RAM to ensure seamless operation.

Input Specifications:

- `mode[1:0]`: Specifies the operation mode.
  - `0` - Key generation.
  - `1` - Encryption.
  - `2` - Decryption.
- `start`: Signal to trigger the Kyber process.
- `restart`: Signal to trigger the Kyber process in encrypt mode in case there is no update in public key.

Output Specifications:

- `finish` (1 bit): trigger when finishing the operation.
- `cbd_w_offset` (8 bits): Write address offset for storing CBD-generated coefficients in RAM.
- `cbd_active`: Control signal to enable the CBD sampling module.
- `cbd_diff` (7 bits): Differential value for CBD coefficient generation.
- `cbd_sel`: Mode selection signal for the CBD module.
- `g_active`: Control signal to enable the SHA3-512 hash module.
- `rho_sel`: Selection signal for sourcing ρ from the public key or alternate input.
- `agen_diff` (15 bits): Offset adjustment for address generation in A_Gen.
- `agen_active`: Control signal to enable the A_Gen module.
- `agen_w_offset` (8 bits): Write address offset for A_Gen-generated coefficients.
- `ntt_mode` (2 bits): Specifies the operation mode:
  - `0` - Execute NTT.
  - `1` - Execute INTT.
  - `2` - Perform point-wise multiplication.
- `ntt_w_offset`(8 bits): start address of RAM for storing output data.
- `ntt_r_offset_A`(8 bits): start address of RAM for reading input data.

– `ntt_r_offset_B`(8 bits): start address of RAM for reading input data (used in multiply).

– `addsub_start` (1 bit): Signal to trigger the NTT process.

– `add_flag` (1 bits): Specifies the operation mode:

  • `0` - addition operation.
  • `1` - subtraction operation.

– `addsub_r_offset_A`(8 bits): start address of RAM for reading input data.

– `addsub_r_offset_B`(8 bits): start address of RAM for reading input data.

– `coder_active`: To activate the `coder` module.

– `coder_in_enc`: Triggers input loading for encryption.

– `coder_in_dec`: Triggers input loading for decryption.

– `coder_mode`: Selects the operation mode of the `coder` module.

4.4. NTT/INTT

4.4.1. INTERFACE

The functional diagram of the NTT block is depicted in Figure 4.4, detailing the interface with RAM and control logic.

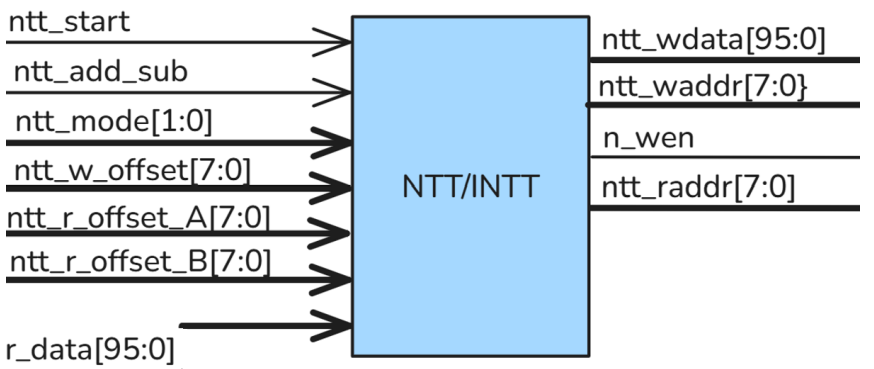

Figure 4.4: NTT block

Objective: Perform operations including the Number Theoretic Transform (NTT), Inverse NTT (INTT), and point-wise multiplication.
Input Specifications:

- `ntt_start` (1 bit): Signal to trigger the NTT process.
- `ntt_mode` (2 bits): Specifies the operation mode:
  - `0` - Execute NTT.
  - `1` - Execute INTT.
  - `2` - Perform point-wise multiplication.
- `ntt_w_offset`(8 bits): start address of RAM for storing output data.
- `ntt_r_offset_A`(8 bits): start address of RAM for reading input data.
- `ntt_r_offset_B`(8 bits): start address of RAM for reading input data (used in multiply).
- `r_data` (96 bits): Data provided for NTT or INTT computations.

Output Specifications:

- `ntt_raddr` (8 bits): Address used to read the input data.
- `ntt_waddr` (8 bits): Address used to store the output data.
- `w_en` (1 bit): Signal to enable writing output data into system memory.
- `ntt_wdata` (96 bits): Resulting data after the selected operation.

4.4.2. Overall structure

The internal structure of the NTT/INTT computation unit is illustrated in Figure 4.5.

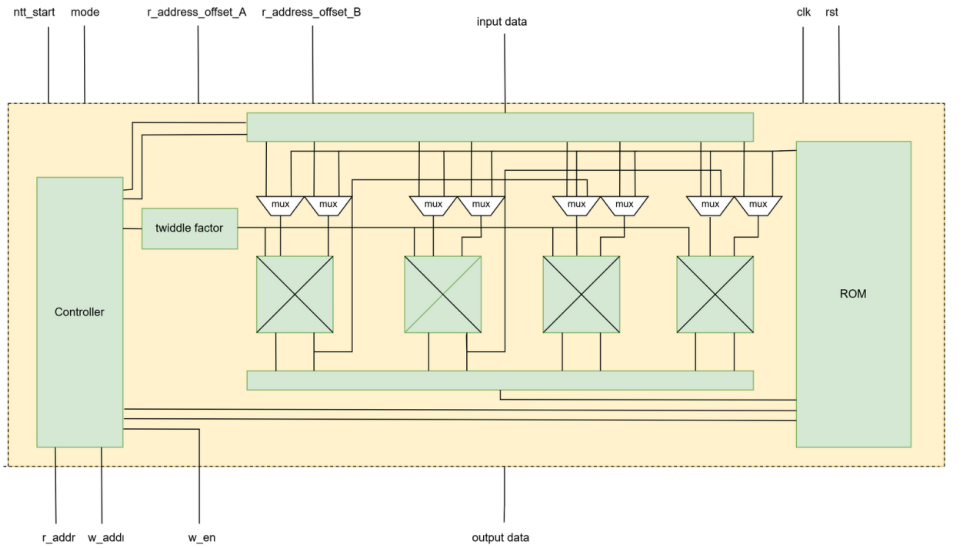

Figure 4.5: NTT struture

Objective: Perform operations including the Number Theoretic Transform (NTT), Inverse NTT (INTT), and point-wise multiplication.

Input Specifications:

– `ntt_start` (1 bit): Signal to trigger the NTT process.

– `mode` (2 bits): Specifies the operation mode:

- `0` - Execute NTT.
- `1` - Execute INTT.
- `2` - Perform point-wise multiplication.

– `input_data` (96 bits): Data provided for NTT or INTT computations.

Output Specifications:

– `r_addr` (7 bits): Address used to read the input data.

– `w_addr` (7 bits): Address used to store the output data.

– `w_en` (1 bit): Signal to enable writing output data into system memory.

– `output_data` (96 bits): Resulting data after the selected operation.

4.4.3. BUTTERFLY CORE

The core butterfly computation block used in NTT/INTT is shown in Figure 4.6.

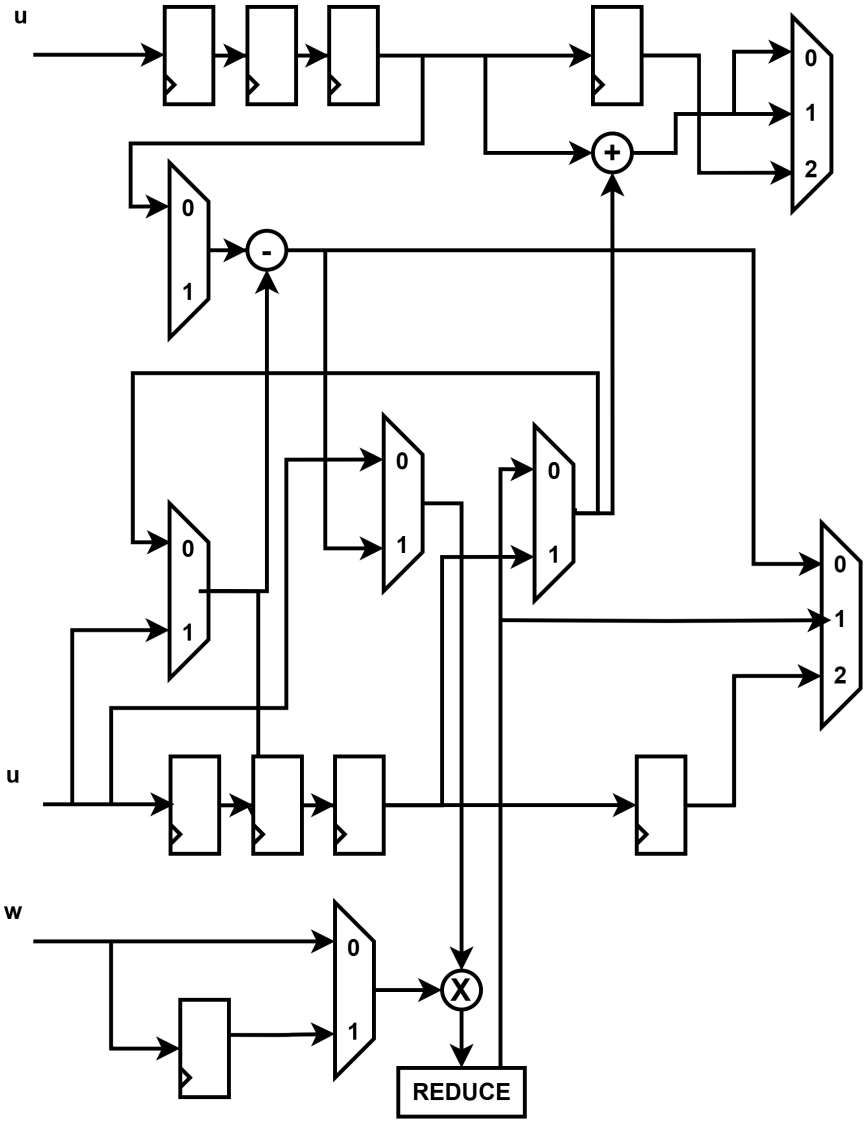

Figure 4.6: Butterfly Structure

Objective: Execute the calculation for NTT/INTT.

Input Specifications:

– `mode` (2 bits): Specifies the operation mode:

- `0` - Execute NTT.
- `1` - Execute INTT.
- `2` - Perform point-wise multiplication.

– `input_1` (12 bits): first input data.

– `input_2` (12 bits): second input data.

– `coef` (12 bits): twiddle factor that is pre-caculated and feed in data.

Output Specifications:

– `output_1` (12 bits): first output data.

– `output_2` (12 bits): second output data.

4.4.4. ROM

Objective: This NTT ROM is designed to store the results of NTT calculations and prepare for subsequent calculations. The ROM has a capacity of 96x32 bits, consisting of 32 blocks, each 96 bits in size.

Input Specifications:

– `in_addr` (4 bits): Address used to store output data.

– `in_data` (96 bits): Input data.

– `out_addr` (4 bits): Address used to read data from the ROM.

Output Specifications:

`out_data` (96 bits): Data retrieved from the ROM.

4.4.5. TWIDDLE FACTOR

Objective: This block stores the twiddle factors required for calculations in NTT and INTT operations.

Input Specifications:

`in_addr` (4 bits): Address used to access the twiddle factor data.

Output Specifications:

`out_data` (24 bits): Data retrieved from the twiddle factor block.

4.4.6. NTT/INTT CONTROLLER

Objective: This function controls the activity of the NTT/INTT module. It determines the address values for the input ROM, twiddle factors, and the output of the NTT/INTT module. This block ensures that the module operates using a low-complexity NTT/INTT algorithm.

Input Specifications:

`ntt_start` (1 bit): Signal to start NTT or INTT operations.

Output Specifications:

All the address of other block.

4.5. HASH MODULES

The CRYSTALS-Kyber algorithm employs several critical hashing functions during key generation, encryption, and decryption processes. The Hash modules play a pivotal role in generating random distribution samples for functional sampling blocks such as the Sampling Unit. This makes the these modules become a potential computational bottleneck in the entire design. Therefore, ensuring the throughput of these Hash modules matches that of other computational blocks is crucial. The hash functions used include `SHAKE128`, `SHAKE256` and `SHA3-512`, all compliant with the *FIPS-202* standard [19].

4.5.1. KECCAK-P

These hash functions share a common foundation in the *Keccak* algorithm [20], differing in their rate r and the suffix appended to each input to separate domains. This enables the reuse of Keccak hardware for the various hashing functions needed in the execution of the CRYSTALS-Kyber algorithm. The Keccak hardware, referenced from [20], is tailored to meet the design requirements. The Keccak hardware requires 24 clock cycles for one Round permutation, corresponding to 24 rounds. Round module is implemented as in Figure 4.7

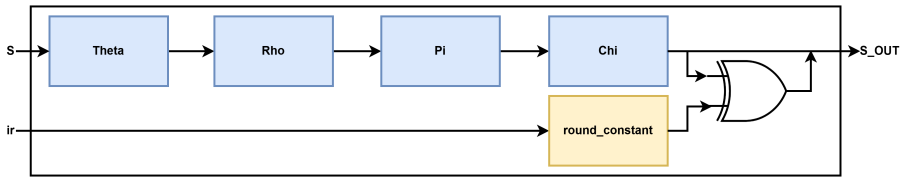

Figure 4.7: Round Module

`Iota` Permutation described in FIPS-202 [19] is converted into registers with corresponding values. From there, we instantiate look-up table named RC to get the round constant with round index parameter. The computational results of this thesis (as shown in Table 4.1) are entirely consistent with the appendix provided by Team Keccak [21].

RC	*Hex Value*	RC	*Hex Value*
RC[0]	*0x0000000000000001*	**RC[12]**	*0x000000008080808B*
RC[1]	*0x0000000000008082*	**RC[13]**	*0x0000000080000001*
RC[2]	*0x800000000000808A*	**RC[14]**	*0x8000000000008089*
RC[3]	*0x8000000080008000*	**RC[15]**	*0x8000000080808089*
RC[4]	*0x000000000000808B*	**RC[16]**	*0x0000000080008003*
RC[5]	*0x0000000080000001*	**RC[17]**	*0x8000000080800001*
RC[6]	*0x8000000000008008*	**RC[18]**	*0x0000000000008009*
RC[7]	*0x0000000080808081*	**RC[19]**	*0x800000008000808B*
RC[8]	*0x8000000080800001*	**RC[20]**	*0x000000008080000B*
RC[9]	*0x800000000000808B*	**RC[21]**	*0x8000000080808003*
RC[10]	*0x8000000080000001*	**RC[22]**	*0x8000000080808001*
RC[11]	*0x8000000080008009*	**RC[23]**	*0x8000000000008001*

Table 4.1: Round constants after computation based on FIPS-202 standard [19]

This approach stores and reuses precomputed constants across rounds, significantly reducing resource usage and computation time.

The KECCAK_p module interacts with external components using the following interface, as depicted in Figure 4.8.

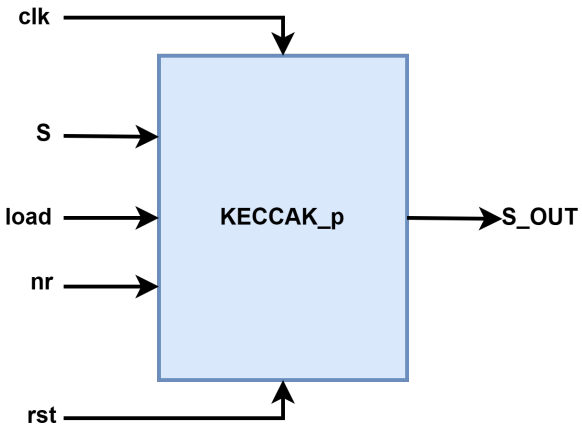

Figure 4.8: Interface of the KECCAK_p Module

- **Inputs:**
  - `clk`: The clock signal to synchronize operations.
  - `rst`: The reset signal to initialize the state of the module.
  - `S`: The input state to be processed by the permutation rounds.
  - `load`: A signal to indicate when to load the input state.
  - `nr`: The number of permutation rounds to apply.
- **Output:**
  - `S_OUT`: The resulting state after the specified number of permutation rounds.

The KECCAK_p module operates based on a finite state machine (FSM), which controls the module's internal operations. The FSM consists of four states: `IDLE`, `INITIALIZE`, `PROCESS`, and `FINALIZE`. The state transitions are shown in Figure 4.9, and the functionality of each state is described below:

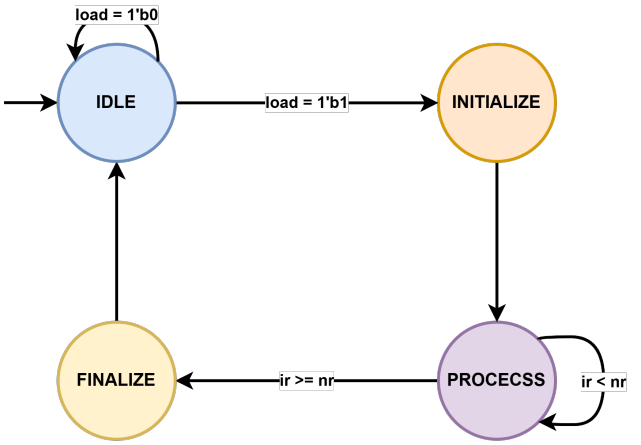

Figure 4.9: Finite State Machine of the KECCAK_p Module

1. **IDLE State:**
   - In this state, the module waits for the `load` signal to be asserted.
   - Once `load` is high, the FSM transitions to the `INITIALIZE` state.
2. **INITIALIZE State:**
   - The input state (`S`) is loaded into the working state (`A`).
   - The FSM then transitions to the `PROCESS` state.
3. **PROCESS State:**
   - The core permutation rounds are applied iteratively using the `Rnd` module.
   - A counter `ir` keeps track of the number of completed rounds.
   - When the specified number of rounds `nr` is reached, the FSM transitions to the `FINALIZE` state.
4. **FINALIZE State:**
   - The resulting state (`A_OUT`) is output through `S_OUT`.
   - The FSM resets to the `IDLE` state, ready for the next operation.

The use of multiple states allows for clear separation of tasks, including initialization, iterative processing, and finalization. By parameterizing the number of rounds (`nr`), the module supports flexible configurations suitable for different cryptographic primitives. By optimizing the Keccak core and leveraging

efficient resource management, the design achieves high performance, scalability, and robustness.

4.5.2. SHA-3 MODULES

The sponge construction is a fundamental algorithm used in cryptographic primitives like SHAKE and SHA3. The steps of the sponge construction based on *FIPS-202* standard [19] are as follows:

The FSM of the Hash modules maps directly to the sponge construction algorithm. Figure 4.10 shows the finite state machine, and Table 4.2 provides the mapping of states to the sponge construction steps. We implemented counters as shown in FSM to track the status of each state.

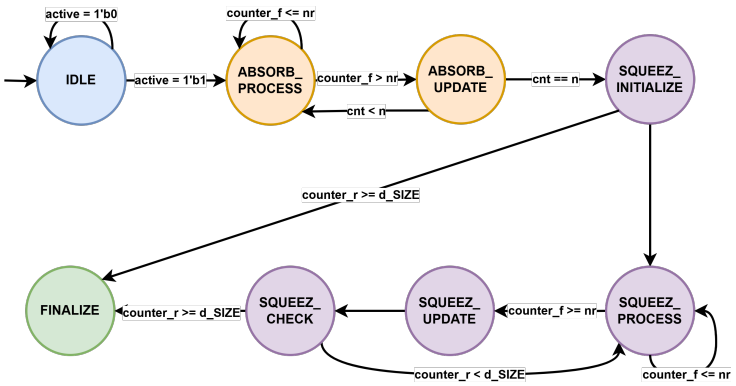

Figure 4.10: FSM of the Hash Modules

FSM State	Description
IDLE	Prepares input P, computes n, initializes variables.
ABSORB_PROCESS	Absorbs P_i into the state S via XOR and calls f.
ABSORB_UPDATE	Updates counters and moves to the next block P_i.
SQUEEZ_INITIALIZE	Initializes the squeezing phase and verifies $\|Z\|$.
SQUEEZ_PROCESS	Processes the state S with f to extract output.
SQUEEZ_UPDATE	Appends $\text{Trunc}_r(S)$ to Z.
SQUEEZ_CHECK	Verifies if the required output length d is met.
FINALIZE	Outputs $\text{Trunc}_d(Z)$ and asserts `finish`.

Table 4.2: Mapping of FSM States to Sponge Construction Steps

The Hash modules interface is shown in Figure 4.11.

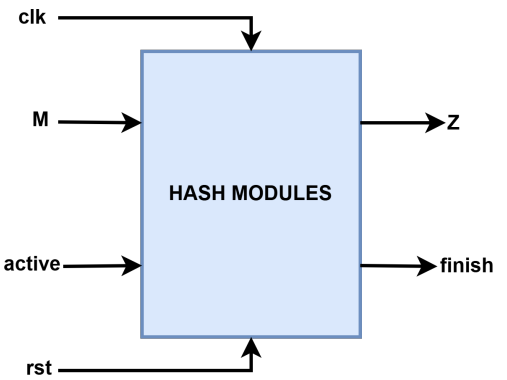

Figure 4.11: Interface of the Hash Modules

- **Inputs:**
  - `M`: Input message of parameterized size.
  - `active`: Activates the FSM to start processing.
  - `clk, rst`: Clock and reset signals.
- **Outputs:**
  - `finish`: Indicates the operation is complete.
  - `Z`: Output of parameterized size.

The Hash modules effectively implement the sponge construction as defined by NIST, ensuring compliance with established cryptographic standards [19]. The finite state machine (FSM) efficiently orchestrates the absorb and squeeze phases, guaranteeing accurate and secure operation for cryptographic processes.

The `SHA3_512 (G)` module plays a pivotal role in the architecture by serving as a pseudo-random function (PRF). Its primary objectives include:

- **Generating Pseudo-Random Outputs:** The module computes a deterministic yet pseudo-random 512-bit hash output using the SHA3-512 algorithm. This output is essential for cryptographic key generation, encapsulation, and decapsulation processes in CRYSTALS-Kyber.
- **Supporting Derived Data Streams:** The output is further utilized to generate input streams for critical components, such as the `CBD` (Centered Binomial Distribution) and `A_Gen` (Matrix Generator) modules. These streams ensure uniformity and enhance the security of subsequent cryptographic operations.

- **Streamlining High-Performance Workflow:** The hash outputs are fed directly into the CBD and A_Gen modules without delay, facilitating efficient data flow and minimizing latency within the encapsulation and decapsulation workflows.
- **Enhancing Cryptographic Security:** By adhering to the SHA3 standard, the SHA3_512 module provides a secure hashing mechanism, ensuring compliance with post-quantum cryptographic standards. This strengthens the architecture against potential vulnerabilities and attacks.

In the case of module SHAKE-256, an additional input n_num is introduced to enable support for the two computational modes corresponding to the two values of η, as detailed in subsequent sections.

4.6. UNIFORM SAMPLING

The A_generator module is a core component of the CRYSTALS-Kyber hardware implementation, designed to generate the polynomial coefficients required for cryptographic operations. Its primary function is to transform the output of the cryptographic hash function SHAKE_128 into polynomial coefficients and manage their storage in memory. In the Kyber cryptosystem, the matrices A and A^T are generated through this uniform sampling process, following the Parse algorithm theory.

The A_generator module performs the following tasks:

- Receives input parameters, including a seed and control signals, to initiate the coefficient generation process.
- Utilizes the output of the SHAKE_128 hash function to generate multiple coefficients by extracting and processing specific bit sequences.
- Implements a finite state machine (FSM) to manage the overall flow of operations, including initialization, processing, and completion phases.
- Outputs coefficients to memory, with proper address management and write control signals.

The Figure 4.12 shows the Interface of A_generator (or A_gen) Module.

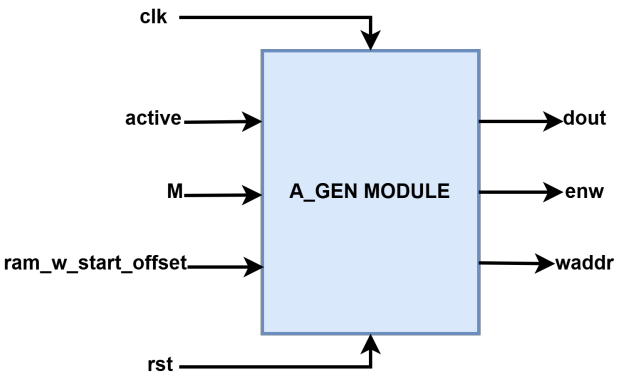

Figure 4.12: Interface of A_Gen Module

The `A_Gen` module interface is defined as follows:

- **Inputs:**
  - `clk`: Clock signal for synchronizing operations.
  - `rst`: Reset signal to initialize the module and clear internal registers.
  - `active`: Activation signal to initiate coefficient generation.
  - `M`: A 272-bit seed consisting of a 256-bit input and an additional 16 bits.
  - `ram_w_start_offset`: Memory address offset for writing the generated coefficients.
- **Outputs:**
  - `dout`: The output data containing eight coefficients, each of 12 bits.
  - `enw`: Write enable signal to store coefficients into memory.
  - `waddr`: Memory write address, incremented with each block of coefficients.

The internal operation of the `A_generator` module is governed by a finite state machine (FSM) illustrated in Figure 4.13.

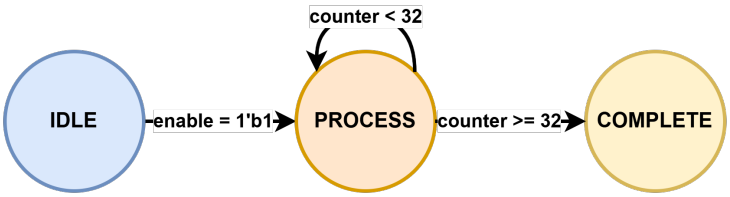

Figure 4.13: Sampling Finite State Machine

The FSM controls the coefficient generation through three main states:

- **IDLE:** Initializes internal registers and waits for the SHAKE_128 computation to complete.
- **PROCESS:** Iteratively processes the SHAKE_128 bitstream to generate blocks of coefficients for the **public matrix A**. These coefficients, extracted from specific bitstream portions, undergo logical operations and modular arithmetic to ensure they comply with CRYSTALS-Kyber parameter specifications. All values are adjusted to remain within the required numerical range before being stored in memory.
- **COMPLETE:** Signals the end of the computation process and resets control signals.

Upon invoking the XOF function, the Hash-Core hardware generates the input for the uniform sampling hardware. Each invocation of the XOF produces four independent 1344-bit input strings, enabling efficient and continuous coefficient generation for the matrix construction.

The Hash-Core outputs from the XOF are streamed directly into the uniform sampling hardware, ensuring a low-latency, high-throughput pipeline. This design minimizes buffering requirements and maximizes performance, aligning with the stringent efficiency needs of the CRYSTALS-Kyber algorithm.

4.7. CENTERED BINOMIAL DISTRIBUTION SAMPLING

The small_poly_generator module is designed to implement the Centered Binomial Distribution (CBD) sampling functionality required by the CRYSTALS-Kyber algorithm. It plays a crucial role in generating polynomial coefficients by processing pseudo-random data derived from the SHAKE_256 hash function. Below is a detailed explanation of the hardware design and operation.

The small_poly_generator module processes the output of the SHAKE_256 hash function to generate polynomial coefficients as follows:

- **Data Extraction:** Specific bit sequences are extracted from the output of the `SHAKE_256` function to serve as the source of randomness.
- **Coefficient Calculation:** The extracted bits are used to compute the coefficients a_i and b_i through modular arithmetic and bit accumulation. The computation method differs depending on whether the parameter η is set to 2 or 3, ensuring correct sampling behavior for different Kyber parameter sets.
- **Finite State Machine (FSM) Control:** A three-state FSM governs the overall control flow, managing initialization, coefficient generation, and completion stages.
- **Memory Write Operations:** Coefficients are output in blocks of eight, with careful management of write addresses. The design ensures that each generated coefficient adheres to the prescribed numerical range as defined by the CRYSTALS-Kyber specifications.

The `small_poly_generator` (or `CBD`) module, which is designed to efficiently generate binomial samples, optimizing data throughput and ensuring high performance. Its interface and internal architecture are illustrated in Figure 4.14.

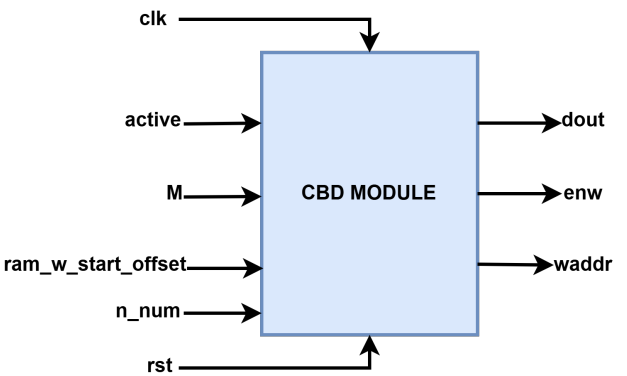

Figure 4.14: Interface of CBD Module

The module interface is defined as follows:

- **Inputs:**
  - `clk`: Clock signal to synchronize operations.

  - `rst`: Reset signal to initialize internal states.
  - `active`: Signal to activate the module for processing.
  - `M`: A 256-bit pseudo-random input seed.
  - `ram_w_start_offset`: Start offset address for coefficient writing.
  - `n_num`: Parameter indicating the mode ($\eta = 2$ or $\eta = 3$).

- **Outputs:**
  - `dout`: Generated polynomial coefficients.
  - `enw`: Enable signal for writing data into memory.
  - `waddr`: Address for memory write operations.

With similar FSM of **Uniform Sampling**, this module operates in three primary states:

- **IDLE:** Initializes all registers and waits for the `SHAKE_256` hash output to be ready.
- **PROCESS:** Iteratively generates polynomial coefficients for the **secret and error polynomials** s, e, r, e_1, and e_2 with parameter η. The process ensures all coefficients comply with CRYSTALS-Kyber parameter specifications.
- **COMPLETE:** Signals the end of the computation and resets control signals.

The hardware supports two operational modes, corresponding to the two possible values of η used in Kyber:

- For $\eta = 2$, the input length required is 1024 bits.
- For $\eta = 3$, the input length required is 1536 bits.

In case of $\eta = 2$, each invocation of the pseudo-random function (PRF), based on the `SHAKE256` hash function, generates an output of 1088 bits. To meet the requirement for $\eta = 3$, where 1536 bits are necessary, multiple outputs are concatenated and compressed appropriately. This approach ensures sufficient randomness and maintains efficient data throughput.

The Hash-Core outputs from the PRF are directly streamed into the CBD module without additional buffering, optimizing the computational pipeline and minimizing latency. By efficiently managing the bit extraction and accumulation processes, the CBD module ensures the correct statistical distribution of generated coefficients in compliance with the CRYSTALS-Kyber specifications.

4.8. CODER MODULE

Figure 4.15 illustrates the internal block diagram of the `coder` module, including RAM access and signal control.

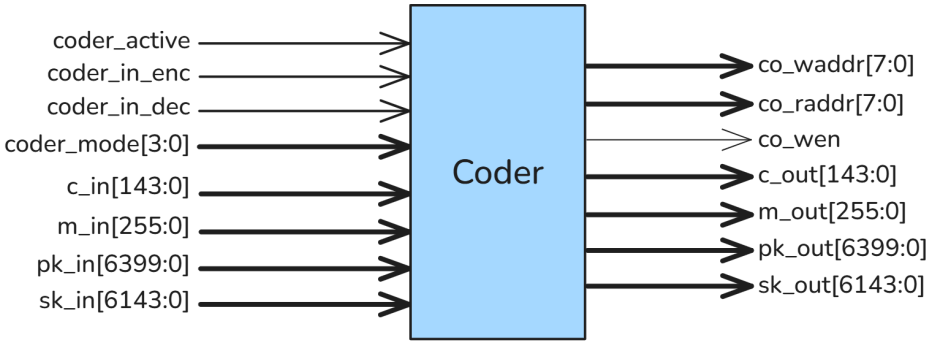

Figure 4.15: Coder module

The `coder` module handles operations for key generation, encryption, and decryption using different sub-modes. It interacts with external RAM for reading and writing data and performs compression/decompression on intermediate data during these operations.

Upon activation, the module transitions to the specified mode, as determined by the `mode` signal, and initiates processing controlled by an internal counter. Each mode governs specific operations, such as RAM read/write, data compression, or decompression. During key generation, encryption, or decryption, data is read from RAM, processed (e.g., compressed or encoded), and temporarily stored in internal registers. Once the operation for the current mode is completed, the processed data is either written back to RAM or assigned to the module's outputs. A `last_cycle` signal detects the final clock cycle of the mode's pre-defined processing period to ensure proper operation timing. This enables the module to correctly conclude the current operation before transitioning to the next state or returning to `WAIT`.

Table 4.3 describes the module's inputs and outputs:

Signal	Direction	Description
clk	Input	Clock signal
rst	Input	Reset signal (active high)
active	Input	Indicates active mode
load_input_Enc	Input	Load input for encryption
load_input_Dec	Input	Load input for decryption
mode [3:0]	Input	4-bit operation mode
pk_in [6399:0]	Input	Public key input
m_in [255:0]	Input	Message input
sk_in [6143:0]	Input	Secret key input
c_in [6143:0]	Input	Ciphertext input
rho_from_G [255:0]	Input	Random seed input from G
ram_rdata [95:0]	Input	Data read from RAM
pk_out [6399:0]	Output	Public key output
m_out [255:0]	Output	Message output
sk_out [6143:0]	Output	Secret key output
c_out [6143:0]	Output	Ciphertext output
rho_from_pk [255:0]	Output	Extracted random seed from public key
ram_wen	Output	RAM write enable
ram_raddr [7:0]	Output	RAM read address
ram_waddr [7:0]	Output	RAM write address
ram_wdata [95:0]	Output	Data to write into RAM

Table 4.3: Module Signals

The operational modes of the coder module are summarized in Table 4.3, covering different phases such as encoding and decoding. The module supports operations through various modes, controlled using a 4-bit input signal mode.

Mode	Name	Operation
0	**WAIT**	Idle mode; no active operations.
1	**KeyGen_encode_sk**	Secret key encoding during key generation
2	**KeyGen_encode_pk**	Public key encoding during key generation
3	**Enc_decode_pk**	Public key decoding during encryption
4	**Enc_decode_m**	Message decoding during encryption
5	**Enc_encode_c**	Ciphertext encoding during encryption
6	**Dec_decode_sk**	Secret key decoding during decryption
7	**Dec_decode_c**	Ciphertext decoding during decryption
8	**Dec_encode_m**	Message encoding during decryption

Table 4.4: Operational Modes of the `coder` Module

Each state defines a specific sequence of memory reads/writes, compression, decompression, or data assembly operations, tailored to its phase in the cryptographic workflow.

4.8.1. COMPRESS

The interface of the `compress` module is shown in Figure 4.16, detailing how input coefficients are scaled and reduced. The `compress` module is designed to compress a 12-bit input integer, scaling it within a reduced bit-width D. Below is the interface of this module:

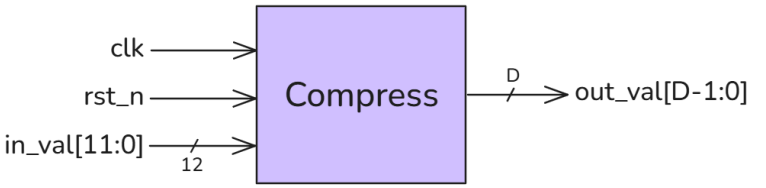

Figure 4.16: Compress module

- **Inputs:**
  - `clk`: Clock signal for synchronous operation.
  - `rst`: Asynchronous reset signal.
  - `d` [3:0]: Data type selector indicating compression level (not directly used inside this module, but relevant externally).
  - `in_data` [95:0]: Concatenated input bus containing eight 12-bit polynomial coefficients.

- **Outputs:**
  - `out_data_d1` [7:0]: 1-bit compressed output per coefficient (total 8 bits).
  - `out_data_d4` [31:0]: 4-bit compressed output per coefficient (total 32 bits).
  - `out_data_d10` [79:0]: 10-bit compressed output per coefficient (total 80 bits).

The `compress` module implements a scaled compression of the 12-bit input, `in_val`, into an output bit-width defined by `D`. This transformation involves scaling the input by 2^D and adding half of the modulus Q to ensure rounding, then dividing by Q to normalize the output. A modulo operation ensures the compressed result fits within `D` bits.

The module supports three compression formats:

- **1-bit Compression (`out_data_d1`):**
  - Each coefficient is compressed into a single bit.
  - If a coefficient lies between 832 and 2496 (exclusive), the corresponding output bit is set to 1; otherwise, it is set to 0.
  - This compression is optimized for extremely low-bandwidth encoding scenarios.

$$\texttt{out\_data\_d1}[i] = \begin{cases} 1 & \text{if } 832 < \texttt{in\_reg}[i] < 2496 \\ 0 & \text{otherwise} \end{cases}$$

- **4-bit Compression (`out_data_d4`):**
  - Each scaled coefficient is right-shifted by 19 bits, and the resulting 4-bit segment is extracted.
  - All eight compressed values are concatenated to form a 32-bit output word.

$$\texttt{out\_data\_d4} = [\texttt{mul\_out}[7][22:19], \texttt{mul\_out}[6][22:19], \ldots, \texttt{mul\_out}[0][22:19]]$$

- **10-bit Compression (`out_data_d10`):**
  - Each scaled coefficient is right-shifted by 13 bits, and the resulting 10-bit segment is extracted.
  - The eight compressed values are concatenated to form an 80-bit output word.

$$\texttt{out\_data\_d10} = [\texttt{mul\_out}[7][22:13], \texttt{mul\_out}[6][22:13], \ldots, \texttt{mul\_out}[0][22:13]]$$

4.8.2. DECOMPRESS

The interface of the `decompress` module is illustrated in Figure 4.17, showing how compressed coefficients are expanded. The `decompress` module is designed to reconstruct approximate 12-bit polynomial coefficients from compressed input data. It operates based on the compression factor d and reverses the effect of the corresponding `compress` module. The decompression supports three modes: very low precision ($d = 1$), medium precision ($d = 4$), and high precision ($d = 10$).

Figure 4.17: Decompress module

- **Inputs:**
  - `clk` (1 bit): Clock signal.
  - `rst` (1 bit): Active-high synchronous reset signal.
  - `d` (4 bits): Compression factor, selects the decompression mode (1, 4, or 10).
  - `in_data_d1` (8 bits): Compressed data for $d = 1$ mode.
  - `in_data_d4` (32 bits): Compressed data for $d = 4$ mode.
  - `in_data_d10` (80 bits): Compressed data for $d = 10$ mode.
- **Outputs:**
  - `out_data` (96 bits): Decompressed output data, containing eight 12-bit reconstructed coefficients concatenated together.

The `decompress` module uses a simple multiply and shift operation to approximate the original coefficients, balancing hardware efficiency and acceptable approximation error depending on the compression level d.

- If $d = 1$, each bit of `in_data_d1` is stored as a 1-bit value in `in_reg[i]`.
- If $d = 4$, `in_data_d4` is divided into eight 4-bit segments, each assigned to `in_reg[i]`.
- If $d = 10$, `in_data_d10` is divided into eight 10-bit segments, each assigned to `in_reg[i]`.

5

EVALUATION

5.1. EXPERIMENTAL METRICS

To evaluate the performance of the proposed implementation, several key metrics are collected and analyzed, as outlined below:

1. **Timing Measurement:** The time consumption for **Key Generation**, **Encryption**, and **Decryption** operations is measured in microseconds (μs) or seconds, depending on the execution platform.

2. **Throughput (TP):** The throughput is defined as the volume of data processed per unit of time and is calculated using the following expression:

$$TP = \frac{N_{Byte}}{T} \tag{5.1}$$

   where N_{Byte} represents the number of bytes processed, and T is the processing time in seconds.

3. **Frequency per Clock Cycle (Freq/CCs):** This metric indicates how efficiently the system operates per clock cycle and is defined as:

$$\text{Freq/CCs} = \frac{F}{C} \tag{5.2}$$

   where F is the maximum operating frequency (in MHz), and C is the number of clock cycles required for the operation. A higher Freq/CCs implies better performance.

4. **Area × Time:** This metric combines area and performance to indicate total hardware cost over execution time:

$$A \times T = N_{LUT} \times T \tag{5.3}$$

where T is the execution time (in milliseconds) and N_{LUT} is the number of LUTs. A smaller value implies higher hardware efficiency.

In addition to these performance indicators, the maximum operational Frequency (Mhz), Cycle Count (k), along with the number of LUTs (k), is also taken into account to assess design quality. These parameters are extracted from the synthesis reports provided by the FPGA design tools. The analysis prioritizes the evaluation and comparison of two key parameters: **Time** and **Throughput**. These metrics are emphasized due to their critical role in demonstrating the strengths and performance capabilities of the accelerator.

Throughout this section, the following notations are consistently used:

- F: Maximum operational frequency, measured in Megahertz (MHz).
- A: Hardware area, quantified as the number of Look-Up Tables (LUTs).
- TP: Throughput. It is expressed in megabytes per second (MB/s) when comparing against other FPGA-based implementations.
- Freq/CCs: Frequency per Clock Cycle, a unitless value indicating efficiency per cycle.
- $A \times T$: Area × Time, representing combined hardware and timing cost in LUT·ms.

The experimental procedures and evaluation tasks conducted in this study are as follows:

- A comparative analysis is performed between the software-based and hardware-accelerated implementations. This comparison considers encryption, decryption, and key generation execution time, as well as their associated throughput.
- The primary data collected directly from the experiments is time consumption. Other performance metrics such as Throughput, Freq/CCs, and Area × Time are derived using the equations defined in the previous subsection.

5.2. EXPERIMENTAL RESULT

In this section, we present the experimental results for each phase of the CRYSTALS-Kyber 512 algorithm using the 331KB real image shown in Figure 5.1.

Figure 5.1: Real image used for evaluating

After processing the image using the `image_to_byte` function in a **Jupyter Notebook** environment, we obtained a text string comprising 12,288 bytes. This byte stream was used as the input for the cryptographic system, which processed it through the following operations:

- **1 Key Generation operation**
- **48 Encryption operations**

- **48 Decryption operations**

Table 5.1 shows the measured performance of each phase in terms of data size, execution time, and computed throughput.

Mode	Byte Processed	Time (μs)	TP (MB/s)
Key.Gen.	32	27.47	1.11
Enc.	39,936	1721.28	22.1
Dec.	73,728	624.96	112.5

Table 5.1: Timing and Throughput evaluation

The results obtained from the hardware implementation were thoroughly validated through comprehensive testing, confirming the functional correctness and consistency of the design.

The hardware implementation of CRYSTALS-Kyber 512 demonstrates correct functionality and high efficiency, as shown by the high throughput, particularly in the decryption phase. These results validate the practical feasibility of deploying the design in real-time cryptographic applications.

5.3. PERFORMANCE COMPARISON

5.3.1. SOFTWARE

Table 5.2 presents the performance comparison between our proposed hardware implementation on the ZCU106 FPGA and existing software implementations on Intel Core i7-4770K (Haswell) processor [39] and ARM Cortex-M4 CPU of STM32F407Discovery [40]. The comparison includes key performance metrics such as cycle count, frequency per clock cycle (Freq/CCs), and total execution time for key generation, encryption, and decryption in the Kyber algorithm.

Parameters	This Work	[39]	[40]
Device	ZCU106	Intel Haswell	ARM Cortex-M4
Frequency (MHz)	100	3492	24
Key.Gen.			
Cycle Count (k)	2.7	122.7	499
Freq/CCs	37	28.5	0.05
Time (μs)	27.47	35.1	20,761,000,000
TP (MB/s)	37.3	28.5	4.8×10^{-8}
Enc.			
Cycle Count (k)	3.5	154.5	634
Freq/CCs	28.6	22.6	0.04
Time (μs)	35.86	44.2	26,417,000,000
TP (MB/s)	28.6	22.6	3.8×10^{-8}
Dec.			
Cycle Count (k)	1.3	187.9	597
Freq/CCs	76.9	18.6	0.04
Time (μs)	13.02	53.8	24,875,000,000
TP (MB/s)	78.6	18.6	4×10^{-8}

Table 5.2: Comparison with Kyber-512 Software Implementations

In general, our hardware implementation outperforms the software-based designs in terms of both execution time and processing efficiency. As shown in Table 5.2, the most significant improvement is observed in the decryption operation, where our design achieves a time of just 13.02 μs, compared to 53.8 μs on Intel Haswell [39] and approximately 24.9 seconds on ARM Cortex-M4 [40]. Similar trends can be seen in key generation and encryption, where the proposed system reaches 27.47 μs and 35.86 μs, respectively—both notably faster than Haswell and orders of magnitude better than Cortex-M4.

From Freq/CCs perspective, which reflects architectural efficiency, our implementation maintains a strong advantage. In particular, decryption reaches a Freq/CCs of 76.9, over 4 times higher than Haswell and nearly 2,000 times that of Cortex-M4. Across all operations, Freq/CCs values in our design consistently outperform the compared software platforms, indicating a well-optimized datapath and efficient resource scheduling.

From a throughput (TP) perspective, our hardware implementation demonstrates substantial performance gains over both software counterparts. For all three core operations—key generation, encryption, and decryption—our design achieves throughput values of 37.3 MB/s, 28.6 MB/s, and 78.6 MB/s, respectively. These numbers exceed those of the Intel Haswell implementation [39],

which reaches only 28.5 MB/s, 22.6 MB/s, and 18.6 MB/s, despite operating at a much higher clock frequency (3492 MHz). The disparity is even more striking when compared to the ARM Cortex-M4 [40], where throughput drops to the order of 10^{-8} MB/s, highlighting the severe limitations of low-power microcontrollers for cryptographic workloads. These results underscore the superior data-handling capacity of our design, making it not only faster in execution time but also far more effective in sustaining high-throughput post-quantum cryptographic operations.

Considering designs most comparable to ours, the implementation [39] provides a useful baseline for evaluating software-optimized high-frequency CPUs. Despite its 35 times higher clock frequency, Haswell only marginally outperforms or is even slower than our ZCU106 design in some operations. On the other hand, the Cortex-M4-based design [40] highlights the limitations of low-frequency microcontrollers for post-quantum cryptographic workloads, with timing results up to six orders of magnitude slower than ours.

These results clearly indicate that our design significantly outperforms the software counterparts in both speed and computational efficiency. The dramatic reduction in execution time and increase in processing throughput make this design highly suitable for real-time and resource-constrained environments.

5.3.2. HARDWARE

Table 5.3 summarizes the performance comparison of our Kyber-512 hardware implementation on the ZCU106 board against other FPGA-based implementations on Artix 7 and XCZU7EV devices [3], [41], [42], [43]. The analysis focuses on key metrics, including execution time, throughput, resource utilization, operating frequency, cycle count, internal efficiency (Freq/CCs), and the Area × Time product, with particular emphasis on execution time and throughput due to their critical role in demonstrating the accelerator's efficiency.

Parameters	This Work	[3]	[41]	[42]	[43]
FPGA Device	ZCU106		Artix 7		Virtex 7
LUTs (k)	151.2	7.4	9	18	1,978
FFs (k)	54.8	4.6	9.2	26.4	194
DSPs	21	2	6	6	0
BRAMs	0	3	10.5	15	0
Frequency (MHz)	100	161	204	115	67
Cycle Count (k)	7.6	15.6	10.6	21	75
Freq/CCs	13.1	10.3	19.2	5.5	0.9
Time (μs)	76.35	95.2	52,300,000	148	1,119
TP (MB/s)	13.4	10.8	1.96×10^{-5}	6.9	0.9
$A \times T$ **(LUTs·ms)**	11.5	0.7	470,700	2.7	148,350

Table 5.3: Comparison of Kyber-512 FPGA Implementations

Our design achieves an execution time of 76.35 μs, which is highly competitive and significantly outperforms the impractical 52.3 seconds reported by [41] and the 1119 μs of [43]. Compared to [3] with 95.2 μs, our implementation is approximately 20% faster, despite operating at a lower clock frequency of 100 MHz versus 161 MHz. Although [42] reports a longer execution time of 148 μs, our design's superior latency makes it more suitable for applications requiring rapid response times.

This efficiency extends to throughput, a vital metric for assessing data processing capabilities in cryptographic accelerators. Our design delivers a throughput of 13.4 MB/s, surpassing [3] (10.8 MB/s) by approximately 24%, [42] (6.9 MB/s) by nearly 94%, and [43] (0.9 MB/s) by over 14 times. The throughput of [41] is negligible at 20.6 B/s, rendering it unsuitable for practical applications. By achieving this high throughput at a modest 100 MHz, our implementation demonstrates robust data handling without relying on high-frequency optimizations, making it well-suited for systems where predictability and compatibility with standard clock domains are prioritized.

The internal efficiency, measured as Frequency per Clock Cycle (Freq/CCs), further highlights our design's strengths. With a Freq/CCs of 13.1, our implementation outperforms [3] (10.3), [42] (5.5), and [43] (0.9), though it is lower than [41] (19.2), which benefits from a higher 204 MHz frequency. This efficiency, achieved at a conservative 100 MHz, reflects a balanced approach that prioritizes ease of timing closure and portability across FPGA platforms.

In terms of resource utilization, our design consumes 151.2k LUTs, 54.8k FFs, 21 DSPs, and no BRAMs on the ZCU106. While this is higher than [3]

(7.4k LUTs, 4.6k FFs, 2 DSPs, 3 BRAMs), [41] (9k LUTs, 9.2k FFs, 6 DSPs, 10.5 BRAMs), and [42] (18k LUTs, 26.4k FFs, 6 DSPs, 15 BRAMs), it is significantly lower than [43] (1,978k LUTs, 194k FFs, 0 DSPs), which employs High-level Synthesis (HLS). Although our design's resource consumption is higher than some other implementations, this is justified as it prioritizes optimization for speed and throughput, aligning with the goals of an accelerator. The absence of BRAM usage simplifies integration and reduces latency, while the modest DSP count ensures efficient arithmetic operations without excessive resource demands.

A key measure of hardware efficiency is the Area × Time product (LUTs·ms), which combines resource utilization and execution time. Our design achieves an Area × Time of 11.5 LUTs·ms, a substantial improvement over [41] (470,700 LUTs·ms) and [43] (148,350 LUTs·ms). While higher than [3] (0.7 LUTs·ms) and [42] (2.7 LUTs·ms), this trade-off is justified by our design's superior execution time and throughput. The conservative 100 MHz operating frequency facilitates easier timing closure compared to the higher frequencies of [3] (161 MHz), [41] (204 MHz), and [42] (115 MHz), enhancing portability and reducing design complexity. Unlike [3], [41], and [42], which rely on BRAMs, our design's BRAM-free approach minimizes memory-related latencies, further improving its suitability for real-world deployment.

Overall, our Kyber-512 implementation on the ZCU106 platform achieves a highly effective balance between performance and practicality. With an execution time of 76.35 μs and throughput of 13.4 MB/s, it significantly outperforms the other implementations as an accelerator. By operating at 100 MHz and avoiding BRAM usage, our design ensures ease of integration, timing closure, and compatibility with mainstream FPGA platforms. This makes it an ideal choice for high-assurance post-quantum cryptographic applications where performance, resource efficiency, and practical deployment considerations are paramount.

6

CONCLUSION

This thesis has presented a comprehensive hardware implementation of the **CRYSTALS-Kyber** post-quantum cryptographic scheme on the **Xilinx ZCU106** System-on-Chip (SoC) Field-Programmable Gate Array (FPGA) platform. Through a meticulous hardware/software co-design methodology, critical components, including the Number Theoretic Transform (NTT), cryptographic hash functions, and encoding/decoding modules, were optimized and seamlessly integrated using an AXI-based memory-mapped control interface. The resulting architecture successfully supports the core functionalities of **Key Generation**, **Encryption**, and **Decryption**. This implementation represents a significant step toward realizing efficient, secure, and scalable post-quantum cryptographic systems suitable for real-world applications.

The contributions of this work lie not only in the successful deployment of the CRYSTALS-Kyber algorithm but also in the demonstration of a modular and extensible design framework that leverages the computational capabilities of modern FPGA platforms. By addressing the challenges of post-quantum cryptography in resource-constrained environments, this thesis provides valuable insights into the practical deployment of next-generation cryptographic accelerators.

6.1. ADVANTAGES OF THE PROPOSED IMPLEMENTATION

The implementation of the CRYSTALS-Kyber algorithm on the Xilinx ZCU106 SoC-FPGA platform has yielded several significant advantages, underscoring its potential for practical deployment in secure computing environments:

- **Superior Computational Efficiency:** The proposed architecture demonstrates exceptional performance, particularly in the decryption phase,

achieving a throughput of 13.4 MB/s and a total execution time of 76.35 μs. These metrics reflect substantial performance improvements over comparable implementations, enabling high-speed cryptographic operations critical for time-sensitive applications such as secure communication protocols and real-time data processing.

- **Scalability and Modularity:** The design adopts a highly modular architecture, with independent components such as the NTT core, cryptographic hash units, and encoding/decoding modules. This modularity facilitates the reuse and reconfiguration of individual components for other cryptographic protocols, enhancing the system's adaptability to future cryptographic standards or hybrid schemes that combine classical and post-quantum algorithms.

- **Suitability for Edge and Embedded Systems:** The successful deployment on the Xilinx ZCU106 platform validates the feasibility of implementing high-throughput post-quantum cryptographic schemes in resource-constrained environments, such as edge-computing devices and embedded systems. This capability is particularly relevant for applications in the Internet of Things (IoT), autonomous systems, and other domains requiring robust security with limited computational resources.

These advantages collectively highlight the proposed implementation's potential to address the performance and scalability demands of post-quantum cryptography, paving the way for its adoption in diverse application domains.

6.2. LIMITATIONS OF THE CURRENT DESIGN

Despite its strengths, the proposed implementation exhibits certain limitations that warrant consideration for future enhancements:

- **High Resource Utilization:** The design consumes a substantial number of Look-Up Tables (LUTs), totaling 171.6k, which may pose challenges for deployment on smaller or more resource-constrained FPGA platforms. This high resource footprint could limit its applicability in ultra-low-power or cost-sensitive applications, necessitating further optimization to achieve a more compact design.

- **Limited Support for Higher-Security Variants:** The current implementation focuses exclusively on the Kyber-512 parameter set, which provides a baseline level of security. However, it does not support higher-security

variants such as Kyber-768 or Kyber-1024, which are essential for applications requiring enhanced cryptographic strength. This limitation restricts the system's versatility in addressing diverse security requirements.

- **Suboptimal Communication Performance:** The implementation relies on the AXI4 memory-mapped protocol for data transfers, which, while effective for general-purpose communication, does not fully meet the demands of high-speed data transfer applications, such as real-time video streaming or large-scale data processing. This limitation highlights the need for more advanced communication protocols to support bandwidth-intensive use cases.

These limitations provide a clear roadmap for future improvements, ensuring that the proposed architecture can achieve broader applicability and enhanced security in practical deployments.

6.3. FUTURE RESEARCH DIRECTIONS

Building on the findings and limitations of this work, several promising directions for future research and development are proposed to enhance the performance, security, and applicability of the CRYSTALS-Kyber implementation:

- **Optimization of Resource Utilization:** Future efforts should focus on resource-sharing techniques, such as time-multiplexing of computational units, and compression strategies to reduce the FPGA resource footprint. These optimizations would enable the deployment of the accelerator on smaller, power-constrained devices, making it suitable for low-power IoT and edge-computing applications where high performance and minimal resource usage are paramount.

- **Support for Higher-Security Variants:** Extending the architecture to accommodate Kyber-768 and Kyber-1024 parameter sets would enhance the system's cryptographic resilience, enabling it to meet the security requirements of a wider range of applications. This extension would involve redesigning critical components, such as the NTT core, to handle larger polynomial degrees while maintaining acceptable performance levels.

- **Enhanced Communication Mechanisms:** Transitioning from the AXI4 memory-mapped protocol to the AXI4-Stream protocol, coupled with the integration of a Direct Memory Access Controller (DMAC), would substantially improve data transfer rates. Such advancements would enable

the system to support high-bandwidth applications, such as real-time video streaming or large-scale cryptographic processing, thereby broadening its practical utility.

- **Porting to Application-Specific Integrated Circuits (ASICs):** Adapting the design for ASIC implementation could yield significant improvements in power efficiency and area utilization, making it suitable for commercial deployment in resource-constrained environments. This transition would require careful optimization of the architecture to balance performance, power, and cost considerations.

These proposed directions aim to address the current limitations while leveraging the strengths of the proposed implementation, ensuring its relevance and impact in the evolving field of post-quantum cryptography.

In conclusion, this thesis has demonstrated the feasibility and efficacy of implementing the CRYSTALS-Kyber post-quantum cryptographic scheme on an FPGA-based platform, achieving notable performance and modularity. While certain limitations remain, the proposed architecture lays a robust foundation for future advancements in secure, high-performance cryptographic systems. By addressing the identified challenges and pursuing the outlined research directions, this work can contribute significantly to the development of scalable and resilient post-quantum cryptographic solutions for next-generation computing environments.

BIBLIOGRAPHY

[1] Avanzi, R., Bos, J., Ducas, L., Kiltz, E., Lepoint, T., Lyubashevsky, V., Schanck, J. M., Schwabe, P., Seiler, G., & Stehlé, D. (2017). CRYSTALS-Kyber Algorithm Specifications and supporting documentation. Available: https://csrc.nist.gov/Projects/post-quantum-cryptography/round-2-submissions

[2] Nguyen, H., & Tran, L. (2022). Design of Polynomial NTT and INTT Accelerator for Post-Quantum Cryptography CRYSTALS-Kyber. Arabian Journal for Science and Engineering, 48(2), 1527–1536. https://doi.org/10.1007/s13369-022-06928-w

[3] Xing, Y., & Li, S. (2021). A compact hardware implementation of CCA-Secure Key Exchange Mechanism CRYSTALS-KYBER on FPGA. IACR Transactions on Cryptographic Hardware and Embedded Systems, 328–356. https://doi.org/10.46586/tches.v2021.i2.328-356

[4] He, S., Li, H., Li, F., & Ma, R. (2024). A lightweight hardware implementation of CRYSTALS-Kyber. Journal of Information and Intelligence, 2(2), 167–176. https://doi.org/10.1016/j.jiixd.2024.02.004

[5] Huang, Y., Huang, M., Lei, Z., & Wu, J. (2020). A pure hardware implementation of CRYSTALS-KYBER PQC algorithm through resource reuse. IEICE Electronics Express, 17(17), 20200234. https://doi.org/10.1587/elex.17.20200234

[6] Nguyen, T. T., Kim, S., Eom, Y., & Lee, H. (2022). Area-Time efficient hardware architecture for CRYSTALS-Kyber. Applied Sciences, 12(11), 5305. https://doi.org/10.3390/app12115305

[7] Guo, W., Li, S., & Kong, L. (2021). An efficient implementation of KYBER. IEEE Transactions on Circuits & Systems II Express Briefs, 69(3), 1562–1566. https://doi.org/10.1109/tcsii.2021.3103184

[8] Ni, Z., Khalid, A., Kundi, D., O'Neill, M., & Liu, W. (2023). HPKA: A High-Performance CRYSTALS-Kyber Accelerator exploring efficient pipelining. IEEE Transactions on Computers, 72(12), 3340–3353. https://doi.org/10.1109/tc.2023.3296899

[9] Nannipieri, P., Di Matteo, S., Zulberti, L., Albicocchi, F., Saponara, S., & Fanucci, L. (2021). A RISC-V post Quantum Cryptography instruction set extension for number theoretic transform to Speed-Up CRYSTALS algorithms. IEEE Access, 9, 150798–150808. https://doi.org/10.1109/access.2021.3126208

[10] Nguyen, D. N., Tran, V. D., Pham, H. L., Le, V. T. D., Lam, D. K., Tran, T. H., & Nakashima, Y. (2024). HyperNTT: A Fast and Accurate NTT/INTT Accelerator with Multi-Level Pipelining and an Improved K2-RED Module. 2024 International Technical Conference on Circuits/Systems, Computers, and Communications (ITC-CSCC), 1–6. https://doi.org/10.1109/itc-cscc62988.2024.10628429

[11] Lyubashevsky, V., Micciancio, D., Peikert, C., & Rosen, A. (2008). SWIFFT: A modest proposal for FFT hashing. In Fast Software Encryption (pp. 54–72). Berlin, Germany: Springer.

[12] Zhang, N., et al. (2020). Highly efficient architecture of NewHope-NIST on FPGA using low complexity NTT/INTT. IACR Transactions on Cryptographic Hardware and Embedded Systems, 49-72.

[13] FPGAEmu Documentation. *AXI Protocol Overview*. Available at: `https://fpgaemu.readthedocs.io/en/latest/axi.html`. Accessed April 27, 2025.

[14] Nguyn Quân. *BUS – Bài 1: Giao thc AMBA AXI*. Blogspot, August 2018. Available at: `https://nguyenquanicd.blogspot.com/2018/08/busbai-1-giao-thuc-amba-axi.html`. Accessed April 27, 2025.

[15] Xilinx, Inc., "ZCU106 Evaluation Board User Guide," UG1244, v1.4, October 23, 2019.

[16] Xilinx, Inc., "Zynq UltraScale+ MPSoC ZCU106 Evaluation Kit," `https://www.xilinx.com/products/boards-and-kits/zcu106.html#information`, Accessed: April 26, 2025.

[17] Reducible. (2021, Nov 14). The Fast Fourier Transform (FFT): Most Ingenious Algorithm Ever? [Video]. YouTube. https://www.youtube.com/watch?v=h7apO7q16V0

[18] Trinh, N., Le, A., Nguyen, H., & Tran, L. (2021). Algorithmic TCAM on FPGA with data collision approach. Indonesian Journal of Electrical Engineering and Computer Science, 22(1), 89-96.

[19] FIPS-202, "SHA-3 Standard: Permutation-Based Hash and Extendable-Output Functions," NIST, 2015.

[20] G. Bertoni, et al., "The Keccak Reference," 2020. [Online]. Available: http://keccak.noekeon.org/

[21] Team Keccak, *Keccak Specifications Summary*, https://keccak.team/keccak_specs_summary.html, accessed December 8, 2024.

[22] CRYSTALS-Kyber Team, *CRYSTALS-Kyber Software Implementations*, https://pq-crystals.org/kyber/software.shtml, accessed December 10, 2024.

[23] National Institute of Standards and Technology, *NIST PQC Round 3 Submissions*, https://csrc.nist.gov/projects/post-quantum-cryptography/round-3-submissions, accessed December 10, 2024.

[24] National Institute of Standards and Technology, *CRYSTALS-Kyber Algorithm Specifications*, https://csrc.nist.gov/, accessed December 10, 2024.

[25] Xilinx Inc., *Vivado Design Suite User Guide*, https://www.xilinx.com/support/documentation/sw_manuals/xilinx2021_2/ug973-vivado-release-notes-install-license.pdf, accessed December 10, 2024.

[26] Xilinx Inc., *Vivado Design Suite HLx Editions*, https://www.xilinx.com/support/documentation/sw_manuals/xilinx2021_2/ug902-vivado-architecture.pdf, accessed December 10, 2024.

[27] Xilinx Inc., *Kria KV260 Vision AI Starter Kit Documentation*, https://www.xilinx.com/products/boards-and-kits/kria/kv260.html, accessed December 10, 2024.

[28] Xilinx Inc., *Kria SOM Overview*, https://www.xilinx.com/products/som/kria.html, accessed December 10, 2024.

[29] Xilinx, "KRIA KV260 Vision AI Starter Kit Product Guide," 2022. [Online]. Available: https://www.xilinx.com/products/boards-and-kits/kria/kv260.html

[30] AMD Xilinx, "Kria K26 SoM Overview," [Online]. Available: https://www.xilinx.com/products/som/kria/k26.html

[31] AMD Xilinx, "Zynq UltraScale+ MPSoC Technical Reference Manual," [Online]. Available: https://www.xilinx.com/products/silicon-devices/soc/zynq-ultrascale-mpsoc.html

[32] Xilinx, "Vision AI Starter Kit User Guide," [Online]. Available: https://www.xilinx.com/products/som/kria/kv260/getting-started.html

[33] Xilinx, "Zynq UltraScale+ MPSoC Integration Guide," 2022. [Online]. Available: https://www.xilinx.com

[34] Xilinx, "AXI Reference Guide," [Online]. Available: https://www.xilinx.com

[35] N. Zhang, Y. Wang, "Efficient hardware accelerators for CRYSTALS-Kyber and its application in post-quantum cryptography," *IEEE Transactions on Circuits and Systems*, vol. 67, pp. 1543-1551, 2023.

[36] Xilinx, "Vivado Design Suite User Guide: High-Level Synthesis," 2021. [Online]. Available: https://www.xilinx.com

[37] L. Wan, F. Zheng, G. Fan, et al., "A novel high-performance implementation of CRYSTALS-Kyber with AI accelerator," *IACR Cryptology ePrint Archive*, vol. 2022, pp. 881, 2022.

[38] A. Ravi, M. R. Islam, and T. Muhlbaier, "Implementation and Evaluation of Post-Quantum Cryptography on FPGA for Image Processing Applications," *ACM Transactions on Reconfigurable Technology and Systems*, vol. 14, no. 2, pp. 1–18, 2021.

[39] 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 (version 3.01)*. January 31, 2021. Available at: https://pq-crystals.org/kyber/data/kyber-specification-round3-20210131.pdf

[40] Leon Botros, Matthias J. Kannwischer, and Peter Schwabe. *Memory-Efficient High-Speed Implementation of Kyber on Cortex-M4*. The Third PQC Standardization Conference, 2021.

[41] Qingru Zeng, Quanxin Li, Baoze Zhao, Han Jiao, and Yihua Huang (2022). Hardware Design and Implementation of Post-Quantum Cryptography Kyber. *2022 IEEE High Performance Ex-*

treme Computing Conference (HPEC), Waltham, MA, USA, pp. 1–6. https://doi.org/10.1109/HPEC55821.2022.9926344

[42] Bisheh-Niasar, Mojtaba and Azarderakhsh, Reza and Mozaffari-Kermani, Mehran (2021). Instruction-Set Accelerated Implementation of CRYSTALS-Kyber. *IEEE Transactions on Circuits and Systems I: Regular Papers*, vol. 68, no. 11, pp. 4648–4659. https://doi.org/10.1109/TCSI.2021.3106639

[43] Kanad Basu and Deepraj Soni and Mohammed Nabeel and Ramesh Karri (2019). NIST Post-Quantum Cryptography–A Hardware Evaluation Study. *Cryptology ePrint Archive, Paper 2019/047*. https://eprint.iacr.org/2019/047