PONTIFICAL CATHOLIC UNIVERSITY OF RIO GRANDE DO SUL
SCHOOL OF TECHNOLOGY
COMPUTER ENGINEERING

HARDWARE ACCELERATION OF THE POST-QUANTUM ML-KEM ALGORITHM ON RISC-V PROCESSORS

VITOR BALBINOT ZANINI

Bachelor Thesis submitted to the Pontifical Catholic University of Rio Grande do Sul in partial fulfillment of the requirements for the degree of Bachelor in Computer Engineering.

Advisor: Prof. Fernando Gehm Moraes

Porto Alegre
2026

ACKNOWLEDGMENTS

I would like to use this space to express my deepest gratitude towards the people who made this work possible.

First of all, I thank my advisor, Prof. Fernando Moraes, for the incredible guidance he offered. Thank you for taking me into this project, and for all the help and patience during this last year. I know it wasn't easy at times, and I hugely appreciate the support you've been giving me.

I also extend my sincere thanks to Prof. Iaçanã Ianiski Weber for the teachings in the course 98G08-04 - Confiabilidade e Segurança de Software. The excellent theoretical foundation built during your classes was absolutely essential for me to fully grasp the highly complex algorithms required for this research.

I also thank my family, especially my parents, Marildo and Siloé, and my sister, Natália, for the constant support they gave me during my graduation. Without them, none of this would've been possible in the first place.

Finally, I thank FAPERGS for financing my participation in this research, thus enabling me to pursue this work.

ACELERAÇÃO EM HARDWARE DO ALGORITMO PÓS-QUÂNTICO ML-KEM EM PROCESSADORES RISC-V

RESUMO

O avanço da computação quântica representa uma ameaça direta aos algoritmos criptográficos tradicionais. Para garantir a segurança futura, dispositivos de borda (*edge computing*) e Internet das Coisas (IoT) precisam migrar para padrões de criptografia pós-quântica (PQC), como o algoritmo ML-KEM. No entanto, a alta complexidade computacional desses novos padrões sobrecarrega processadores embarcados limitados por restrições de energia e área. Este trabalho aborda esse desafio propondo a integração de aceleradores escalares dedicados como extensões do conjunto de instruções (ISA) em um processador RISC-V (RS5). O projeto foca na aceleração em hardware das operações mais custosas do ML-KEM: a multiplicação polinomial no domínio da Transformada Numérica de Teoria (NTT) e a compressão de dados. A arquitetura foi avaliada através de prototipagem em FPGA e síntese ASIC utilizando a tecnologia de 28nm da TSMC. A análise da implementação física revelou que abordagens puramente combinacionais para a aritmética pós-quântica degradam o roteamento e violam o fechamento de tempo (*timing closure*). Para solucionar essa limitação, adotou-se uma arquitetura estruturalmente serializada, isolando a lógica criptográfica através de fronteiras de registradores e máquinas de estados. Essa estratégia consolidou um compromisso área-desempenho (*trade-off*) favorável: a integração impôs custos de 5,8% de aumento na área de silício e 3,4% na dissipação de potência típica. Em contrapartida, a extensão acelerou a execução e entregou uma redução de 45,1% na latência de decapsulação do ML-KEM, o que diminuiu o consumo total de energia em 43,3%. Esse compromisso arquitetural permitiu que o núcleo sustentasse a restrição de 250 MHz sob piores condições operacionais (*worst-case*), validando uma implementação física com eficiência energética otimizada.

Palavras-Chave: Criptografia pós-quântica, ML-KEM, RISC-V, aceleração em hardware, ASIC 28nm, compromisso área-desempenho.

HARDWARE ACCELERATION OF THE POST-QUANTUM ML-KEM ALGORITHM ON RISC-V PROCESSORS

ABSTRACT

The advancement of quantum computing poses a direct threat to traditional cryptographic algorithms. To ensure future security, edge computing and Internet of Things (IoT) devices must migrate to post-quantum cryptography (PQC) standards, such as the ML-KEM algorithm. However, the computational complexity of these new standards burdens embedded processors constrained by power and area limits. This work addresses this challenge by proposing the integration of dedicated custom scalar accelerators as Instruction Set Architecture (ISA) extensions into an RS5 RISC-V processor. The design focuses on the hardware acceleration of the most computationally expensive ML-KEM operations: Number Theoretic Transform (NTT) polynomial multiplication and data compression. The architecture was evaluated through FPGA prototyping and ASIC synthesis targeting a TSMC 28nm standard-cell library. The physical implementation analysis revealed that purely combinational approaches for post-quantum arithmetic degrade global routing and violate timing closure. To resolve this bottleneck, a structurally serialized architecture was adopted, isolating the cryptographic logic through register boundaries and state machines. This strategy established a clear area-performance trade-off: the integration incurred overheads of a 5.8% increase in silicon footprint and a 3.4% rise in typical power dissipation. In return, the custom extension achieved a 45.1% reduction in ML-KEM decapsulation latency and decreased total energy consumption by 43.3%. This architectural compromise allowed the core to sustain the 250 MHz constraint under worst-case operating conditions, validating a physical implementation with optimized efficiency.

Keywords: Post-quantum cryptography, ML-KEM, RISC-V, hardware acceleration, 28nm ASIC, area-performance trade-off.

LIST OF FIGURES

LIST OF TABLES

LIST OF ALGORITHMS

CONTENTS

1. INTRODUCTION

Cryptography plays a fundamental role in modern communication by ensuring that sensitive information remains inaccessible to unauthorized parties. Throughout the development of cryptographic systems, standardization organizations and researchers have defined a set of security objectives, often referred to as security services [Avižienis et al., 2004]. These services establish the foundational requirements that algorithms and their respective hardware or software implementations must guarantee.

While specific mechanisms focus on particular subsets of these goals, a comprehensive security system is generally designed to address the following fundamental objectives:

- **Confidentiality:** Information is kept secret from all but authorized entities.
- **Integrity:** The prevention of modification or destruction of an asset by an unauthorized user, guaranteeing that messages have not been altered in transit.
- **Message Authentication:** Verifies the authenticity of the sender, also known as data origin authentication.
- **Nonrepudiation:** Ensures that the creator or sender of a message cannot falsely deny their actions.

By fulfilling these objectives, cryptographic systems provide a robust and reliable layer of security. This level of confidence is essential for protecting data across all domains, safeguarding personal privacy, corporate assets, and confidential government information.

The rapid advancement of quantum computing poses a critical threat to the security and reliability of modern digital systems [Thamilarasi et al., 2024]. Traditional asymmetric encryption schemes, such as RSA and Elliptic Curve Cryptography (ECC), rely on mathematical problems that will be easily broken by quantum computers [Tiwari et al., 2024]. Consequently, there is a need to develop and deploy new cryptographic paradigms to ensure that sensitive data remains accessible only to authenticated entities in the post-quantum era.

In response to this threat, the National Institute of Standards and Technology (NIST) initiated the Post-Quantum Cryptography (PQC) Standardization Process [NIST, 2024b]. Following an evaluation of candidate algorithms, NIST selected a set of mechanisms resilient to quantum attacks. For public-key encryption (PKE) and secure Key Encapsulation Mechanisms (KEM), essential for establishing symmetric keys between communicating parties, the Kyber algorithm [Avanzi et al., 2021], officially standardized as ML-KEM [NIST, 2024a], was chosen as the primary standard.

1.1 Motivation

The advent of large-scale quantum computing poses an existential threat to modern cryptographic infrastructure. Shor's algorithm demonstrates that widely deployed public-key cryptosystems, such as RSA and Elliptic Curve Cryptography (ECC), can be broken in polynomial time. Furthermore, Grover's algorithm significantly reduces the effective security of symmetric ciphers, necessitating larger key sizes to maintain adequate protection [Yılmaz and Özbudak, 2026]. While fault-tolerant quantum computers capable of executing these algorithms are not yet available, the cryptographic community faces an immediate vulnerability known as the "Harvest Now, Decrypt Later" (HNDL) strategy. In this attack model, adversaries intercept and store encrypted sensitive data today, intending to decrypt it in the future once quantum capabilities are realized [Kagai et al., 2025]. This imminent threat necessitates an urgent transition to Post-Quantum Cryptography (PQC) standards.

To mitigate this vulnerability, algorithms like Kyber (standardized by NIST as ML-KEM) have been developed. While Kyber provides robust post-quantum security guarantees, its computational complexity introduces performance challenges, particularly in resource-constrained environments [Alnaseri et al., 2025]. Profiling of the Kyber algorithm reveals that the Number Theoretic Transform (NTT) operation is the primary computational bottleneck, accounting for a substantial portion of the overall execution time [Kannwischer et al., 2019]. Therefore, optimizing the Kyber workflow is necessary for its deployment in embedded systems.

Over recent years, several studies have focused on accelerating Kyber operations. Although software-level optimizations can yield performance improvements [Zhang et al., 2024], they are limited by the underlying general-purpose processor architecture. Dedicated hardware acceleration, conversely, offers efficiency. By offloading operations to custom hardware modules, hardware implementations may outperform software-based approaches [Dinh et al., 2024; Yaman et al., 2021], delivering gains in both processing speed and energy efficiency.

Building upon this premise, this undergraduate thesis aims to enable secure information exchange between a System-on-Chip (SoC) and an external host device. This is achieved by introducing hardware accelerators not only for the Kyber workflow but also for the symmetric cryptographic primitives underlying the key encapsulation scheme. To this end, this work integrates a custom RISC-V instruction set extension, named Xkyber [Gewehr et al., 2024], designed specifically to accelerate core Kyber operations. Furthermore, it incorporates the ZKNH and ZBKB standard extensions to accelerate the symmetric operations of SHA-2 and SHA-3, respectively. All hardware modifications were implemented within the RS5 processor, the processor core of the WiMed project, paving the way for these contributions to be fully integrated into the complete SoC architecture.

1.2 Objectives

The **strategic** objective of this undergraduate thesis is to enhance the performance of the Kyber workflow along with its underlying symmetric cryptographic primitives. This involves modifying the RS5 processor to support hardware accelerators for the key generation, encapsulation, and decapsulation phases of the algorithm. The work focuses on optimizing the Number Theoretic Transform (NTT) operation, the primary computational bottleneck, while also integrating standard RISC-V extensions to accelerate the essential SHA-2 and SHA-3 hashing and bit-manipulation tasks. To achieve this general goal, the **specific** objectives are:

I **Toolchain Modification**.
Modify and rebuild the compiler toolchain to support the custom XKYBER instructions.

II **ZKNH Extension Integration**.
Integrate the ZKNH extension into the RS5 processor to accelerate SHA-2 cryptographic hashing, a feature specifically implemented to fulfill the requirements of the WIMED project.

III **XKYBER Custom Instruction Set**.
Port and integrate the XKYBER custom instruction set to optimize the core operations of the Kyber workflow. The original XKyber extension was developed in [Gewehr et al., 2024].

IV **ZBKB Extension Addition**.
Add the ZBKB extension to accelerate SHA-3 bit-manipulation tasks.

V **Functional Verification and Evaluation**.
Verify the functional correctness of the integrated modules and evaluate the resulting performance improvements.

VI **Post-Synthesis Analysis**.
Perform post-synthesis analysis to assess hardware resource utilization and achieve timing closure.

1.3 Document Organization

This work is organized as follows:

- Chapter 2 defines the foundational terminology, acronyms, and mathematical notation utilized throughout the text, strictly aligned with the NIST FIPS 203 standard.

- Chapter 3 provides the theoretical foundation required to comprehend this work, including the fundamentals of the ML-KEM algorithm, polynomial arithmetic over rings, and the NTT operation, which is the most complex part.
- Chapter 4 details the core contribution of this work: the integration of the custom scalar cryptographic extensions. It covers the structural evolution of the `kybermul` and `kybercompress` instructions, from the initial combinational versions to the final serialized implementations.
- Chapter 5 evaluates the physical hardware implementation, presenting the ASIC synthesis results targeting a 28nm standard-cell library in GENUS and FPGA synthesis in Vivado. This chapter discusses execution latency, area footprints, power dissipation, and the necessary physical trade-offs required to achieve timing closure.
- Chapter 6 concludes this work, summarizing the findings regarding the physical cost of embedding post-quantum security into embedded processors, and points out directions for future research.

2. TERMS, ACRONYMS, AND NOTATION

This chapter establishes the terminology, acronyms, and mathematical notation utilized throughout this work. To ensure alignment with current cryptographic standardization, the definitions and symbols presented herein follow the official National Institute of Standards and Technology (NIST) Federal Information Processing Standard (FIPS) 203 for the Module-Lattice-Based Key-Encapsulation Mechanism (ML-KEM) NIST [2024a].

2.1 Terms and Definitions

Cryptographic module: The set of hardware, software, and/or firmware that implements approved cryptographic functions (including key generation) that are contained within the cryptographic boundary of the module.

Decapsulation: The process of applying the Decaps algorithm of a KEM. This algorithm accepts a KEM ciphertext and the decapsulation key as input and produces a shared secret key as output.

Encapsulation: The process of applying the Encaps algorithm of a KEM. This algorithm accepts the encapsulation key as input, requires private randomness, and produces a shared secret key and an associated ciphertext as output.

(KEM) Ciphertext: A bit string that is produced by encapsulation and used as an input to decapsulation.

Shared secret key: A shared secret that can be used directly as a cryptographic key in symmetric-key cryptography. It does not require additional key derivation.

Security strength: A number associated with the amount of work (i.e., the number of operations) that is required to break a cryptographic algorithm or system.

2.2 Acronyms

The following list encompasses both the cryptographic acronyms defined by NIST and the microarchitectural acronyms relevant to the hardware implementation discussed in this work.

AES Advanced Encryption Standard

ASIC Application-Specific Integrated Circuit

CBD Centered Binomial Distribution

FIPS Federal Information Processing Standard

FPGA Field-Programmable Gate Array

FSM Finite State Machine

GE Gate Equivalent

IS Instruction Set

ISA Instruction Set Architecture

KEM Key-Encapsulation Mechanism

KGE Kilo Gate Equivalent

LUT Look-Up Table

MAC Multiply-Accumulate

ML-KEM Module-Lattice-Based Key-Encapsulation Mechanism Standard

MLWE Module Learning with Errors

NIST National Institute of Standards and Technology

NTT Number-Theoretic Transform

PKE Public-Key Encryption

PQC Post-Quantum Cryptography

PVT Process, Voltage, and Temperature

QoR Quality of Results

SHA Secure Hash Algorithm

SHAKE Secure Hash Algorithm KECCAK

SIMD Single Instruction, Multiple Data

TNS Total Negative Slack

WNS Worst Negative Slack

2.3 Mathematical Symbols

The implementation of ML-KEM relies on polynomial arithmetic over rings. The mathematical notation utilized for the algorithmic description and hardware acceleration is defined below.

Symbol	Description
n	Denotes the integer 256 throughout this document.
q	Denotes the prime integer $3329 = 2^8 \cdot 13 + 1$ throughout this document.
ζ	Denotes the integer 17, which is a primitive n-th root of unity modulo q.
$\mathbb{B}$	The set $\{0, 1, \ldots, 255\}$ of unsigned 8-bit integers (bytes).
$\mathbb{Z}_q$	The ring of integers modulo q (i.e., the set $\{0, 1, \ldots, q-1\}$ equipped with the operations of addition and multiplication modulo q).
R_q	The ring $\mathbb{Z}_q[X]/(X^n + 1)$ consisting of polynomials of the form $f = f_0 + f_1 X + \cdots + f_{255} X^{255}$, where $f_j \in \mathbb{Z}_q$ for all j.
T_q	The image of R_q under the number-theoretic transform. Its elements are called "NTT representations" of polynomials in R_q.
$\hat{f}$	The element of T_q that is equal to the NTT representation of a polynomial $f \in R_q$.
f_j	The coefficient of X^j of a polynomial $f \in R_q$.
$\mathcal{D}_\eta(R_q)$	A certain distribution of polynomials in R_q with small coefficients, from which noise is sampled.
$\times_{T_q}$	Denotes the operation on coefficient arrays that implements product in the ring T_q.
$\lceil x \rceil$	The ceiling of x (i.e., the smallest integer greater than or equal to x).
$\lfloor x \rfloor$	The floor of x (i.e., the largest integer less than or equal to x).
$\lfloor x \rceil$	The rounding of x to the nearest integer.
$A \parallel B$	The concatenation of two arrays or bit strings A and B.
$\text{BitRev}_7(r)$	Bit reversal of a seven-bit integer r.

3. BACKGROUND ON CRYPTOGRAPHIC ALGORITHMS

This Chapter presents the mathematical concepts and algorithms used in the construction of SHA-2, SHA-3, and KYBER extensions.

3.1 Hash Functions

Hash functions compute a digest of a message, generating a short, fixed-length bit string. For a given message, this hash value serves as a unique fingerprint. Unlike many other cryptographic algorithms, hash functions do not rely on a secret key. They are components of digital signature schemes, message authentication codes, and Post-Quantum Cryptography (PQC) algorithms. Furthermore, they are applied in other domains, such as cryptocurrency architectures, pseudo-random number generation, secure password storage, and data integrity verification.

To be effective, cryptographic hash functions must exhibit specific characteristics:

- **Arbitrary message size:** The function $h(x)$ can be applied to a message x of any arbitrary size.
- **Fixed output length:** The function $h(x)$ produces a hash value z of a fixed length, regardless of the input size.
- **Efficiency:** The hash value $h(x)$ must be computationally efficient to generate for any given input x.

Although hash functions do not use cryptographic keys, their security properties guarantee the robustness of the systems that employ them. Vulnerabilities in hash functions can be exploited to compromise security architectures. Therefore, a cryptographically secure hash function must fulfill three security requirements:

- **Preimage Resistance (One-wayness):** For a given output z, it must be computationally infeasible to find any input x such that $h(x) = z$. This ensures the function cannot be easily reversed.
- **Second Preimage Resistance (Weak Collision Resistance):** Given an input x_1 and its corresponding hash $h(x_1)$, it must be computationally infeasible to find a different input x_2 ($x_1 \neq x_2$) such that $h(x_1) = h(x_2)$.
- **Collision Resistance (Strong Collision Resistance):** It must be computationally infeasible to find any pair of distinct inputs x_1 and x_2 ($x_1 \neq x_2$) that produce the same hash value, meaning $h(x_1) = h(x_2)$.

3.2 Secure Hash Algorithm 2 (SHA-2)

The Secure Hash Algorithm 2 (SHA-2) is one of the most widely adopted cryptographic hash functions in security protocols. Designed by the National Security Agency (NSA), it was introduced as the successor to SHA-1, building upon the principles of the MD4 family of message digest algorithms [Rivest, 1992]. The SHA-2 standard encompasses a set of variants distinguished by their output digest sizes, primarily: SHA-224, SHA-256, SHA-384, and SHA-512.

Because the core operations are highly similar across these variants, the following explanation of the internal workflow will focus on the SHA-256 architecture as a representative model, as depicted in Figure 3.1.

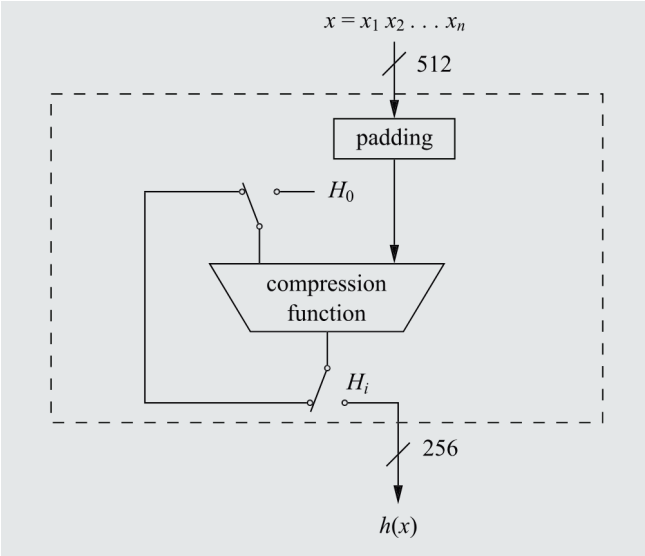


Figure 3.1: High-level diagram of SHA-256 [Paar et al., 2024].

3.2.1 SHA-256 Padding

The padding phase in the SHA-2 algorithm serves a dual purpose: it guarantees structural compliance with the Merkle-Damgård (MD) construction (method of building collision-resistant cryptographic hash functions [Merkle, 1989; Damgård, 1989])and mitigates cryptographic vulnerabilities. By appending a block that securely encodes the original message length before hashing, the algorithm effectively mitigates length-extension, collision, and pseudo-collision attacks. This strategy prevents adversaries from exploiting length differentials to discover secondary messages, thereby thwarting long-message vulnerabilities, fixed-point exploits, and trivial collisions associated with random Initialization Vectors (IVs).

Furthermore, to maintain the security of the MD paradigm, SHA-2 must implement an MD-compliant padding scheme that adheres to the following three mathematical properties:

- The original message M must always be preserved as the exact prefix of the padded output Pad(M).
- Any two original messages of equal length, denoted as $|M_1| = |M_2|$, must result in padded representations of identical sizes, meaning $|\text{Pad}(M_1)| = |\text{Pad}(M_2)|$.
- For any two messages with distinct lengths ($|M_1| \neq |M_2|$), the final block of their respective padded outputs must be unequivocally different, ensuring unique length encoding.

From an architectural viewpoint, the internal compression function of SHA-256 operates on 512-bit data blocks. Consequently, the padding mechanism dynamically shapes the input to ensure the total length is always a multiple of 512 bits. A consequence of this boundary requirement is that even if the original message length is already aligned to a multiple of 512 bits, the algorithm is forced to append an entire additional 512-bit block to encapsulate the padding sequence and the length descriptor.

As Figure 3.2 shows, given an initial message x with l bits, the standard padding procedure is executed as follows: a single 1 bit is appended immediately after the message data, followed by exactly k zero bits, and finally concluded with a 64-bit block containing the binary representation of the original length l. The required number of zero bits, k, is calculated as the smallest non-negative integer that satisfies the eq. (3.1).

$$k \equiv (448 - (l + 1)) \pmod{512} \tag{3.1}$$

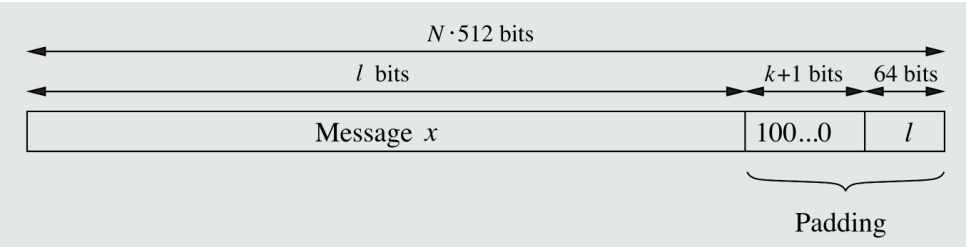


Figure 3.2: Padding of a message in SHA-256 [Paar et al., 2024].

3.2.2 SHA-256 Compression Function

The compression function is the computational core of the SHA-256 algorithm. For each 512-bit block of the padded message, this function processes three primary inputs: the

intermediate hash state from the previous iteration (denoted as $H^{(i-1)}$), the current message schedule (W_j), and an array of sixty-four 32-bit constants (K_j).

These constants ($K_0 \dots K_{63}$) are not arbitrary; they represent the first thirty-two bits of the fractional parts of the cube roots of the first sixty-four prime numbers. A hexadecimal representation of these constants is illustrated in Figure 3.3.

```
428A2F98 71374491 B5C0FBCF E9B5DBA5 3956C25B 59F111F1 923F82A4 AB1C5ED5
D807AA98 12835B01 243185BE 550C7DC3 72BE5D74 80DEB1FE 9BDC06A7 C19BF174
E49B69C1 EFBE4786 0FC19DC6 240CA1CC 2DE92C6F 4A7484AA 5CB0A9DC 76F988DA
983E5152 A831C66D B00327C8 BF597FC7 C6E00BF3 D5A79147 06CA6351 14292967
27B70A85 2E1B2138 4D2C6DFC 53380D13 650A7354 766A0ABB 81C2C92E 92722C85
A2BFE8A1 A81A664B C24B8B70 C76C51A3 D192E819 D6990624 F40E3585 106AA070
19A4C116 1E376C08 2748774C 34B0BCB5 391C0CB3 4ED8AA4A 5B9CCA4F 682E6FF3
748F82EE 78A5636F 84C87814 8CC70208 90BEFFFA A4506CEB BEF9A3F7 C67178F2
```

Figure 3.3: The sixty-four 32-bit constants (K_j) used in SHA-256 [Paar et al., 2024].

To manipulate the data at the bit level, the SHA-256 architecture relies on specific bitwise operations, primarily circular right shifts (rotations) and logical right shifts, defined as follows:

- $\text{ROTR}^n(x)$: A circular bit-shift (rotation) of the 32-bit word x by n positions to the right, as illustrated in Figure 3.4.
- $\text{SHR}^n(x)$: A logical right shift of the 32-bit word x by n positions, equivalent to $x \gg n$.

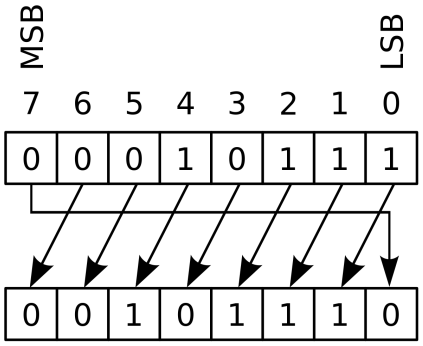


Figure 3.4: Visual representation of a circular bit shift. [TopTechBoy, 2026].

Before entering the main compression loop, the 512-bit message block is expanded into a message schedule comprising sixty-four 32-bit words, denoted as $W_0 \dots W_{63}$. The definition of the message schedule is formalized in eq. (3.2):

$$W_j = \begin{cases} x_i^{(j)} & 0 \leq j \leq 15 \\ \sigma_1(W_{j-2}) \boxplus W_{j-7} \boxplus \sigma_0(W_{j-15}) \boxplus W_{j-16} & 16 \leq j \leq 63 \end{cases} \tag{3.2}$$

Where the auxiliary functions σ_0 and σ_1 are defined in Equation (3.3):

$$\begin{aligned}\sigma_0(x) &= \text{ROTR}^7(x) \oplus \text{ROTR}^{18}(x) \oplus \text{SHR}^3(x)\\ \sigma_1(x) &= \text{ROTR}^{17}(x) \oplus \text{ROTR}^{19}(x) \oplus \text{SHR}^{10}(x)\end{aligned} \tag{3.3}$$

During the actual compression phase, the algorithm utilizes eight 32-bit working variables (labeled A through H). Over the course of 64 discrete rounds, the state of these variables is updated through a non-linear network, as depicted in Figure 3.5.

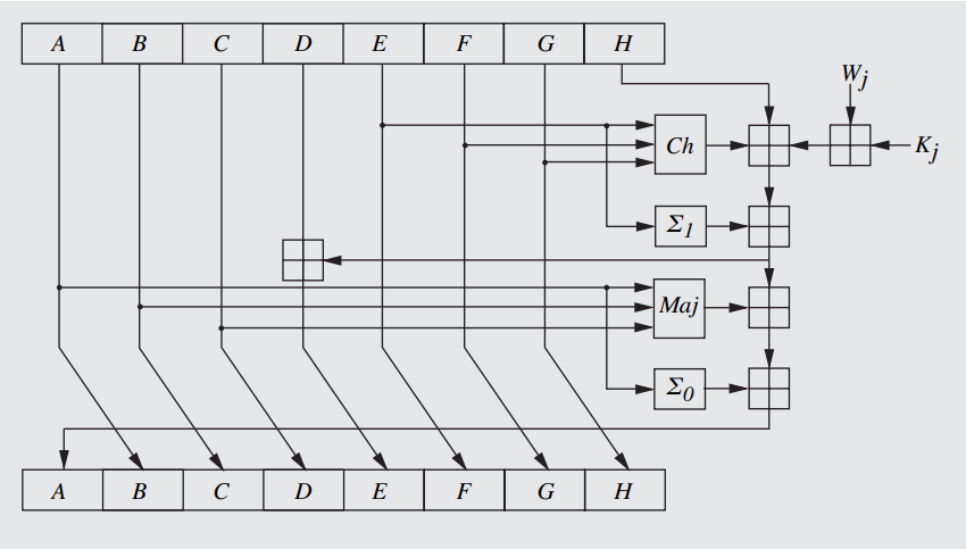


Figure 3.5: Iteration j in the SHA-256 compression function [Paar et al., 2024].

The $\boxplus$ operator denotes modulo 2^{32} addition for 32-bit words. During each iteration j (where $0 \leq j \leq 63$), the eight state variables are updated according to the system shown in Equation (3.4):

$$\begin{aligned} T_1 &= H + \Sigma_1(E) + \text{Ch}(E,F,G) + K_j + W_j & T_2 &= \Sigma_0(A) + \text{Maj}(A,B,C)\\ H &= G & G &= F\\ F &= E & E &= D + T_1\\ D &= C & C &= B\\ B &= A & A &= T_1 + T_2 \end{aligned} \tag{3.4}$$

Where the primary logical functions and rotational operations are defined in eq. (3.5):

$$\begin{aligned}\text{Ch}(x,y,z) &= (x \wedge y) \oplus (\neg x \wedge z)\\ \text{Maj}(x,y,z) &= (x \wedge y) \oplus (x \wedge z) \oplus (y \wedge z)\\ \Sigma_0(x) &= \text{ROTR}^2(x) \oplus \text{ROTR}^{13}(x) \oplus \text{ROTR}^{22}(x)\\ \Sigma_1(x) &= \text{ROTR}^6(x) \oplus \text{ROTR}^{11}(x) \oplus \text{ROTR}^{25}(x)\end{aligned} \tag{3.5}$$

After all 64 iterations are complete, the resulting working variables (A through H) are added modulo 2^{32} to the initial hash values of that block to form the new intermediate state $H^{(i)}$. The final 256-bit message digest is produced by concatenating the final eight 32-bit state variables, as shown in eq. (3.6):

$$\text{SHA-256}(x) = H_n^0 \parallel H_n^1 \parallel H_n^2 \parallel H_n^3 \parallel H_n^4 \parallel H_n^5 \parallel H_n^6 \parallel H_n^7 \tag{3.6}$$

3.3 Secure Hash Algorithm 3 (SHA-3)

The SHA-2 family shares structural similarities with its predecessor, SHA-1, as both rely on the Merkle-Damgård construction. When theoretical vulnerabilities were discovered in SHA-1, the National Institute of Standards and Technology (NIST) initiated a public competition to develop a new cryptographic hash standard. This measure was intended to provide an alternative if SHA-2 were compromised. Ultimately, NIST selected the Keccak algorithm as the foundation for the new SHA-3 standard due to its secure architecture.

3.3.1 Keccak

To understand how SHA-3 operates, it is necessary to present the Keccak implementation. Unlike SHA-1 and SHA-2, SHA-3 departs from the Merkle-Damgård architecture and is instead built upon the sponge construction [Borowski, 2013]. The Keccak process begins with a preprocessing step that segments the message into blocks and applies the required padding. This is followed by two primary operational phases: absorbing and squeezing, as illustrated in Figure 3.6.

The two main stages in the Keccak implementation can be classified as follows:

1. The **Absorbing (or input) Phase**: In this stage, the algorithm sequentially processes the message blocks x_i, integrating them into its internal state.
2. The **Squeezing (or output) Phase**: In this stage, the algorithm produces an output of configurable length by extracting data from its internal state.

3.3.2 SHA-3 Padding

Before the message m is processed, both a domain separation suffix and a padding sequence are appended to it. It is important to note that the suffix is not part of the core

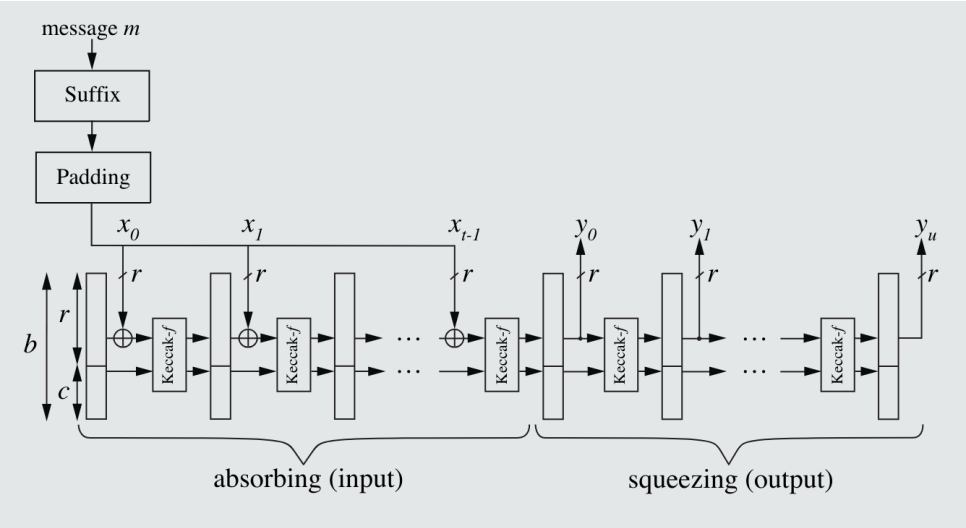


Figure 3.6: The Keccak sponge construction on which SHA-3 and SHAKE128/256 are based. [Paar et al., 2024].

Keccak function but is a specific requirement mandated by the SHA-3 and SHAKE128/256 standards. The suffix values vary depending on the specific algorithm variant, as defined in Table 3.1. In contrast, the multi-rate padding is defined as an integral part of the Keccak function.

Table 3.1: Suffix rules for SHA-3 and SHAKE128/256. [Paar et al., 2024].

	suffix
SHA3	suf = 01
SHAKE128/256	suf = 1111

The padding is appended at the end of the input sequence. Therefore, the final constructed string consists of the original message, the suffix (distinguishing between SHA-3 and SHAKE), and the padding sequence. Keccak uses the ‘pad10*1‘ rule, which appends a single ‘1‘ bit, followed by a necessary number of ‘0‘ bits, and terminated by a final ‘1‘ bit. This ensures that the total length of the input aligns with the block size required for the subsequent absorbing phase, preventing boundary ambiguities as shown in Figure 3.7.

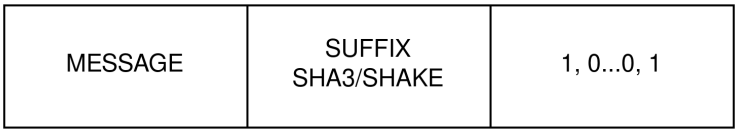


Figure 3.7: Structure of the appended suffix and padding in SHA-3. Source: Author.

3.3.3 The Function Keccak-f

Once the input message m undergoes domain separation and padding, the core processing shifts to the Keccak-f permutation. This function operates on a fixed 1600-bit state, producing an output of the same size. Internally, these 1600 bits are partitioned into two variables: the bit rate (r) and the capacity (c). The exact allocation between r and c determines the algorithm's target security level. Table 3.2 outlines this distribution for the standardized hash and XOF (Extendable-output function) variants.

Table 3.2: The parameters of SHA-3 and SHAKE128/256. [Paar et al., 2024].

	function type	b (state) [bits]	r (rate) [bits]	c (capacity) [bits]	security level [bits]	hash output [bits]
SHA3-224	hash	1600	1152	448	112	224
SHA3-256	hash	1600	1088	512	128	256
SHA3-384	hash	1600	832	768	192	384
SHA3-512	hash	1600	576	1024	256	512
SHAKE128	XOF	1600	1344	256	128	arbitrary
SHAKE256	XOF	1600	1088	512	256	arbitrary

A single round of the Keccak-f permutation is divided into five distinct internal transformations: θ (Theta), ρ (Rho), π (Pi), χ (Chi), and ι (Iota). Figure 3.8 illustrates this sequence.

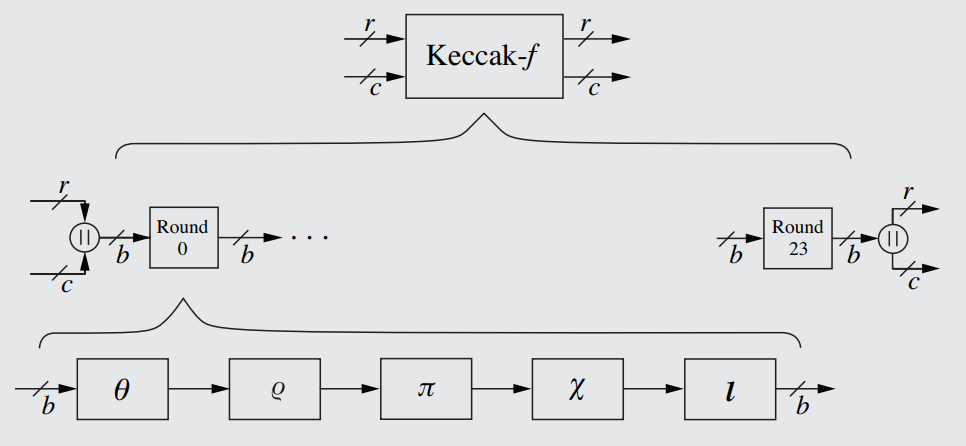


Figure 3.8: Internal structure of the function Keccak-f [Paar et al., 2024].

To achieve high execution performance—whether using matrix operations in software or instantiating 25 distinct registers in hardware—the 1600-bit state is mapped onto a three-dimensional grid. As depicted in Figure 3.9, the data is arranged in a 5×5 spatial matrix. For SHA-3 and SHAKE128/256, the depth of this array along the z-axis is

$w = 64$ bits. Consequently, each (x, y) coordinate forms a "lane" of 64 bits, yielding the total $5 \times 5 \times 64 = 1600$ bits.

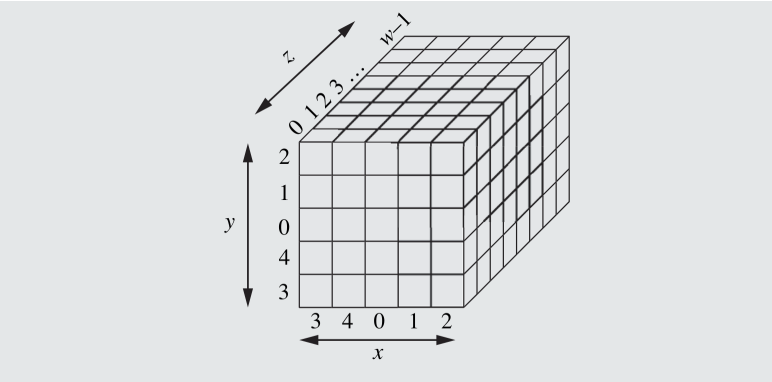


Figure 3.9: The state of Keccak where each small cube represents one bit [Paar et al., 2024].

3.3.3.1 Step θ (Theta)

The Theta (θ) step is the first internal transformation of the round. It relies on XOR operations across w-bit words, paired with a 1-bit rotation. From a hardware perspective, rotating a full 64-bit word by one position is physically equivalent to fetching a bit from the adjacent column at depth $z - 1$. Figure 3.10 maps out this spatial relationship, while Equation (3.7) formalizes the underlying mathematical logic.

$$\begin{aligned} C[x] &= A[x,0] \oplus A[x,1] \oplus A[x,2] \oplus A[x,3] \oplus A[x,4] && ,x = 0,1,2,3,4 \\ D[x] &= C[x-1] \oplus \text{rot}(C[x+1],1) && ,x = 0,1,2,3,4 \\ A'[x,y] &= A[x,y] \oplus D[x] && ,x,y = 0,1,2,3,4 \end{aligned} \tag{3.7}$$

In the diagram, the two highlighted columns compute the parity affecting the target bit. The left column, $C[x-1]$, compresses five w-bit words via XOR. Meanwhile, the right column is shifted to $z-1$, translating the mathematical rotation directly into the 3D grid. The central dark block then absorbs both parities, resulting in the newly updated $A'[x,y]$ state.

3.3.3.2 Step ρ (Rho)

The ρ (Rho) step introduces bitwise rotations along the z-axis for each individual lane $A[x,y]$. The rotation offset is not uniform; it is determined directly by the specific (x,y) coordinates of the lane. The calculation of these offsets is optimized for software implementations, executing the logic detailed in Figure 3.11.

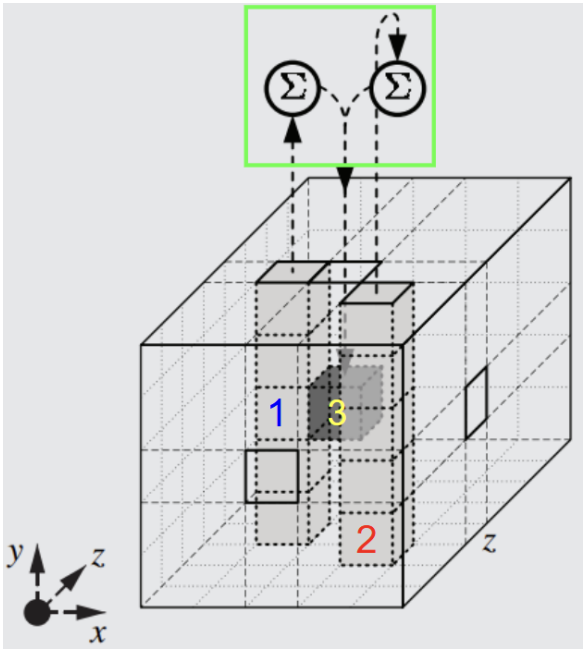


Figure 3.10: The θ step mapping in the Keccak-f permutation. [Paar et al., 2024].

Offset Computation Algorithm for the ρ Step

Output: offset table $T[x, y]$, $x, y = 0, 1, 2, 3, 4$

Initialization: $(x, y) = (1, 0)$, $T[0, 0] = 0$

Algorithm:

1 FOR $t = 0$ TO 23

1.1 $T[x, y] = (t + 1)(t + 2)/2$

1.2 $(x, y) = (y, 2x + 3y \bmod 5)$

2 RETURN ($T[\,]$)

Figure 3.11: Algorithm to compute the rotation offsets for the ρ step. [Paar et al., 2024].

Executing this algorithm generates a fixed set of constants for the entire 5×5 matrix. Table 3.3 maps the final resulting offset value that must be applied to rotate each corresponding $A[x, y]$ lane.

Table 3.3: The rotation offsets of the ρ Step. [Paar et al., 2024].

	$x = 3$	$x = 4$	$x = 0$	$x = 1$	$x = 2$
$y = 2$	153	231	3	10	171
$y = 1$	55	276	36	300	6
$y = 0$	28	91	0	1	190
$y = 4$	120	78	210	66	253
$y = 3$	21	136	105	45	15

3.3.3.3 Step π (Pi)

The π (Pi) step is responsible for the spatial dispersion of the data. Instead of altering the internal bits of the lanes, this function purely rearranges the positions of the 64-bit lanes across the 5×5 state matrix. The permutation mapping is defined by Equation 3.8.

$$A'[x, y] = A[x + 3y, x] \quad , \quad x, y \in \{0, 1, 2, 3, 4\} \tag{3.8}$$

To ensure the coordinates strictly remain within the valid boundaries of the grid, all calculations involving the spatial indices x and y are evaluated using modulo 5 arithmetic.

3.3.3.4 Step χ (Chi)

The χ (Chi) step provides the only nonlinear transformation within the Keccak-f round. It modifies the state array by operating on w-bit lanes along the x-axis. The updated state A' is computed using Equation 3.9.

$$A'[x, y] = A[x, y] \oplus ((\bar{A}[x + 1, y]) \wedge A[x + 2, y]) \quad , \quad x, y \in \{0, 1, 2, 3, 4\} \tag{3.9}$$

Here, the overline notation $\bar{A}[x + 1, y]$ represents a bitwise NOT applied to the immediate neighbor lane, and the $\wedge$ symbol denotes a bitwise AND operation with the lane located two positions away. Because this transformation slides horizontally across the row, the spatial indices wrap around the grid edges using modulo 5 arithmetic, as illustrated by the logic gate mapping in Figure 3.12.

3.3.3.5 Step ι (Iota)

Concluding the round sequence, the ι (Iota) step prevents structural symmetries from propagating across iterations. This operation is performed by XORing the state's origin lane at $[0, 0]$ with a predefined w-bit round constant, denoted as $RC[i]$, as defined in Equation 3.10.

$$A'[0, 0] = A[0, 0] \oplus RC[i] \tag{3.10}$$

An 8-stage Linear Feedback Shift Register (LFSR) derives these pseudo-random values. The exact bit patterns vary according to the permutation's state size (b) and round count (n_r), following the logic established in the NIST FIPS 202 standard.

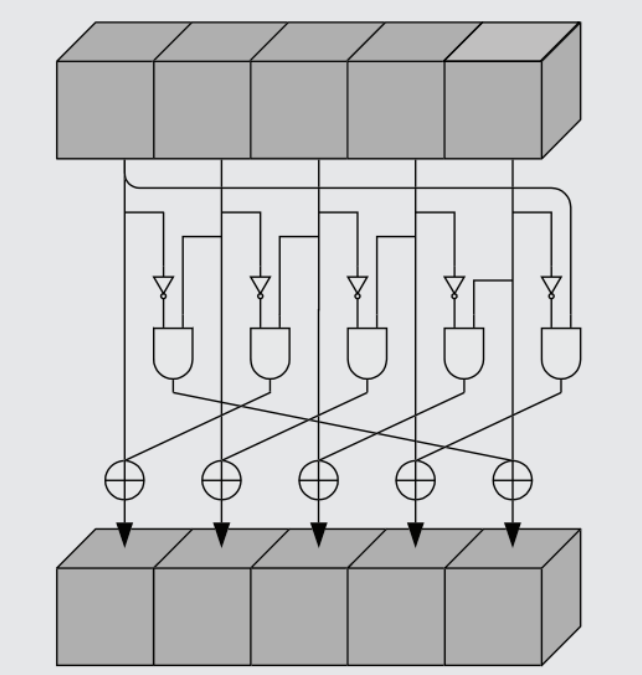


Figure 3.12: Mapping of the χ step in the Keccak-*f* permutation [Paar et al., 2024].

Table 3.4: The round constants for SHA-3 and SHAKE128/256; each constant is a 64-bit word given in hexadecimal notation. [Paar et al., 2024]

$RC[0]$	= 0x0000000000000001	$RC[12]$	= 0x000000008000808B
$RC[1]$	= 0x0000000000008082	$RC[13]$	= 0x800000000000008B
$RC[2]$	= 0x800000000000808A	$RC[14]$	= 0x8000000000008089
$RC[3]$	= 0x8000000080008000	$RC[15]$	= 0x8000000000008003
$RC[4]$	= 0x000000000000808B	$RC[16]$	= 0x8000000000008002
$RC[5]$	= 0x0000000080000001	$RC[17]$	= 0x8000000000000080
$RC[6]$	= 0x8000000080008081	$RC[18]$	= 0x000000000000800A
$RC[7]$	= 0x8000000000008009	$RC[19]$	= 0x800000008000000A
$RC[8]$	= 0x000000000000008A	$RC[20]$	= 0x8000000080008081
$RC[9]$	= 0x0000000000000088	$RC[21]$	= 0x8000000000008080
$RC[10]$	= 0x0000000080008009	$RC[22]$	= 0x0000000080000001
$RC[11]$	= 0x000000008000000A	$RC[23]$	= 0x8000000080008008

The target parameters of the algorithm set the required number of permutations. Since the SHA-3 and SHAKE128/256 instances operate with $n_r = 24$, the implementation cycles through all 24 constants mapped in Table 3.4 over the course of execution.

3.4 Nomenclature: ML-KEM and CRYSTALS-Kyber

Before detailing the operational phases, a brief clarification regarding nomenclature is necessary. The official NIST standard, formally designated as Module-Lattice-Based Key-Encapsulation Mechanism Standard(ML-KEM) in FIPS 203, is a direct standardization of the

CRYSTALS-Kyber algorithm, which was the primary key-encapsulation mechanism selected during the NIST Post-Quantum Cryptography (PQC) standardization process. Because the underlying mathematics and structural logic are fundamentally the same, the terms *ML-KEM*, *CRYSTALS-Kyber*, and simply *Kyber* are frequently used interchangeably throughout literature, hardware implementations, and within the custom architectural extensions proposed in this document.

3.5 Module-Lattice-Based Key-Encapsulation Mechanism Standard

Operating over module lattices, the cryptographic architecture of Module-Lattice-Based Key-Encapsulation Mechanism Standard [NIST, 2024a] derives its security from the computational intractability of the Module Learning with Errors (MLWE) problem. The foundational layer implements a Public-Key Encryption (PKE) primitive that supports fixed 32-byte payloads and provides baseline IND-CPA (Indistinguishability under Chosen-Plaintext Attack) security [Goldwasser and Micali, 1984]. To elevate this underlying scheme into a fully IND-CCA2-secure (Indistinguishability under Adaptive Chosen-Ciphertext Attack) [Rackoff and Simon, 1992; Bellare et al., 1998] Module-Lattice-Based Key Encapsulation Mechanism (ML-KEM), the protocol applies a modified Fujisaki-Okamoto (FO) transform [Lippert, 2014].

While the base public-key encryption (PKE) primitive of ML-KEM provides IND-CPA security, this baseline is insufficient for modern standardization criteria. To meet NIST requirements against active adversaries, the protocol must guarantee IND-CCA2 security, which is exclusively achieved through the ML-KEM formulation. Beyond security models, physical deployment is dictated by computational overhead. Asymmetric lattice-based algorithms inherently exhibit significantly higher latency and resource consumption compared to symmetric block cipher, [Paar et al., 2024]. Because public-key operations execute orders of magnitude slower than symmetric standards like AES, the ML-KEM is strictly reserved for establishing a shared secret, offloading the bulk data encryption to the high-speed symmetric core.

The objective of the ML-KEM standard is to securely establish a shared symmetric key between two communicating parties, conventionally referred to as Alice and Bob. As illustrated in Figure 3.13, the protocol operates sequentially across the three core algorithms.

The process initiates with Alice executing the `KeyGen` algorithm to produce a public encapsulation key and a strictly private decapsulation key. Upon receiving Alice's public encapsulation key, Bob executes the `Encaps` algorithm. This operation deterministically produces Bob's copy of the shared secret key, denoted as K, alongside a corresponding ciphertext that cryptographically encapsulates this secret.

Bob then transmits the ciphertext across the potentially insecure channel back to Alice. To conclude the key establishment phase, Alice processes this ciphertext using her

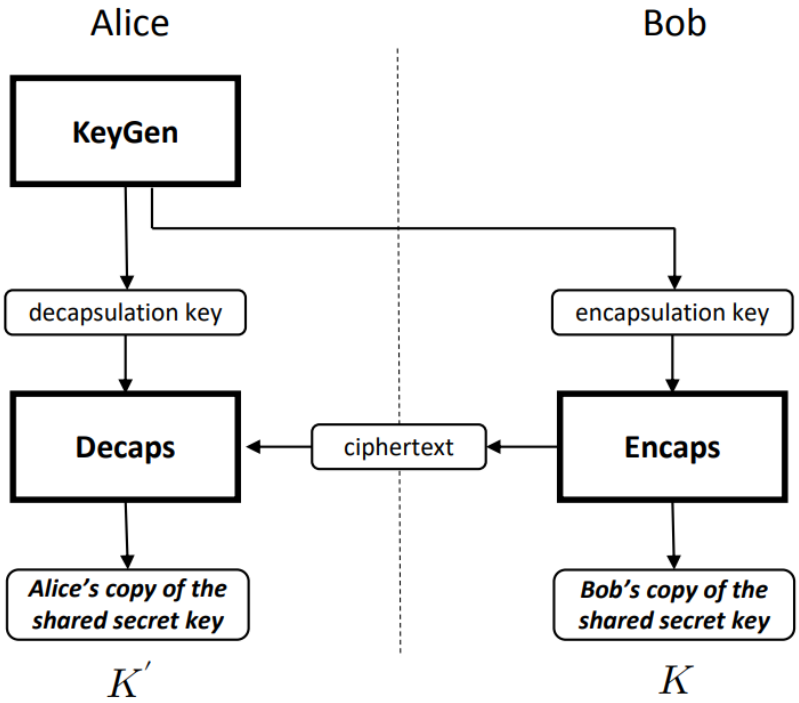


Figure 3.13: High-level view of key establishment using a Key Encapsulation Mechanism (KEM). [NIST, 2024a].

private decapsulation key via the `Decaps` algorithm. This final step yields her copy of the shared secret key, denoted as K'.

The goal of this sequence is to ensure that the outputs satisfy the condition $K' = K$, thereby guaranteeing that both parties share an identical, secure, and uniformly random symmetric key for subsequent bulk data encryption. As specified by the standard, this guarantee holds exclusively when all formal security conditions and input validations are rigorously satisfied.

The following subsections detail ML-KEM protocol, directly mirroring the sequence illustrated in Figure 3.13:

- Section 3.5.1 details the `KeyGen` algorithm, executed by Alice to generate the foundational key pair: the public encapsulation key and the private decapsulation key.
- Section 3.5.2 breaks down the `Encaps` algorithm. Executed by Bob, this function utilizes the internal `Encrypt` primitive to deterministically generate his copy of the shared secret key (K) alongside the ciphertext that encapsulates it.
- Section 3.5.3 examines the `Decaps` algorithm. Executed by Alice upon receiving the message, this function wraps the `Decrypt` primitive to process the ciphertext and output her identical copy of the shared secret key (K').
- Section 3.5.4 describes the underlying `Encrypt` primitive, which utilizes the public encapsulation key to securely encrypt a baseline plaintext.

- Section 3.5.5 explains the corresponding `Decrypt` primitive, which requires the private decapsulation key to recover the original plaintext from a given ciphertext.

3.5.1 Key Generation

The key generation algorithm, specified as `ML-KEM.KeyGen`, accepts no inputs and relies strictly on internal randomness generation. Its primary function is to produce a public encapsulation key (*ek*) and a private decapsulation key (*dk*).

As detailed in Algorithm 3.1 from NIST [2024a], the procedure begins by sampling two distinct 32-byte random seeds, *d* and *z*. The algorithm incorporates a mandatory verification step: if the random bit generation fails (yielding a NULL value for either seed), the process halts and returns an error indicator ($\perp$). If the randomness is successfully generated, these seeds are passed to the internal key generation function, `ML-KEM.KeyGen_internal`, which computes and returns the final encapsulation and decapsulation key pair.

Algorithm 3.1: `ML-KEM.KeyGen()`

Output: Encapsulation key $ek \in \mathcal{B}^{384k+32}$
Output: Decapsulation key $dk \in \mathcal{B}^{768k+96}$

1: $d \leftarrow \mathcal{B}^{32}$ ▷ d is 32 random bytes
2: $z \leftarrow \mathcal{B}^{32}$ ▷ z is 32 random bytes
3: **if** d == NULL **or** z == NULL **then**
4: **return** $\perp$ ▷ Return an error indication if random bit generation failed
5: **end if**
6: $(ek, dk) \leftarrow$ `ML-KEM.KeyGen_internal`(d, z)
7: **return** (ek, dk)

Furthermore, the FIPS 203 standard specifies that if a key pair is received from an external trusted third party rather than generated locally, the owner may optionally perform a series of key pair checks. These include verifying seed consistency by re-running the internal generation, executing structural checks on both the encapsulation and decapsulation keys, and performing a pair-wise consistency test. While these checks can detect certain corruptions, they do not provide an absolute mathematical guarantee that the key pair was properly produced by `ML-KEM.KeyGen`.

Returning to the core execution flow, the internal key generation procedure, defined as `ML-KEM.KeyGen_internal` and formalized in Algorithm 3.2, functions as the primary subroutine for producing the key pair. Unlike its external counterpart, this algorithm does not generate internal entropy; instead, it accepts the two pre-generated 32-byte random seeds, *d* and *z*, directly as inputs.

The execution delegates the cryptographic derivation to the underlying public-key encryption scheme. It invokes the `K-PKE.KeyGen` algorithm using the seed *d* to generate a

base PKE encryption key (ek_{PKE}) and a base PKE decryption key (dk_{PKE}). The final KEM encapsulation key ek is formed by a direct assignment of the PKE encryption key. The final KEM decapsulation key dk, however, is a composite structure. It is constructed by concatenating four specific elements: the PKE decryption key, the encapsulation key itself, the cryptographic hash of the encapsulation key ($H(ek)$), and the second random seed z. As mandated by the standard, this embedded seed z is explicitly reserved to drive the implicit rejection value during the internal decapsulation phase.

Algorithm 3.2: `ML-KEM.KeyGen_internal(d, z)`

Input: Randomness $d \in \mathcal{B}^{32}$
Input: Randomness $z \in \mathcal{B}^{32}$
Output: Encapsulation key $ek \in \mathcal{B}^{384k+32}$
Output: Decapsulation key $dk \in \mathcal{B}^{768k+96}$

1: $(ek_{PKE}, dk_{PKE}) \leftarrow$ `K-PKE.KeyGen`(d) ▷ run key generation for K-PKE
2: $ek \leftarrow ek_{PKE}$ ▷ KEM encaps key is strictly the PKE encryption key
3: $dk \leftarrow (dk_{PKE} \parallel ek \parallel H(ek) \parallel z)$ ▷ KEM decaps key assembly
4: **return** (ek, dk)

To fulfill this underlying cryptographic derivation, the foundational public-key encryption scheme (K-PKE) executes its own key generation algorithm, designated as `K-PKE.KeyGen`. As detailed in Algorithm 3.3, this procedure requires the 32-byte random seed d as input to deterministically compute both the public encryption key (ek_{PKE}) and the private decryption key (dk_{PKE}).

The procedure begins by expanding the input seed d and the module dimension parameter k through the hash function G, producing two independent 32-byte pseudo-random seeds: ρ and σ. The seed ρ is utilized to pseudo-randomly generate the coefficients of the matrix $\hat{\mathbf{A}}$ directly in the Number Theoretic Transform (NTT) domain via the `SampleNTT` function.

Concurrently, the seed σ serves as the cryptographic basis for sampling the secret vector $\mathbf{s}$ and the noise vector $\mathbf{e}$. These vectors are drawn from a centered binomial distribution (`SamplePolyCBD`) using a pseudo-random function (`PRF`) governed by a strictly incrementing counter N. This ensures that every polynomial element is statistically independent.

Once the secret and noise vectors are generated, they are transformed into the NTT domain (yielding $\hat{\mathbf{s}}$ and $\hat{\mathbf{e}}$). The algorithm then evaluates the core Learning With Errors (LWE) equation, as defined in Equation 3.11 ([Paar et al., 2024]).

$$\hat{\mathbf{t}} \leftarrow \hat{\mathbf{A}} \circ \hat{\mathbf{s}} + \hat{\mathbf{e}} \tag{3.11}$$

Algorithm 3.3: `K-PKE.KeyGen(d)`

Input: Randomness $d \in \mathcal{B}^{32}$
Output: Encryption key $ek_{\text{PKE}} \in \mathcal{B}^{384k+32}$
Output: Decryption key $dk_{\text{PKE}} \in \mathcal{B}^{384k}$

$(\rho, \sigma) \leftarrow \text{G}(d \parallel k)$ ▷ expand input to two pseudorandom seeds
$N \leftarrow 0$
for $i = 0$ **to** $k - 1$ **do** ▷ generate matrix $\hat{\mathbf{A}}$
  for $j = 0$ **to** $k - 1$ **do**
    $\hat{\mathbf{A}}[i, j] \leftarrow \texttt{SampleNTT}(\rho \parallel j \parallel i)$
  end for
end for
for $i = 0$ **to** $k - 1$ **do** ▷ generate secret vector $\mathbf{s}$
  $\mathbf{s}[i] \leftarrow \texttt{SamplePolyCBD}_{\eta_1}(\text{PRF}_{\eta_1}(\sigma, N))$
  $N \leftarrow N + 1$
end for
for $i = 0$ **to** $k - 1$ **do** ▷ generate noise vector $\mathbf{e}$
  $\mathbf{e}[i] \leftarrow \texttt{SamplePolyCBD}_{\eta_1}(\text{PRF}_{\eta_1}(\sigma, N))$
  $N \leftarrow N + 1$
end for
$\hat{\mathbf{s}} \leftarrow \texttt{NTT}(\mathbf{s})$ ▷ transform $\mathbf{s}$ to NTT domain
$\hat{\mathbf{e}} \leftarrow \texttt{NTT}(\mathbf{e})$ ▷ transform $\mathbf{e}$ to NTT domain
$\hat{\mathbf{t}} \leftarrow \hat{\mathbf{A}} \circ \hat{\mathbf{s}} + \hat{\mathbf{e}}$ ▷ compute noisy linear system
$ek_{\text{PKE}} \leftarrow \texttt{ByteEncode}_{12}(\hat{\mathbf{t}}) \parallel \rho$
$dk_{\text{PKE}} \leftarrow \texttt{ByteEncode}_{12}(\hat{\mathbf{s}})$
return $(ek_{\text{PKE}}, dk_{\text{PKE}})$

The final encryption key is constructed by byte-encoding the resulting noisy vector $\hat{\mathbf{t}}$ and appending the public seed ρ, which allows the encapsulating party to reconstruct the matrix $\hat{\mathbf{A}}$. The decryption key simply consists of the byte-encoded secret vector $\hat{\mathbf{s}}$.

3.5.2 Encapsulation

The encapsulation procedure, denoted as `ML-KEM.Encaps`, is responsible for generating a shared secret key and its associated ciphertext using a provided encapsulation key. According to the FIPS 203 standard, this algorithm mandates a strict input validation phase for any candidate encapsulation key, represented as $\overline{ek}$, prior to execution.

This validation consists of two specific verification steps:

1. **Type check:** The procedure verifies that the candidate key $\overline{ek}$ is strictly a byte array of length $384k + 32$. If the length does not match, the check fails.

2. **Modulus check:** The procedure ensures that the integers encoded within the public key fall within the valid operational range $[0, q - 1]$. This is executed by decoding and

subsequently re-encoding the first $384k$ bytes of the key, as expressed in Equation 3.12.

$$\text{test} \leftarrow \texttt{ByteEncode}_{12}(\texttt{ByteDecode}_{12}(\overline{ek}[0 : 384k])) \tag{3.12}$$

If the resulting test array is not strictly identical to the original $\overline{ek}[0 : 384k]$, the input checking fails.

Only upon the successful completion of these checks is the candidate key accepted as a valid input, assigned as $ek := \overline{ek}$, and passed to the main algorithm.

The core encapsulation sequence is formalized in Algorithm 3.4 [NIST, 2024a]. It requires the validated encapsulation key ek to generate the outputs. The algorithm starts by sampling 32 bytes of internal randomness, m. Similar to the key generation phase, it verifies the integrity of this random bit generation, returning an error indicator ($\perp$) if the system fails to produce the necessary entropy. The validated key ek and the random message m are then processed by the internal module, `ML-KEM.Encaps_internal`, which computes and returns the final 32-byte shared secret key K and the corresponding ciphertext c.

Algorithm 3.4: `ML-KEM.Encaps(ek)`

Checked input: Encapsulation key $ek \in \mathcal{B}^{384k+32}$
Output: Shared secret key $K \in \mathcal{B}^{32}$
Output: Ciphertext $c \in \mathcal{B}^{32(d_u k + d_v)}$

1: $m \leftarrow \mathcal{B}^{32}$ ▷ m is 32 random bytes
2: **if** m == NULL **then**
3: **return** $\perp$ ▷ Return an error indication if random bit generation failed
4: **end if**
5: $(K, c) \leftarrow$ `ML-KEM.Encaps_internal`(ek, m)
6: **return** (K, c)

Once the entropy is validated, the execution transitions to the internal subroutine, denoted as `ML-KEM.Encaps_internal` and formalized in Algorithm 3.5. This function serves as the core cryptographic engine for generating the shared secret key and its corresponding ciphertext. As defined in the standard, this function requires two specific inputs directly from the upper layer: the previously validated encapsulation key (ek) and the newly generated 32-byte array of randomness (m).

The execution fundamentally relies on the underlying public-key encryption algorithm. Initially, the procedure derives the shared secret key K alongside the necessary encryption randomness r. This is accomplished by applying the hash function H to the encapsulation key ek, concatenating the resulting hash with the random input m, and subsequently processing this combined data block through the hash function G.

Following this derivation, the algorithm invokes the `K-PKE.Encrypt` routine. It encrypts the random value m using the provided encapsulation key ek and the newly derived

randomness r, which directly yields the ciphertext c. The function concludes by outputting both the computed shared secret key K and the associated ciphertext c.

Algorithm 3.5: `ML-KEM.Encaps_internal(ek, m)`

Input: Encapsulation key $ek \in \mathcal{B}^{384k+32}$
Input: Randomness $m \in \mathcal{B}^{32}$
Output: Shared secret key $K \in \mathcal{B}^{32}$
Output: Ciphertext $c \in \mathcal{B}^{32(d_u k+d_v)}$

1: $(K, r) \leftarrow \mathrm{G}(m \,\|\, \mathrm{H}(ek))$
2: $c \leftarrow$ `K-PKE.Encrypt`(ek, m, r)
3: **return** (K, c)

3.5.3 Decapsulation

The decapsulation algorithm, designated as `ML-KEM.Decaps`, is responsible for recovering the shared secret key from a provided ciphertext using the decapsulation key. Operating entirely deterministically, it accepts a candidate decapsulation key ($\overline{dk}$) and a candidate ciphertext ($\overline{c}$).

The FIPS 203 standard strictly mandates a tripartite input checking phase prior to processing these inputs:

1. **Ciphertext type check:** The procedure verifies that the candidate ciphertext $\overline{c}$ is a byte array of exact length $32(d_u k + d_v)$.

2. **Decapsulation key type check:** The procedure verifies that the candidate key $\overline{dk}$ is a byte array of exact length $768k + 96$.

3. **Hash check:** The procedure validates the internal consistency of the decapsulation key by computing the hash of its embedded PKE encryption key component, as formalized in Equation 3.13.

$$\text{test} \leftarrow \mathrm{H}(\overline{dk}[384k : 768k + 32]) \tag{3.13}$$

   If the computed test value does not identically match the stored hash value located at $\overline{dk}[768k + 32 : 768k + 64]$, the input checking fails.

If any of these checks fail, the algorithm must not proceed. The standard specifies that while assurance of the decapsulation key's validity can be established through external means (and therefore the key check may not need to be re-run on every execution), the ciphertext checking must be performed with every single execution of `ML-KEM.Decaps`.

Upon successful validation, the inputs are formally assigned as $dk := \overline{dk}$ and $c := \overline{c}$. These checked inputs are then passed to Algorithm 3.6, which simply delegates the core cryptographic operations to the internal subroutine, `ML-KEM.Decaps_internal`, and returns the resulting shared secret key K'.

Algorithm 3.6: `ML-KEM.Decaps(dk, c)`

Checked input: Decapsulation key $dk \in \mathcal{B}^{768k+96}$
Checked input: Ciphertext $c \in \mathcal{B}^{32(d_u k+d_v)}$
Output: Shared secret key $K \in \mathcal{B}^{32}$

1: $K' \leftarrow$ `ML-KEM.Decaps_internal`(dk, c)
2: **return** K'

Following this delegation, the internal decapsulation procedure, defined as `ML-KEM.Decaps_internal`, accepts the validated decapsulation key (dk) and ciphertext (c) to produce the shared secret key. Unlike the external key generation and encapsulation processes, this algorithm operates entirely deterministically and does not consume any internal randomness.

As detailed in Algorithm 3.7, the operation begins by parsing the decapsulation key into four distinct components: the underlying PKE decryption key (dk_{PKE}), the PKE encryption key (ek_{PKE}), the hash of the encryption key (h), and the pseudo-random implicit rejection value (z).

The procedure decrypts the received ciphertext using dk_{PKE} to recover a candidate plaintext m'. Subsequently, the algorithm initiates a re-encryption sequence to verify the ciphertext's integrity. It computes a candidate shared secret K' and the required encryption randomness r' by hashing the recovered plaintext m' alongside h. The plaintext m' is then re-encrypted via the `K-PKE.Encrypt` function using r' and ek_{PKE}, yielding a verification ciphertext c'.

To finalize the process, the algorithm compares the newly generated c' against the original input c. If the ciphertexts do not match, the algorithm performs an implicit rejection. In this scenario, the output key is replaced by $\bar{K}$, which is derived by hashing the rejection value z together with the ciphertext c using the function J.

Crucially, the FIPS 203 standard dictates that the boolean result of the ciphertext comparison (the "implicit reject" flag) constitutes sensitive intermediate data. This flag must be strictly destroyed prior to the algorithm's termination and is explicitly forbidden from being returned or exposed in any form.

Algorithm 3.7: `ML-KEM.Decaps_internal(dk, c)`

Input: Decapsulation key $dk \in \mathcal{B}^{768k+96}$
Input: Ciphertext $c \in \mathcal{B}^{32(d_u k + d_v)}$
Output: Shared secret key $K \in \mathcal{B}^{32}$

$dk_{\text{PKE}} \leftarrow dk[0 : 384k]$ ▷ extract PKE decryption key
$ek_{\text{PKE}} \leftarrow dk[384k : 768k + 32]$ ▷ extract PKE encryption key
$h \leftarrow dk[768k + 32 : 768k + 64]$ ▷ extract hash of PKE encryption key
$z \leftarrow dk[768k + 64 : 768k + 96]$ ▷ extract implicit rejection value
$m' \leftarrow \texttt{K-PKE.Decrypt}(dk_{\text{PKE}}, c)$ ▷ decrypt ciphertext
$(K', r') \leftarrow \text{G}(m' \,\|\, h)$
$\bar{K} \leftarrow \text{J}(z \,\|\, c)$
$c' \leftarrow \texttt{K-PKE.Encrypt}(ek_{\text{PKE}}, m', r')$ ▷ re-encrypt using the derived randomness r'
if $c \neq c'$ **then**
    $K' \leftarrow \bar{K}$ ▷ if ciphertexts do not match, "implicitly reject"
end if
return K'

3.5.4 K-PKE Encryption

As established in the protocol hierarchy, the foundational public-key encryption primitive is not exposed directly for data transmission, compared with decapsulation and encapsulation. Instead, it serves as the core computational engine embedded within the upper-layer `Encaps` (algorithm 3.5) and `Decaps` (algorithm 3.7) mechanisms, operating exclusively to encapsulate the symmetric shared secret.

Specifically, this underlying algorithm, denoted as `K-PKE.Encrypt`, is responsible for encrypting a 32-byte plaintext message m into a ciphertext c. As detailed in Algorithm 3.8, this procedure requires the public encryption key ek_{PKE}, the message m, and a 32-byte array of randomness r.

The execution begins by parsing the encryption key. The algorithm extracts the byte-encoded vector, decodes it into the polynomial vector $\hat{\mathbf{t}}$ in the NTT domain, and isolates the 32-byte seed ρ. Utilizing this seed, the procedure regenerates the pseudo-random matrix $\hat{\mathbf{A}}$ exactly as it was formulated during the key generation phase.

Subsequently, the algorithm samples a secret vector $\mathbf{y}$ and two noise components, $\mathbf{e}_1$ and e_2. These elements are drawn from a centered binomial distribution, relying on pseudo-randomness expanded from the input r via a PRF and managed by an incrementing counter N.

To construct the ciphertext, $\mathbf{y}$ is transformed into the NTT domain to produce $\hat{\mathbf{y}}$. The procedure then computes a new set of noisy linear equations. The first component, $\mathbf{u}$, is computed by applying the inverse NTT to the point-wise multiplication of the transposed matrix $\hat{\mathbf{A}}^T$ and $\hat{\mathbf{y}}$, and then adding the noise vector $\mathbf{e}_1$. The second component, v, is com-

puted similarly using $\hat{\mathbf{t}}^T$ and adding the noise term e_2. To embed the data, the input message m is decoded, decompressed into a polynomial μ, and added directly to v.

Finally, the resulting polynomials $\mathbf{u}$ and v are compressed, serialized into byte arrays c_1 and c_2, and concatenated to form the final ciphertext c.

Algorithm 3.8: `K-PKE.Encrypt(`ek_{PKE}`, m, r)`

Input: Encryption key $ek_{\text{PKE}} \in \mathcal{B}^{384k+32}$
Input: Message $m \in \mathcal{B}^{32}$
Input: Randomness $r \in \mathcal{B}^{32}$
Output: Ciphertext $c \in \mathcal{B}^{32(d_u k + d_v)}$

$N \leftarrow 0$
$\hat{\mathbf{t}} \leftarrow \texttt{ByteDecode}_{12}(ek_{\text{PKE}}[0 : 384k])$
$\rho \leftarrow ek_{\text{PKE}}[384k : 384k + 32]$
for $i = 0$ **to** $k - 1$ **do** ▷ re-generate matrix $\hat{\mathbf{A}}$
  for $j = 0$ **to** $k - 1$ **do**
    $\hat{\mathbf{A}}[i, j] \leftarrow \texttt{SampleNTT}(\rho \,\|\, j \,\|\, i)$
  end for
end for
for $i = 0$ **to** $k - 1$ **do** ▷ generate $\mathbf{y}$
  $\mathbf{y}[i] \leftarrow \texttt{SamplePolyCBD}_{\eta_1}(\texttt{PRF}_{\eta_1}(r, N))$
  $N \leftarrow N + 1$
end for
for $i = 0$ **to** $k - 1$ **do** ▷ generate $\mathbf{e}_1$
  $\mathbf{e}_1[i] \leftarrow \texttt{SamplePolyCBD}_{\eta_2}(\texttt{PRF}_{\eta_2}(r, N))$
  $N \leftarrow N + 1$
end for
$e_2 \leftarrow \texttt{SamplePolyCBD}_{\eta_2}(\texttt{PRF}_{\eta_2}(r, N))$ ▷ sample e_2
$\hat{\mathbf{y}} \leftarrow \texttt{NTT}(\mathbf{y})$
$\mathbf{u} \leftarrow \texttt{NTT}^{-1}(\hat{\mathbf{A}}^T \circ \hat{\mathbf{y}}) + \mathbf{e}_1$
$\mu \leftarrow \texttt{Decompress}_1(\texttt{ByteDecode}_1(m))$
$v \leftarrow \texttt{NTT}^{-1}(\hat{\mathbf{t}}^T \circ \hat{\mathbf{y}}) + e_2 + \mu$
$c_1 \leftarrow \texttt{ByteEncode}_{d_u}(\texttt{Compress}_{d_u}(\mathbf{u}))$
$c_2 \leftarrow \texttt{ByteEncode}_{d_v}(\texttt{Compress}_{d_v}(v))$
return $c \leftarrow (c_1 \,\|\, c_2)$

3.5.5 K-PKE Decryption

Complementing the encryption primitive, the base decryption algorithm serves a singular role within the protocol hierarchy. While the encryption engine is utilized by both encapsulating and decapsulating parties to facilitate ciphertext verification, the underlying decryption function is executed exclusively within the upper-layer `Decaps` mechanism (algorithm 3.7).

Specifically, this algorithm, denoted as `K-PKE.Decrypt`, recovers the 32-byte plaintext message m from a ciphertext c utilizing the secret decryption key dk_{PKE}. As detailed in Algorithm 3.9, the procedure operates deterministically and requires no additional randomness.

The algorithm begins by partitioning the serialized ciphertext c into its two constituent components: the vector c_1 and the scalar c_2. These components are individually decoded and decompressed into the polynomial vector $\mathbf{u}'$ and the polynomial v', respectively. These elements represent the coefficients and the "noisy" constant term of the underlying linear system.

Subsequently, the algorithm decodes the private key dk_{PKE} to retrieve the secret polynomial vector $\hat{\mathbf{s}}$ in the NTT domain. The core decryption operation calculates the error-free constant term w by computing the difference between the received constant term v' and the inner product of the secret vector with the decoded ciphertext vector $\mathbf{u}'$, as defined in Equation 3.14:

$$w \leftarrow v' - \texttt{NTT}^{-1}(\hat{\mathbf{s}}^T \circ \texttt{NTT}(\mathbf{u}')) \tag{3.14}$$

Finally, the message m is extracted by applying a 1-bit compression to the polynomial w, which effectively filters out the accumulated noise, followed by a final byte-encoding step to reconstruct the original 32-byte plaintext.

Algorithm 3.9: `K-PKE.Decrypt(`dk_{PKE}`, c)`

Input: Decryption key $dk_{\text{PKE}} \in \mathcal{B}^{384k}$
Input: Ciphertext $c \in \mathcal{B}^{32(d_u k + d_v)}$
Output: Message $m \in \mathcal{B}^{32}$

1: $c_1 \leftarrow c[0 : 32d_u k]$
2: $c_2 \leftarrow c[32d_u k : 32(d_u k + d_v)]$
3: $\mathbf{u}' \leftarrow \texttt{Decompress}_{d_u}(\texttt{ByteDecode}_{d_u}(c_1))$ ▷ decompress ciphertext vector
4: $v' \leftarrow \texttt{Decompress}_{d_v}(\texttt{ByteDecode}_{d_v}(c_2))$ ▷ decompress ciphertext scalar
5: $\hat{\mathbf{s}} \leftarrow \texttt{ByteDecode}_{12}(dk_{\text{PKE}})$ ▷ decode secret vector
6: $w \leftarrow v' - \texttt{NTT}^{-1}(\hat{\mathbf{s}}^T \circ \texttt{NTT}(\mathbf{u}'))$ ▷ compute error-free constant
7: $m \leftarrow \texttt{ByteEncode}_1(\texttt{Compress}_1(w))$ ▷ decode plaintext message
8: **return** m

3.5.6 Auxiliary Cryptographic Functions

The ML-KEM protocol algorithms require the use of several underlying cryptographic functions, all instantiated by means of approved hash functions or Extendable-Output Functions (XOFs) from the SHA-3 family, as specified in the FIPS 202 standard. To guarantee full interoperability and mitigate ambiguities between lengths measured in bits

and bytes, the FIPS 203 standard establishes wrappers where all inputs and outputs are processed solely as byte arrays.

3.5.6.1 Pseudorandom Function (PRF)

The PRF is instantiated for deterministic seed expansion and noise generation. It takes a parameter $\eta \in \{2, 3\}$, a 32-byte input s, and a 1-byte counter b, producing an output of $64 \cdot \eta$ bytes. The operation is defined over the SHAKE256 primitive in Equation 3.15:

$$\mathtt{PRF}_\eta(s, b) := \mathtt{SHAKE256}(s \parallel b, 8 \cdot 64 \cdot \eta) \tag{3.15}$$

In this formulation, the parameter η is utilized exclusively to specify the desired output length in bits, and does not perform domain separation.

3.5.6.2 Hash Functions (H, J, G)

The specification defines three independent hash functions for key derivation and cryptographic integrity checks. Functions H and J take variable-length inputs and produce static 32-byte outputs, as defined in Equations 3.16 and 3.17:

$$\mathrm{H}(s) := \mathtt{SHA3\text{-}256}(s) \tag{3.16}$$

$$\mathrm{J}(s) := \mathtt{SHAKE256}(s, 8 \cdot 32) \tag{3.17}$$

Function G also processes a variable-length input but produces a 64-byte output. This output is structurally designed to be partitioned into two independent 32-byte blocks ($a \parallel b$), according to Equation 3.18:

$$\mathrm{G}(c) := \mathtt{SHA3\text{-}512}(c) \tag{3.18}$$

3.5.6.3 Extendable-Output Function (XOF)

To support iterative operations such as public matrix expansion, the standard adopts a XOF wrapper founded on the incremental API for SHAKE128 (documented in SP 800-185). This incremental interface is defined by three standardized routines:

- **XOF.Init():** Initializes the internal SHAKE128 context (`ctx`).
- **XOF.Absorb(ctx, str):** Injects the byte array `str` into the absorbing phase and updates the context state.

- **XOF.Squeeze(ctx, ℓ):** Extracts ℓ bytes of output produced during the squeezing phase. To guarantee type consistency, this routine is internally mapped to $\texttt{SHAKE128.Squeeze}(\texttt{ctx}, 8 \cdot \ell)$, ensuring that the length request is always expressed in bytes to the primitive.

3.5.7 Data Conversion and Compression Primitives

The ML-KEM specification relies on a set of deterministic functions to manage polynomial coefficient sizes and to serialize mathematical structures into transmittable byte arrays.

3.5.7.1 Compression and Decompression

To bound the ciphertext size and intentionally discard low-order bits—which inherently manages the Learning With Errors (LWE) noise—the protocol defines the $\texttt{Compress}_d$ and $\texttt{Decompress}_d$ functions. Given the ML-KEM prime modulus $q = 3329$ (which has a bit length of 12), and for a parameterized bit depth $d < 12$, these functions are defined over the integers modulo q and modulo 2^d, as shown in Equations 3.19 and 3.20:

$$\texttt{Compress}_d(x) := \lceil (2^d/q) \cdot x \rfloor \pmod{2^d} \tag{3.19}$$

$$\texttt{Decompress}_d(y) := \lceil (q/2^d) \cdot y \rfloor \tag{3.20}$$

As strictly mandated by the FIPS 203 standard, the division and rounding operations ($\lceil \cdot \rfloor$) required by these functions must be performed entirely within the set of rational numbers. The use of floating-point arithmetic is explicitly prohibited to prevent platform-dependent rounding discrepancies that could compromise cryptographic consistency.

3.5.7.2 Encoding and Decoding

The conversion between arrays of 256 integers modulo m and standardized byte arrays is governed by the $\texttt{ByteEncode}_d$ and $\texttt{ByteDecode}_d$ algorithms. The target modulus m is directly determined by the bit depth d: $m = 2^d$ for instances where $d < 12$, and $m = q$ when $d = 12$.

Algorithm 3.10 sequentially extracts d bits from each integer in the input array to construct a packed bit stream, which is subsequently converted into a byte array. Algorithm 3.11 performs the exact inverse operation, unpacking the byte array back into an array of d-bit integers.

Algorithm 3.10: $\texttt{ByteEncode}_d(\texttt{F})$

Input: Integer array $F \in \mathbb{Z}_m^{256}$
Output: Byte array $B \in \mathcal{B}^{32d}$

for $i = 0$ **to** 255 **do**
  $a \leftarrow F[i]$ ▷ $a \in \mathbb{Z}_m$
  for $j = 0$ **to** $d - 1$ **do**
    $b[i \cdot d + j] \leftarrow a \bmod 2$
    $a \leftarrow (a - b[i \cdot d + j])/2$ ▷ Note: $a - b[\dots]$ is always even
  end for
end for
$B \leftarrow \texttt{BitsToBytes}(b)$
return B

Algorithm 3.11: $\texttt{ByteDecode}_d(\texttt{B})$

Input: Byte array $B \in \mathcal{B}^{32d}$
Output: Integer array $F \in \mathbb{Z}_m^{256}$

$b \leftarrow \texttt{BytesToBits}(B)$
for $i = 0$ **to** 255 **do**
  $F[i] \leftarrow \sum_{j=0}^{d-1} b[i \cdot d + j] \cdot 2^j \bmod m$
end for
return F

3.5.8 Number Theoretic Transform (NTT)

A prerequisite to understanding the ML-KEM workflow is learning the Number Theoretic Transform (NTT). Across all lattice-based cryptographic schemes, accelerating polynomial multiplication remains the most critical computational challenge. Modern implementations bypass this bottleneck by leveraging the NTT. Operating as an FFT equivalent over finite rings, the NTT utilizes the convolution theorem to compress the algorithmic complexity from $O(n^2)$ to $O(n \log n)$. To establish a comprehensive background, this section maps out both the theoretical framework and the practical execution of the transform.

In cryptographic literature, the terminology surrounding the Number Theoretic Transform (NTT) is frequently overloaded, often blurring the distinction between the mathematical transform itself and its optimized algorithmic implementations. A direct, naive evaluation of the standard NTT dictates a quadratic computational cost of $O(n^2)$. To circumvent this bottleneck, practical deployments rely on FFT-inspired architectures. These “fast-NTT” algorithms reorganize the arithmetic data flow, compressing the operational footprint to a $O(n \log n)$ complexity.

Prior to detailing the NTT protocol, it is necessary to establish the specific parameter sets that configure the ML-KEM algorithm, as detailed in Table 3.5.

Table 3.5: Parameter sets for Module-Lattice-Based Key-Encapsulation Mechanism Standard. [NIST, 2024a]

	n	k	q	η_1	η_2	(d_u, d_v)	δ
ML-KEM-512	256	2	3329	3	2	$(10, 4)$	2^{-139}
ML-KEM-768	256	3	3329	2	2	$(10, 4)$	2^{-164}
ML-KEM-1024	256	4	3329	2	2	$(11, 5)$	2^{-174}

3.5.8.1 Primitive n-th Root of Unity

Before analyzing operations over finite rings, it is useful to observe how the primitive n-th root of unity functions within the standard Discrete Fourier Transform (DFT). The DFT leverages these roots to define its rotational phase factors, structuring them into a dense transformation matrix. Consequently, the entire frequency transform is flattened into a straightforward matrix-vector multiplication, an architecture that naturally favors computational acceleration.

In the context of polynomial operations over a finite ring $\mathbb{Z}_q$, an element $\omega \in \mathbb{Z}_q$ is classified as a *primitive n-th root of unity* if it strictly satisfies two algebraic conditions. First, computing its n-th power must result in the multiplicative identity. Second, no smaller positive integer power k can yield this identity. Formally, ω is governed by the relations expressed in Equations 3.21 and 3.22.

$$\omega^n \equiv 1 \pmod{q} \tag{3.21}$$

$$\omega^k \not\equiv 1 \pmod{q} \quad \text{for all } 1 \leq k < n \tag{3.22}$$

3.5.8.2 NTT-Based Positive-Wrapped Convolution

While the standard Discrete Fourier Transform (DFT) maps real-world signals into a tangible frequency spectrum, the Number Theoretic Transform (NTT) operates entirely as an abstract algebraic construct over finite rings. It lacks any physical equivalent. However, its value in cryptography lies exclusively in a structural parallel: the NTT perfectly inherits the convolution theorem from the DFT framework.

Definition 3.5.1. *The Number Theoretic Transform (NTT) of a vector of polynomial coefficients* $\boldsymbol{a}$ *is defined as* $\hat{\boldsymbol{a}} = NTT(\boldsymbol{a})$*, where:*

$$\hat{\boldsymbol{a}}_j = \sum_{i=0}^{n-1} \omega^{ij} a_i \pmod{q} \tag{3.23}$$

for $j = 0, 1, 2, \ldots, n-1$.

Based on Equation 3.23, the polynomial evaluation can be reformulated as a linear transformation using the sequence of coefficients and the powers of the n-th root of unity. Consequently, the NTT computation can be mapped into a matrix-vector multiplication, as demonstrated in Eq. 3.24.

Example. *Let $G(x) = 1 + 2x + 3x^2 + 4x^3$ or in vector notation $\boldsymbol{g} = [1, 2, 3, 4]$. We can infer that $n = 4$. Suppose we work in the ring $\mathbb{Z}_{7681}$ and ω is its primitive n-th root of unity. The NTT of $\boldsymbol{g}$, $\hat{\boldsymbol{g}}$, can be calculated by the following matrix multiplication:*

$$\hat{\boldsymbol{g}} = \begin{bmatrix} \omega^{0\times0} & \omega^{0\times1} & \omega^{0\times2} & \omega^{0\times3} \\ \omega^{1\times0} & \omega^{1\times1} & \omega^{1\times2} & \omega^{1\times3} \\ \omega^{2\times0} & \omega^{2\times1} & \omega^{2\times2} & \omega^{2\times3} \\ \omega^{3\times0} & \omega^{3\times1} & \omega^{3\times2} & \omega^{3\times3} \end{bmatrix} \begin{bmatrix} 1 \\ 2 \\ 3 \\ 4 \end{bmatrix} \tag{3.24}$$

3.5.8.3 Inverse Number Theoretic Transform Based on ω

Definition 3.5.2. *The Inverse Number Theoretic Transform (INTT) of a transformed vector $\hat{\boldsymbol{a}}$ is defined as $\boldsymbol{a} = INTT(\hat{\boldsymbol{a}})$, where:*

$$\boldsymbol{a}_i = n^{-1} \sum_{j=0}^{n-1} \omega^{-ij} \hat{a}_j \pmod{q} \tag{3.25}$$

for $i = 0, 1, 2, \ldots, n-1$.

Note that the INTT, as defined in Equation 3.25, shares a structural symmetry with the forward NTT. The primary mathematical distinctions are the substitution of the root ω with its multiplicative inverse in $\mathbb{Z}_q$, alongside the introduction of the scaling factor n^{-1}. By definition, this transform is invertible, guaranteeing the property that $\boldsymbol{a} = \text{INTT}(\text{NTT}(\boldsymbol{a}))$.

3.5.8.4 Using NTT to Calculate Positive-Wrapped Convolutions

Operating as the finite-ring analogue of the Discrete Fourier Transform (DFT), the NTT inherently supports the convolution theorem. This parallel is crucial, as it allows the system to bypass standard quadratic multiplication. Instead, the positive-wrapped convolution of two polynomials can be calculated efficiently by translating the operands into the transform domain, executing a point-wise multiplication, and reversing the mapping.

Definition 3.5.3. *Let $\boldsymbol{a}$ and $\boldsymbol{b}$ represent the coefficient vectors of two polynomials defined over the ring $\mathbb{Z}_q$. Their positive-wrapped convolution, yielding the resultant vector $\boldsymbol{c}$, is formally defined by the operation sequence in Equation 3.26:*

$$\boldsymbol{c} = INTT(NTT(\boldsymbol{a}) \circ NTT(\boldsymbol{b})) \tag{3.26}$$

where the $\circ$ symbol denotes the Hadamard (point-wise) product, with all resulting scalar multiplications strictly reduced modulo q.

3.5.8.5 Examples

To validate the convolution theorem over finite rings, the following examples from [Satriawan and Mareta, 2024] compute the positive-wrapped convolution of two polynomials using both the traditional algebraic approach and the NTT framework. The operand vectors are defined as $\boldsymbol{g} = [1, 2, 3, 4]$ and $\boldsymbol{h} = [5, 6, 7, 8]$, corresponding to the polynomials $G(x) = 1 + 2x + 3x^2 + 4x^3$ and $H(x) = 5 + 6x + 7x^2 + 8x^3$. All operations are evaluated in the ring $\mathbb{Z}_{7681}$, with $n = 4$ and a primitive n-th root of unity $\omega = 3383$.

Method 1: Schoolbook Multiplication

The standard algebraic approach requires computing the linear convolution of $G(x)$ and $H(x)$, followed by a modular reduction. The full polynomial multiplication yields Equation 3.27:

$$Y(x) = G(x)H(x) = 32x^6 + 52x^5 + 61x^4 + 60x^3 + 34x^2 + 16x + 5 \tag{3.27}$$

To enforce the positive-wrapped boundary, $Y(x)$ is divided by the modulo polynomial $x^4 - 1$. Applying polynomial long division, the higher-order terms wrap around, reducing the equation to its remainder as shown in Equation 3.28:

$$Y(x) \pmod{x^4 - 1} = 60x^3 + 66x^2 + 68x + 66 \tag{3.28}$$

Extracted as a coefficient array, the final convolution vector is $\boldsymbol{c} = [66, 68, 66, 60]$.

Method 2: NTT Framework

The same circular convolution can be computed in the transform domain by using the Number Theoretic Transform (NTT). In this example, the transform length is $n = 4$ and the arithmetic is performed over the finite field defined by the prime modulus

$$q = 7681. \tag{3.29}$$

The value $q = 7681$ is the modulus used in all modular additions and multiplications. It is suitable for an NTT of size 4 because $q - 1 = 7680$ is divisible by 4. The selected root of unity is

$$\omega = 3383. \tag{3.30}$$

This value is a primitive fourth root of unity modulo 7681, since

$$\omega^0 \equiv 1 \pmod{7681}, \tag{3.31}$$
$$\omega^1 \equiv 3383 \pmod{7681}, \tag{3.32}$$
$$\omega^2 \equiv 7680 \equiv -1 \pmod{7681}, \tag{3.33}$$
$$\omega^3 \equiv 4298 \pmod{7681}, \tag{3.34}$$
$$\omega^4 \equiv 1 \pmod{7681}. \tag{3.35}$$

Therefore, the NTT of a vector

$$\boldsymbol{x} = [x_0, x_1, x_2, x_3] \tag{3.36}$$

is defined as

$$\hat{x}_k = \sum_{j=0}^{3} x_j \omega^{jk} \pmod{7681}, \qquad k = 0, 1, 2, 3. \tag{3.37}$$

Using the powers of ω, the direct NTT matrix for $n = 4$ is Using the powers of ω, the direct NTT matrix for $n = 4$ is defined by

$$W_{k,j} = \omega^{kj} \pmod{7681}, \qquad k, j \in \{0, 1, 2, 3\}. \tag{3.38}$$

Thus,

$$\boldsymbol{W} = \begin{bmatrix} \omega^0 & \omega^0 & \omega^0 & \omega^0 \\ \omega^0 & \omega^1 & \omega^2 & \omega^3 \\ \omega^0 & \omega^2 & \omega^4 & \omega^6 \\ \omega^0 & \omega^3 & \omega^6 & \omega^9 \end{bmatrix} = \begin{bmatrix} 1 & 1 & 1 & 1 \\ 1 & 3383 & 7680 & 4298 \\ 1 & 7680 & 1 & 7680 \\ 1 & 4298 & 7680 & 3383 \end{bmatrix} \pmod{7681}. \tag{3.39}$$

Considering the input vectors

$$\boldsymbol{g} = [1, 2, 3, 4], \qquad \boldsymbol{h} = [5, 6, 7, 8], \tag{3.40}$$

their NTTs are computed as

$$\hat{\boldsymbol{g}} = \boldsymbol{W} \begin{bmatrix} 1 \\ 2 \\ 3 \\ 4 \end{bmatrix} \pmod{7681}. \tag{3.41}$$

Expanding each component gives

$$\hat{g}_0 = 1 + 2 + 3 + 4 = 10, \tag{3.42}$$
$$\hat{g}_1 = 1 + 2(3383) + 3(7680) + 4(4298) \equiv 913 \pmod{7681}, \tag{3.43}$$
$$\hat{g}_2 = 1 + 2(7680) + 3(1) + 4(7680) \equiv 7679 \pmod{7681}, \tag{3.44}$$
$$\hat{g}_3 = 1 + 2(4298) + 3(7680) + 4(3383) \equiv 6764 \pmod{7681}. \tag{3.45}$$

Thus,

$$\hat{\boldsymbol{g}} = [10, 913, 7679, 6764]. \tag{3.46}$$

Similarly,

$$\hat{\boldsymbol{h}} = \boldsymbol{W} \begin{bmatrix} 5 \\ 6 \\ 7 \\ 8 \end{bmatrix} \pmod{7681}. \tag{3.47}$$

Expanding the transform of $\boldsymbol{h}$ gives

$$\hat{h}_0 = 5 + 6 + 7 + 8 = 26, \tag{3.48}$$
$$\hat{h}_1 = 5 + 6(3383) + 7(7680) + 8(4298) \equiv 913 \pmod{7681}, \tag{3.49}$$
$$\hat{h}_2 = 5 + 6(7680) + 7(1) + 8(7680) \equiv 7679 \pmod{7681}, \tag{3.50}$$
$$\hat{h}_3 = 5 + 6(4298) + 7(7680) + 8(3383) \equiv 6764 \pmod{7681}. \tag{3.51}$$

Therefore,

$$\hat{\boldsymbol{h}} = [26, 913, 7679, 6764]. \tag{3.52}$$

By the convolution theorem, the circular convolution in the original domain is equivalent to an element-wise multiplication in the NTT domain. Hence, the Hadamard product is

$$\hat{\boldsymbol{g}} \circ \hat{\boldsymbol{h}} = \begin{bmatrix} 10 \times 26 \\ 913 \times 913 \\ 7679 \times 7679 \\ 6764 \times 6764 \end{bmatrix} \pmod{7681}. \tag{3.53}$$

Each product is reduced modulo 7681:

$$10 \times 26 \equiv 260 \pmod{7681}, \tag{3.54}$$
$$913 \times 913 = 833569 \equiv 4021 \pmod{7681}, \tag{3.55}$$
$$7679 \times 7679 = 58967041 \equiv 4 \pmod{7681}, \tag{3.56}$$
$$6764 \times 6764 = 45751696 \equiv 3660 \pmod{7681}. \tag{3.57}$$

Thus,

$$\hat{\boldsymbol{g}} \circ \hat{\boldsymbol{h}} = \begin{bmatrix} 260 \\ 4021 \\ 4 \\ 3660 \end{bmatrix}. \tag{3.58}$$

The result is then converted back to the original domain using the Inverse Number Theoretic Transform (INTT). The inverse root is

$$\omega^{-1} \equiv 4298 \pmod{7681}, \tag{3.59}$$

and the modular inverse of the transform length is

$$n^{-1} = 4^{-1} \pmod{7681} = 5761, \tag{3.60}$$

because

$$4 \times 5761 = 23044 \equiv 1 \pmod{7681}. \tag{3.61}$$

Therefore, the inverse NTT is computed as

$$\text{INTT}(\hat{\boldsymbol{g}} \circ \hat{\boldsymbol{h}}) = 5761 \begin{bmatrix} 1 & 1 & 1 & 1 \\ 1 & 4298 & 7680 & 3383 \\ 1 & 7680 & 1 & 7680 \\ 1 & 3383 & 7680 & 4298 \end{bmatrix} \begin{bmatrix} 260 \\ 4021 \\ 4 \\ 3660 \end{bmatrix} \pmod{7681}. \tag{3.62}$$

First, the matrix-vector multiplication gives

$$\begin{bmatrix} 1 & 1 & 1 & 1 \\ 1 & 4298 & 7680 & 3383 \\ 1 & 7680 & 1 & 7680 \\ 1 & 3383 & 7680 & 4298 \end{bmatrix} \begin{bmatrix} 260 \\ 4021 \\ 4 \\ 3660 \end{bmatrix} \equiv \begin{bmatrix} 264 \\ 272 \\ 264 \\ 240 \end{bmatrix} \pmod{7681}. \tag{3.63}$$

Multiplying by $n^{-1} = 5761$ yields

$$5761 \begin{bmatrix} 264 \\ 272 \\ 264 \\ 240 \end{bmatrix} \equiv \begin{bmatrix} 66 \\ 68 \\ 66 \\ 60 \end{bmatrix} \pmod{7681}. \tag{3.64}$$

Thus,

$$\text{INTT}(\hat{\boldsymbol{g}} \circ \hat{\boldsymbol{h}}) = \begin{bmatrix} 66 \\ 68 \\ 66 \\ 60 \end{bmatrix}. \tag{3.65}$$

This vector is the circular convolution of $\boldsymbol{g}$ and $\boldsymbol{h}$. Therefore, the NTT-based method produces the same result as the direct schoolbook computation, while replacing the convolution operation by two forward transforms, one element-wise multiplication, and one inverse transform.

3.5.8.6 Modulus Constraints and Convolution Methods

Implementing the Number Theoretic Transform (NTT) requires the modulus q to satisfy specific algebraic conditions. A modulus is classified as PWC-NTT friendly if a primitive n-th root of unity (ω) exists within the finite ring $\mathbb{Z}_q$. Conversely, Negative-Wrapped Convolution (NWC)—a standard optimization technique designed to handle modular reductions without doubling the transform size—strictly requires the existence of both an n-th root and a $2n$-th root of unity (ψ).

As established in Table 3.6, the standardized parameter set for ML-KEM dictates a modulus of $q = 3329$ and a degree of $n = 256$. While this specific ring successfully yields the ω necessary for Positive-Wrapped Convolutions, a $2n$-th root of unity (ψ) simply does not exist in $\mathbb{Z}_{3329}$. Consequently, any NWC-based architecture is mathematically incompatible with the current ML-KEM protocol. Given this fundamental limitation, NWC-NTT designs fall entirely outside the scope of this work, and the subsequent sections will analyze and develop hardware workflows based strictly on PWC-NTT frameworks.

Table 3.6: Standardized values of n and q for NIST-PQC schemes [Satriawan and Mareta, 2024].

Scheme	n	q	PWC-NTT Friendly	NWC-NTT Friendly
Dilithium	256	8380417	✓	✓
Falcon	512	12289	✓	✓
Falcon	1024	12289	✓	✓
ML-KEM (Versions 1 and 2)	256	7681	✓	✓
ML-KEM (Version 3)	256	3329	✓	×

3.5.8.7 Fast NTT: Adapting the Fast-Fourier Transform

Up to this point, the mathematical definitions of the NTT and INTT have relied on direct matrix-vector multiplications. While theoretically sound, this naive approach imposes

a quadratic computational cost of $O(n^2)$. To overcome this bottleneck, the structural equivalence between the NTT and the Discrete Fourier Transform (DFT) can be exploited. By mapping classic FFT optimization strategies—such as the Cooley-Tukey [Cooley and Tukey, 1965] and Gentleman-Sande [Gentleman and Sande, 1966] decimation algorithms—into finite rings, the operation is fundamentally restructured. These "Fast-NTT" architectures employ a divide-and-conquer approach, recursively breaking down the transform via butterfly networks to achieve a highly efficient $O(n \log n)$ complexity.

The core mechanism behind this acceleration relies on the inherent algebraic properties of the primitive $2n$-th root of unity (ψ). Specifically, the butterfly structures leverage its periodicity and symmetry to halve the computational workload at each algorithmic stage. For any non-negative integer k, these properties are formally defined in Equations 3.66 and 3.67.

$$\text{Periodicity:} \quad \psi^{k+2n} \equiv \psi^k \pmod{q} \tag{3.66}$$

$$\text{Symmetry:} \quad \psi^{k+n} \equiv -\psi^k \pmod{q} \tag{3.67}$$

By exploiting these identities, a n-point transform is divided into smaller $(n/2)$-point operations. It is important to note, however, that while both the forward NTT and the inverse INTT rely on this decimation concept, their specific routing techniques and scaling factors differ slightly during execution.

3.5.8.8 Cooley-Tukey (CT) Algorithm for Fast-NTT

To reduce the computational complexity from $\mathcal{O}(n^2)$ to $\mathcal{O}(n \log n)$, the Cooley-Tukey (CT) algorithm decomposes the discrete transform by partitioning the summation index based on parity. Splitting the sequence into even $(2i)$ and odd $(2i + 1)$ indices, and subsequently factoring out the common twiddle factor from the odd-indexed partition, yields Equation 3.68.

$$\begin{aligned} \hat{a}_j &= \sum_{i=0}^{n/2-1} \psi^{4ij+2i} a_{2i} + \sum_{i=0}^{n/2-1} \psi^{4ij+2j+2i+1} a_{2i+1} \pmod{q} \\ &= \sum_{i=0}^{n/2-1} \psi^{4ij+2i} a_{2i} + \psi^{2j+1} \sum_{i=0}^{n/2-1} \psi^{4ij+2i} a_{2i+1} \pmod{q} \end{aligned} \tag{3.68}$$

By exploiting the inherent symmetry of the primitive root ψ, evaluating the upper half of the output spectrum ($\hat{a}_{j+n/2}$) introduces a deterministic sign inversion across the odd-parity

summation components, as shown in Equation 3.69.

$$\hat{a}_{j+n/2} = \sum_{i=0}^{n/2-1} \psi^{4ij+2i} a_{2i} - \psi^{2j+1} \sum_{i=0}^{n/2-1} \psi^{4ij+2i} a_{2i+1} \pmod{q} \tag{3.69}$$

To map this mathematical decomposition onto a structured hardware datapath, the independent even and odd sub-summations are defined as intermediate variables A_j and B_j, respectively, where $A_j = \sum_{i=0}^{n/2-1} \psi^{4ij+2i} a_{2i}$ and $B_j = \sum_{i=0}^{n/2-1} \psi^{4ij+2i} a_{2i+1}$. Consequently, the expanded expressions reduce to the canonical Radix-2 Cooley-Tukey butterfly formulation, expressed in Equations 3.70 and 3.71.

$$\hat{a}_j = A_j + \psi^{2j+1} B_j \pmod{q} \tag{3.70}$$

$$\hat{a}_{j+n/2} = A_j - \psi^{2j+1} B_j \pmod{q} \tag{3.71}$$

This algebraic simplification demonstrates that both A_j and B_j can be computed recursively as independent $n/2$-point transforms, forming the algorithmic basis for the standard decimation-in-time processing element, visually represented in Figure 3.14.

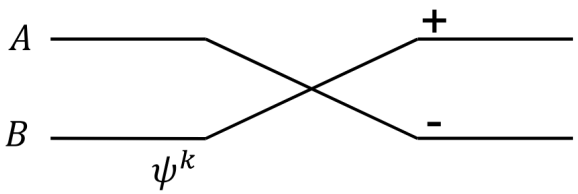


Figure 3.14: Cooley-Tukey (CT) butterfly unit for calculating NTT [Satriawan and Mareta, 2024].

3.5.9 ML-KEM Implementation of Fast-NTT

Standard literature often illustrates the Cooley-Tukey butterfly structure utilizing ψ, the $2n$-th root of unity required to compute a complete Negative-Wrapped Convolution (NWC). However, the specific parameter set of ML-KEM ($q = 3329$, $n = 256$) is not NWC-NTT friendly. Within this finite field, the 512-th root of unity is mathematically non-existent, making it impossible to instantiate ψ. Consequently, ML-KEM must perform an incomplete NTT.

Instead of ψ, the ML-KEM architecture relies on the principal n-th root of unity, defined as the integer 17 and denoted herein as ω. Despite the algorithmic absence of ψ, the structural logic of the standard Cooley-Tukey algorithm maps directly to the ML-KEM

parameter set via the established quadratic relation formalized in Equation 3.72.

$$\psi^2 \equiv \omega \pmod{q} \tag{3.72}$$

This mathematical equivalence proves that the foundational butterfly network can be seamlessly transposed to an ω-based datapath. Because ω is a primitive 256-th root of unity, its powers generate a cyclic subgroup yielding the periodic property $\omega^i \equiv \omega^{i+256} \pmod{q}$.

Crucially for hardware optimization, these roots exhibit exact algebraic symmetry. It must be emphasized that this property is not a universal occurrence, but a consequence of the chosen finite field parameters. Because the integer 17 is specifically a primitive 256-th root of unity in $\mathbb{Z}_{3329}$, evaluating it at its half-period deterministically yields $\omega^{n/2} \equiv \omega^{128} \equiv -1 \pmod{3329}$. Therefore, while the first 128 powers $(\omega^0, \omega^1, \ldots, \omega^{127})$ are unique elements in the field, multiplying any arbitrary power ω^i by the half-period naturally results in its additive inverse. This guarantees that the upper half of the sequence perfectly mirrors the first half with an inverted sign, as demonstrated in Equation 3.73.

$$\omega^{i+n/2} \equiv \omega^{i+128} \equiv -\omega^i \pmod{q} \quad \text{for} \quad 0 \leq i \leq 127 \tag{3.73}$$

By leveraging this deterministic symmetry, the hardware implementation can significantly reduce its memory footprint. Storing the complete set of 256 constants is highly redundant. The ω-based Cooley-Tukey architecture only requires fetching the first 128 twiddle factors from memory. By reading a single constant ω^i, the datapath applies its positive value to the addition branch and inherently evaluates the mirrored exponent (ω^{i+128}) by routing the same constant to the subtraction branch. This structural exploitation enables a highly optimized hardware footprint without compromising the mathematical correctness of the transform.

3.5.10 Example

To demonstrate the elimination of the constant ψ^3 in hardware through root symmetry, consider the transform operating over the finite ring $\mathbb{Z}_{13}$ with the primitive root $\psi = 5$. The precomputed constants are $\psi^1 = 5$ and $\psi^2 = 12 \equiv -1 \pmod{13}$.

Theoretically, the constant ψ^3 has a value of 8. However, the hardware architecture proposes substituting it with $-\psi^1$ (i.e., -5). To prove this equivalence, we first resolve the independent terms of the original equations, as shown in Equations 3.74, 3.75, and 3.76:

$$A = 1 + 3\psi^2 = 1 + 3(12) = 37 \equiv 11 \pmod{13} \tag{3.74}$$

$$B = 1 - 3\psi^2 = 1 - 3(12) = -35 \equiv 4 \pmod{13} \tag{3.75}$$

$$C = 2 + 4\psi^2 = 2 + 4(12) = 50 \equiv 11 \pmod{13} \tag{3.76}$$

The output equations $\hat{g}_1$ and $\hat{g}_3$ are the only ones that theoretically depend on ψ^3. Substituting the known values ($B = 4$ and $C = 11$), we calculate $\hat{g}_1$ using both methods in Equations 3.77 and 3.78 to prove the symmetry:

Theoretical Scenario (Using $\psi^3 = 8$):

$$\hat{g}_1 = B + \psi^3(C) = 4 + 8(11) = 4 + 88 = 92 \equiv 1 \pmod{13} \tag{3.77}$$

Hardware Scenario (Substituting ψ^3 with $-\psi^1 = -5$):

$$\hat{g}_1 = B - \psi^1(C) = 4 - 5(11) = 4 - 55 = -51 \equiv 1 \pmod{13} \tag{3.78}$$

Performing the same proof for equation $\hat{g}_3$, as detailed in Equations 3.79 and 3.80:

Theoretical Scenario (Using $\psi^3 = 8$):

$$\hat{g}_3 = B - \psi^3(C) = 4 - 8(11) = 4 - 88 = -84 \equiv 7 \pmod{13} \tag{3.79}$$

Hardware Scenario (Substituting ψ^3 with $-\psi^1 = -5$):

$$\hat{g}_3 = B - (-\psi^1)(C) = 4 + 5(11) = 4 + 55 = 59 \equiv 7 \pmod{13} \tag{3.80}$$

The final results $\hat{g}_1 = 1$ and $\hat{g}_3 = 7$ are identical in both scenarios. This numerical proof demonstrates that the hardware can discard the storage of ψ^3, relying solely on the constants ψ^1 and ψ^2, and inverting the algebraic sign in the datapath to achieve the exact same result within the modular ring.

3.5.11 Matrix Representation of the $N = 8$ Butterfly Network

For a transform of length $n = 8$, the twiddle factor matrix and the input vector expand to an 8×8 dimension. Evaluating the exponents through the relation $2ij + i$ yields the initial power matrix presented in Equation 3.81:

$$\hat{\boldsymbol{g}} = \begin{bmatrix} \psi^0 & \psi^1 & \psi^2 & \psi^3 & \psi^4 & \psi^5 & \psi^6 & \psi^7 \\ \psi^0 & \psi^3 & \psi^6 & \psi^9 & \psi^{12} & \psi^{15} & \psi^{18} & \psi^{21} \\ \psi^0 & \psi^5 & \psi^{10} & \psi^{15} & \psi^{20} & \psi^{25} & \psi^{30} & \psi^{35} \\ \psi^0 & \psi^7 & \psi^{14} & \psi^{21} & \psi^{28} & \psi^{35} & \psi^{42} & \psi^{49} \\ \psi^0 & \psi^9 & \psi^{18} & \psi^{27} & \psi^{36} & \psi^{45} & \psi^{54} & \psi^{63} \\ \psi^0 & \psi^{11} & \psi^{22} & \psi^{33} & \psi^{44} & \psi^{55} & \psi^{66} & \psi^{77} \\ \psi^0 & \psi^{13} & \psi^{26} & \psi^{39} & \psi^{52} & \psi^{65} & \psi^{78} & \psi^{91} \\ \psi^0 & \psi^{15} & \psi^{30} & \psi^{45} & \psi^{60} & \psi^{75} & \psi^{90} & \psi^{105} \end{bmatrix} \begin{bmatrix} 1 \\ 2 \\ 3 \\ 4 \\ 5 \\ 6 \\ 7 \\ 8 \end{bmatrix} \tag{3.81}$$

Because the transform operates over a ring of size $2n = 16$, the powers of the primitive root are strictly periodic, satisfying $\psi^{16} = 1$. Applying a modulo 16 reduction to all exponents simplifies the matrix to Equation 3.82:

$$\hat{\boldsymbol{g}} = \begin{bmatrix} \psi^0 & \psi^1 & \psi^2 & \psi^3 & \psi^4 & \psi^5 & \psi^6 & \psi^7 \\ \psi^0 & \psi^3 & \psi^6 & \psi^9 & \psi^{12} & \psi^{15} & \psi^2 & \psi^5 \\ \psi^0 & \psi^5 & \psi^{10} & \psi^{15} & \psi^4 & \psi^9 & \psi^{14} & \psi^3 \\ \psi^0 & \psi^7 & \psi^{14} & \psi^5 & \psi^{12} & \psi^3 & \psi^{10} & \psi^1 \\ \psi^0 & \psi^9 & \psi^2 & \psi^{11} & \psi^4 & \psi^{13} & \psi^6 & \psi^{15} \\ \psi^0 & \psi^{11} & \psi^6 & \psi^1 & \psi^{12} & \psi^7 & \psi^2 & \psi^{13} \\ \psi^0 & \psi^{13} & \psi^{10} & \psi^7 & \psi^4 & \psi^1 & \psi^{14} & \psi^{11} \\ \psi^0 & \psi^{15} & \psi^{14} & \psi^{13} & \psi^{12} & \psi^{11} & \psi^{10} & \psi^9 \end{bmatrix} \begin{bmatrix} 1 \\ 2 \\ 3 \\ 4 \\ 5 \\ 6 \\ 7 \\ 8 \end{bmatrix} \tag{3.82}$$

Finally, the symmetry property of the negative-wrapped convolution is applied, where a half-cycle shift results in an algebraic sign inversion ($\psi^{k+8} = -\psi^k$). By substituting all exponents greater than or equal to 8, the matrix reveals the optimized structure in Equation 3.83 that directly maps onto the hardware butterfly architecture:

$$\hat{\boldsymbol{g}} = \begin{bmatrix} \psi^0 & \psi^1 & \psi^2 & \psi^3 & \psi^4 & \psi^5 & \psi^6 & \psi^7 \\ \psi^0 & \psi^3 & \psi^6 & -\psi^1 & -\psi^4 & -\psi^7 & \psi^2 & \psi^5 \\ \psi^0 & \psi^5 & -\psi^2 & -\psi^7 & \psi^4 & -\psi^1 & -\psi^6 & \psi^3 \\ \psi^0 & \psi^7 & -\psi^6 & \psi^5 & -\psi^4 & \psi^3 & -\psi^2 & \psi^1 \\ \psi^0 & -\psi^1 & \psi^2 & -\psi^3 & \psi^4 & -\psi^5 & \psi^6 & -\psi^7 \\ \psi^0 & -\psi^3 & \psi^6 & \psi^1 & -\psi^4 & \psi^7 & \psi^2 & -\psi^5 \\ \psi^0 & -\psi^5 & -\psi^2 & \psi^7 & \psi^4 & \psi^1 & -\psi^6 & -\psi^3 \\ \psi^0 & -\psi^7 & -\psi^6 & -\psi^5 & -\psi^4 & -\psi^3 & -\psi^2 & -\psi^1 \end{bmatrix} \begin{bmatrix} 1 \\ 2 \\ 3 \\ 4 \\ 5 \\ 6 \\ 7 \\ 8 \end{bmatrix} \tag{3.83}$$

Figure 3.15 depicts the butterfly structure together with the associated coefficient terms, corresponding to the transformation matrix presented in Equation 3.83.

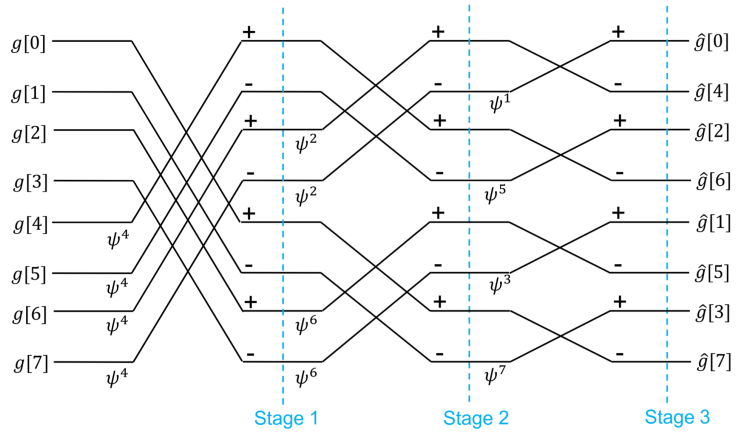


Figure 3.15: CT butterfly structure for the NTT with natural-order input (NO-input) and bit-reversed-order output (BO-output), for $n = 8$ [Satriawan and Mareta, 2024].

Tracing the signal propagation through the architectural datapath provides a direct mapping to the underlying system of equations. Figure 3.16 illustrates this structural dataflow for a transform of length $n = 4$.

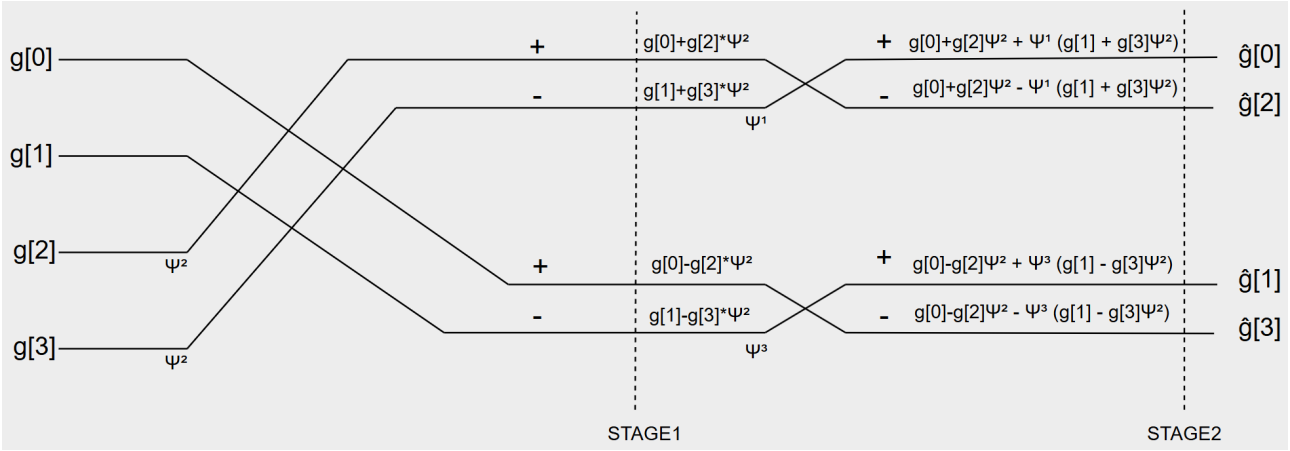


Figure 3.16: Dataflow of the Cooley-Tukey butterfly architecture for an $n = 4$ Number Theoretic Transform, detailing the intermediate algebraic evaluations across stages. Source: Author.

By resolving the operational nodes at the final stage of the network, the equivalent algebraic system for the transformed output coefficients is explicitly formulated in Equations 3.84 through 3.87:

$$\hat{g}_0 = \psi^0(g[0] + g[2]\psi^2) + \psi^1(g[1] + g[3]\psi^2) \tag{3.84}$$

$$\hat{g}_1 = \psi^0(g[0] - g[2]\psi^2) + \psi^3(g[1] - g[3]\psi^2) \tag{3.85}$$

$$\hat{g}_2 = \psi^0(g[0] + g[2]\psi^2) - \psi^1(g[1] + g[3]\psi^2) \tag{3.86}$$

$$\hat{g}_3 = \psi^0(g[0] - g[2]\psi^2) - \psi^3(g[1] - g[3]\psi^2) \tag{3.87}$$

4. INTEGRATION OF CRYSTALS-KYBER IN RS5

This chapter details the implementations made in the Kyber instruction set accelerator, specifically focusing on the Xkyber extension proposed by [Gewehr et al., 2024]. The Xkyber extension introduces six new custom instructions designed to optimize Crystals-Kyber operations within 32-bit RISC-V processors.

4.1 XKyber ISE

The instruction set extension proposed by [Gewehr et al., 2024] introduces six new custom instructions for the RISC-V architecture. The following text details the functionality of each instruction and explains how they are integrated into the processor environment.

4.1.1 KYBERCBD2

The first custom instruction, `kybercbd2`, was designed to accelerate the pseudo-random sampling of polynomial coefficients from a Centered Binomial Distribution (CBD). It specifically targets operations where the distribution parameter η is set to 2. The procedure for this sampling process is outlined in Algorithm 4.1.

Algorithm 4.1: SamplePolyCBD$_\eta(B)$. [NIST, 2024a]

Takes a seed as input and outputs a pseudorandom sample from the distribution $\mathcal{D}_\eta(R_q)$.
Input: byte array $B \in \mathbb{B}^{64\eta}$.
Output: array $f \in \mathbb{Z}_q^{256}$. ▷ the coefficients of the sampled polynomial

1: $b \leftarrow$ BytesToBits(B)
2: **for** $(i \leftarrow 0; i < 256; i$++$)$ **do**
3: $\quad x \leftarrow \sum_{j \leftarrow 0}^{\eta-1} b[2i\eta + j]$ ▷ $0 \leq x \leq \eta$
4: $\quad y \leftarrow \sum_{j \leftarrow 0}^{\eta-1} b[2i\eta + \eta + j]$ ▷ $0 \leq y \leq \eta$
5: $\quad f[i] \leftarrow x - y \bmod q$ ▷ $0 \leq f[i] \leq \eta$ or $q - \eta \leq f[i] \leq q - 1$
6: **end for**
7: **return** f

4.1.2 KYBERCBD3

The Kyber specification defines different η values depending on the target security level and the specific operational phase (as summarized previously in Table 3.5). While

`kybercbd2` handles the $\eta = 2$ cases, Kyber-512 also requires sampling with $\eta = 3$ for its primary error vectors.

To support this variation in hardware, the `kybercbd3` instruction was introduced. It executes the same procedure detailed in Algorithm 4.1, but is strictly parameterized for $\eta = 3$. Similar to its counterpart, this instruction takes advantage of the 12-bit coefficient size to sample and process two coefficients in parallel within a single 32-bit register, optimizing the overall throughput of the sampling phase.

4.1.3 KYBERCBD2 and KYBERCBD3

Both `kybercbd2` and `kybercbd3` are designed to operate as SIMD instructions. At first glance, one might assume a higher degree of parallelization is possible due to the minimal input requirements: CBD2 only needs 4 bits to generate a coefficient, while CBD3 requires 6 bits. However, the actual bottleneck that dictates the parallelization limit is the output bandwidth.

Since every sampled coefficient must be formatted as a 12-bit integer, a standard 32-bit word can only accommodate a maximum of two complete results simultaneously. Figure 4.1 illustrates this architectural constraint, showing how the relatively small input seeds are packed into the source register (`RS1`), while the destination register (`RD`) reaches its capacity by isolating the two resulting 12-bit coefficients.

4.1.4 KYBERCOMPRESS

The `kybercompress` instruction is designed to accelerate the polynomial coefficient compression required during the encryption and decryption phases (Algorithms 3.8 and 3.9, respectively). The mathematical formulation of this operation is outlined in Algorithm 4.2.

At first glance, a SIMD approach for this instruction appears highly appealing. Since the input is a 12-bit coefficient and the compressed output requires at most 11 bits ($d \leq 11$), two results could theoretically fit within a single 32-bit register. However, the hardware and area implications of parallelizing this execution were not evaluated within the scope of this work. Consequently, `kybercompress` is implemented as a strictly scalar (non-SIMD) instruction, leaving potential SIMD optimizations as an avenue for future research.

Algorithm 4.2: *Compress*$_d$ Kyber Coefficient Compression. [Gewehr et al., 2024].

Require: x is a polynomial coefficient $\in \mathbb{Z}_q$
Ensure: Returns an element $\in \mathbb{Z}_{2^d}$
return $\lceil (2^d/q) \cdot x \rfloor \bmod 2^d$

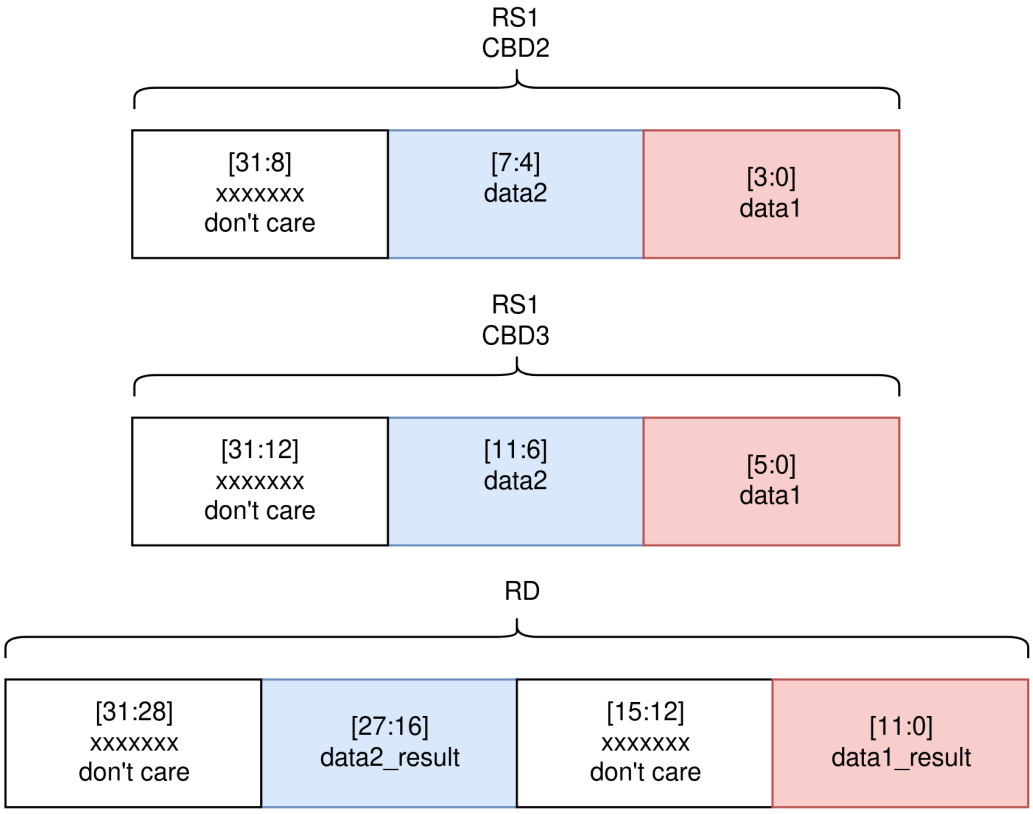


Figure 4.1: Data packing structure for CBD sampling instructions. The source register (`RS1`) holds the compact input seeds, while the destination register (`RD`) stores the two resulting 12-bit coefficients (`data1_result` and `data2_result`). Source: Author.

4.1.5 KYBERADD

The `kyberadd` instruction performs modular addition within the polynomial ring $\mathbb{Z}_q$, where $q = 3329$. Because this specific modulus requires only 12 bits of length, the 32-bit word length of our SoC provides an excellent opportunity for hardware optimization. We can easily pack two independent 12-bit coefficients into a single 32-bit register, which allows the instruction to operate in a Single Instruction Multiple Data (SIMD) fashion.

As illustrated in Figure 4.2, `kyberadd` extracts operands from bits [11:0] and [27:16] of the source registers (`RS1` and `RS2`) to compute two modular additions concurrently. This parallel execution reuses the processor's existing 32-bit ALU. To make it work, a simple mask is applied to prevent carry propagation from the lower 16-bit boundary into the upper half. Finally, the modular reduction is handled efficiently via conditional subtraction, guaranteeing that both outputs stay within the valid $\mathbb{Z}_q$ range before they are written back to the destination register.

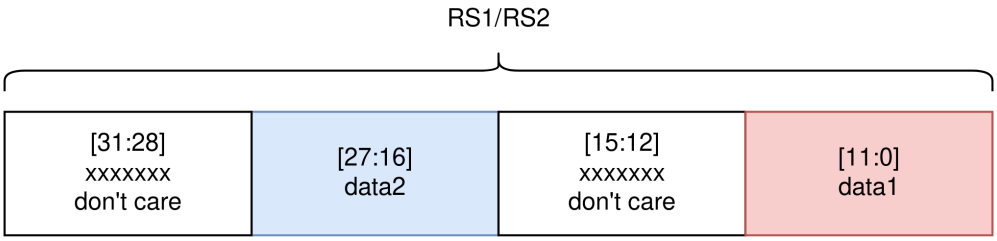


Figure 4.2: Data packing structure within the 32-bit source registers (`RS1`/`RS2`) for KYBERADD execution. The 12-bit coefficients (`data1` and `data2`) are isolated in bits [11:0] and [27:16]. Source: Author.

4.1.6 KYBERSUB

Complementing the addition operation, the `kybersub` instruction performs modular subtraction within the polynomial ring $\mathbb{Z}_q$. Because it operates under the exact same data constraints as its addition counterpart, `kybersub` also executes as a SIMD instruction, computing two independent 12-bit modular subtractions in parallel.

To handle the modular arithmetic efficiently at the hardware level, the instruction employs a conditional addition approach. It adds the modulus q (3329) whenever an intermediate subtraction results in a negative value, guaranteeing that the final coefficients strictly remain within the valid $[0, 3328]$ range. Since it processes two coefficients concurrently, the data packing structure for the source registers is identical to the one previously illustrated in Figure 4.2.

4.1.7 KYBERMUL

The final instruction, `kybermul`, executes a modular multiplication between two 12-bit coefficients within $\mathbb{Z}_q$. Unlike modular addition or subtraction, multiplication represents a significant bottleneck in hardware implementations of lattice-based cryptography. The core challenge arises because the intermediate product of two coefficients expands to 24 bits (reaching up to maximum number of 3328×3328). Consequently, the required modular reduction cannot be resolved with a simple conditional subtraction. This specific step attempts to balance fast execution with low silicon area overhead.

To solve this without increasing the processor's area, `kybermul` operates as a non-SIMD instruction that implements the Barrett reduction, Algorithm 4.3. Instead of relying on a large, dedicated modular multiplier, the instruction completes the entire operation over multiple cycles by sharing resources with the processor's existing 16-bit multiplier from RS5.

Algorithm 4.3: Multiplication Mod q (Barrett Reduction [Gewehr et al., 2024])

Require: $a, b \in \mathbb{Z}_q$, $k \approx 2 \cdot \lceil \log_2 q \rceil$, $m = \lfloor 2^k / q \rfloor = 5039$
Ensure: $z \in \mathbb{Z}_q$
$temp_0 \leftarrow a \cdot b$
$temp_1 \leftarrow (temp_0 \cdot m) \gg k$
$temp_1 \leftarrow (temp_1 \cdot q)$
return $((temp_0 - temp_1) \geq q)\ ?\ (temp_0 - temp_1) - q : (temp_0 - temp_1)$

4.1.8 Acceleration of NTT

As established earlier, polynomial multiplication—and by extension, the Number Theoretic Transform (NTT)—represents one of the most significant computational bottlenecks in Kyber, as demonstrated by Di Matteo et al. [2024].To tackle this, the `kyberadd`, `kybersub`, and `kybermul` instructions were designed to optimize the core building block of the transform: the butterfly unit.

As shown in Figure 4.3, both the Cooley-Tukey (CT) and Gentleman-Sande (GS) butterflies are composed entirely of modular addition, subtraction, and multiplication. By mapping these mathematical steps to our custom hardware instructions, the processor reduces the latency of each individual butterfly execution.

This optimization scales due to the NTT structure. Rather than computing a naive $O(n^2)$ multiplication, the algorithm organizes these butterflies in a layered, iterative manner, computing $n/2 = 128$ parallel butterflies per layer across $\log_2(n)$ layers. While the theoretical time complexity remains $O(n \log n)$, accelerating the butterfly with these operations, drastically cuts down the absolute execution time of the entire polynomial multiplication phase.

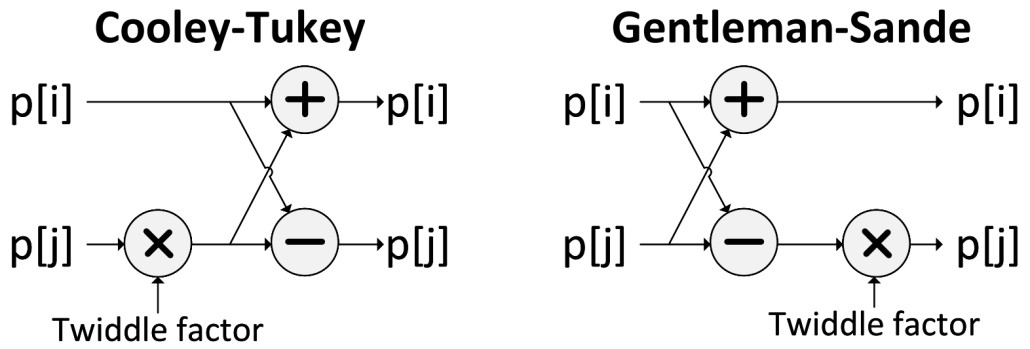


Figure 4.3: Data flow diagrams for the Cooley-Tukey (CT) and Gentleman-Sande (GS) butterfly operations utilized in the NTT and INTT computations, respectively. [Gewehr et al., 2024].

4.2 Integration into RS5

This section details the methodology applied to integrate the custom Xkyber instruction set into the RS5 processor [Nunes et al., 2024]. The following text outlines the hardware modifications and design decisions required to adapt the processor's decoding logic and execution block to support the new cryptographic operations.

4.2.1 Modifications on the RISC-V Compiler

To enable the standard RISC-V GNU Toolchain to recognize and assemble the custom Xkyber instructions, the compiler's opcode definitions had to be modified and recompiled. This integration process is relatively straightforward and primarily involves updating the `riscv-opc.c` source file and its corresponding header (`riscv-opc.h`).

For each new instruction, a unique `MATCH` and `MASK` macro must be defined in the header file. To understand this encoding mechanism, we can analyze the standard `ADD` instruction as a baseline:

```
#define MATCH_ADD 0x33
#define MASK_ADD  0xfe00707f
```

Table 4.1: Field encoding of the `ADD` instruction. [RISC-V, 2024]

ADD	0000000	rs2	rs1	000	rd	0110011

As demonstrated in Table 4.1, the `MATCH_ADD` hexadecimal value (`0x33`) corresponds directly to the fixed 7-bit opcode (`0110011`). The `MASK_ADD` (`0xFE00707F`) acts as a bitwise filter. It places logical '1's over the fixed architectural bits (opcode, `funct3`, and `funct7`) and '0's over the variable operand fields (`rd`, `rs1`, and `rs2`). During the assembly or disassembly process, the toolchain applies a bitwise AND operation between the fetched instruction and this mask; if the result matches the predefined `MATCH` value, the instruction is successfully identified.

Building upon this bitwise matching logic, the specific hexadecimal definitions for the Xkyber extension were integrated into the `riscv-opc.h` header. To guarantee that the new operations would not conflict with the standard RISC-V Base Integer Instruction Set, the entire cryptographic extension was mapped to the `custom-0` opcode space (`0x2B`).

As shown in the code snippet below, instructions that follow a standard register-to-register architecture (`kyberadd`, `kybersub`, `kybermul`, and `kybercompress`) utilize the traditional R-type mask (`0xFE00707F`). This standard mask allows the toolchain to differentiate

the operations using only their `funct7` and `funct3` fields. Conversely, the `kybercbd2` and `kybercbd3` instructions require a stricter mask (`0xFFF0707F`), which locks the entire upper 12 bits to uniquely identify their specific data sampling configurations.

```
/* Kyber Instructions */
#define MATCH_KYBERADD 0x602b
#define MASK_KYBERADD 0xfe00707f
#define MATCH_KYBERCBD2 0x820602b
#define MASK_KYBERCBD2 0xfff0707f
#define MATCH_KYBERCBD3 0x830602b
#define MASK_KYBERCBD3 0xfff0707f
#define MATCH_KYBERCOMPRESS 0x600602b
#define MASK_KYBERCOMPRESS 0xfe00707f
#define MATCH_KYBERMUL 0x400602b
#define MASK_KYBERMUL 0xfe00707f
#define MATCH_KYBERSUB 0x200602b
#define MASK_KYBERSUB 0xfe00707f
```

With the matching logic established in the header, the final step to correctly rebuild the compiler toolchain involves mapping the instructions within the `riscv-opc.c` source file. This file contains the primary array where the assembler looks up the syntax, architecture restrictions, and characteristics of every supported opcode.

As illustrated in the code block below, each instruction is declared as a C structure containing its mnemonic, architecture support, instruction class, operand format, and the previously defined MACROS.

```
/* Kyber instructions. */
{"kybercbd2",     32, INSN_CLASS_I, "d,s",   MATCH_KYBERCBD2,
  MASK_KYBERCBD2, match_opcode, 0 },
{"kybercbd3",     32, INSN_CLASS_I, "d,s",   MATCH_KYBERCBD3,
  MASK_KYBERCBD3, match_opcode, 0 },
{"kybercompress", 32, INSN_CLASS_I, "d,s,t", MATCH_KYBERCOMPRESS,
  MASK_KYBERCOMPRESS, match_opcode, 0 },
{"kyberadd",      32, INSN_CLASS_I, "d,s,t", MATCH_KYBERADD,
  MASK_KYBERADD, match_opcode, 0 },
{"kybermul",      32, INSN_CLASS_I, "d,s,t", MATCH_KYBERMUL,
  MASK_KYBERMUL, match_opcode, 0 },
{"kybersub",      32, INSN_CLASS_I, "d,s,t", MATCH_KYBERSUB,
  MASK_KYBERSUB, match_opcode, 0 },
```

A structural analysis of these definitions reveals how the assembler handles the new extension compared to a standard instruction like `add` (e.g., `{"add", 0, INSN_CLASS_I, "d,s,t", ...}`):

- **Register Width (`xlen`) (Field 2):** The `xlen` parameter defines the required width of the integer registers in bits. By setting this field to `32`, the Xkyber instructions are explicitly restricted to 32-bit architectures (RV32). In contrast, standard instructions like `add` use `0` to indicate that they are agnostic to the register width, allowing them to be seamlessly compiled for RV32, RV64, or RV128 platforms.
- **ISA Extension (Field 3):** The `INSN_CLASS_I` parameter defines the specific ISA extension to which the instruction belongs. The custom Xkyber instructions are integrated directly into the base Integer (I) extension.
- **Operand Syntax (Field 4):** The string dictates the expected assembly syntax format. The `"d,s,t"` format tells the parser that the instruction requires a destination register (`rd`), a first source register (`rs1`), and a second source register (`rs2`).

Noticeably, the `kybercbd2` and `kybercbd3` instructions use the abbreviated `"d,s"` format. This accurately reflects their specific hardware implementation, as the binomial sampling process only requires a single source register holding the seed to produce the formatted output.

4.2.2 ZKNH Extension

The NIST Suite: Hash Function Instructions(ZKNH) extension was the first cryptographic instruction set integrated into the RS5 processor, specifically designed to accelerate the execution of the SHA-2 hash functions. As a standardized component of the RISC-V scalar cryptography extension, the RV32I instruction set includes four dedicated instructions for SHA-256 and six for SHA-512, as detailed in Table 4.2. These algorithms differ primarily in their internal state size and final digest length, producing 256-bit and 512-bit outputs, respectively. The larger digest of SHA-512 translates to a higher security margin against collision attacks.

At their core, the SHA-2 cryptographic hash functions rely on iterative sequences of bitwise shifts, rotations, and XOR operations. Executing these operations directly on dedicated hardware significantly reduces clock-cycle overhead compared to standard software implementations. Algorithm 4.4 describes the combinational logic mapped inside the RS5 hardware module to accelerate these functions.

Table 4.2: Cryptographic extension instructions (SHA-256 and SHA-512) supported by the RV32 architecture of the RS5 processor. [RISC-V, 2021].

Mnemonic	Algorithm Description	Mathematical Definition
`sha256sig0`	SHA-256 Sigma 0 (σ_0)	eq. (3.3)
`sha256sig1`	SHA-256 Sigma 1 (σ_1)	eq. (3.3)
`sha256sum0`	SHA-256 Sum 0 (Σ_0)	eq. (3.5)
`sha256sum1`	SHA-256 Sum 1 (Σ_1)	eq. (3.5)
`sha512sig0h`	SHA-512 Sigma 0 (High)	
`sha512sig0l`	SHA-512 Sigma 0 (Low)	
`sha512sig1h`	SHA-512 Sigma 1 (High)	
`sha512sig1l`	SHA-512 Sigma 1 (Low)	
`sha512sum0r`	SHA-512 Sum 0 (Register pair)	
`sha512sum1r`	SHA-512 Sum 1 (Register pair)	

Algorithm 4.4: Hardware Module of SHA-256 and SHA-512 Instructions.

Require: $op\_a, op\_b \in \mathbb{Z}_{2^{32}}$, and operation selector *sha2_op*
Ensure: $result \in \mathbb{Z}_{2^{32}}$
Define $\text{ROTR}(x, n) = (x \gg n) \vee (x \ll (32 - n))$
Switch *sha2_op* **do**
  Case `SIG0`:
    return $\text{ROTR}(op\_a, 7) \oplus \text{ROTR}(op\_a, 18) \oplus (op\_a \gg 3)$
  Case `SIG1`:
    return $\text{ROTR}(op\_a, 17) \oplus \text{ROTR}(op\_a, 19) \oplus (op\_a \gg 10)$
  Case `SUM0`:
    return $\text{ROTR}(op\_a, 2) \oplus \text{ROTR}(op\_a, 13) \oplus \text{ROTR}(op\_a, 22)$
  Case `SUM1`:
    return $\text{ROTR}(op\_a, 6) \oplus \text{ROTR}(op\_a, 11) \oplus \text{ROTR}(op\_a, 25)$
  Case `SIG0H`:
    return $(op\_a \gg 1) \oplus (op\_a \gg 7) \oplus (op\_a \gg 8) \oplus (op\_b \ll 31) \oplus (op\_b \ll 24)$
  Case `SIG0L`:
    return $(op\_a \gg 1) \oplus (op\_a \gg 7) \oplus (op\_a \gg 8) \oplus (op\_b \ll 31) \oplus (op\_b \ll 25) \oplus (op\_b \ll 24)$
  Case `SIG1H`:
    return $(op\_a \ll 3) \oplus (op\_a \gg 6) \oplus (op\_a \gg 19) \oplus (op\_b \gg 29) \oplus (op\_b \ll 13)$
  Case `SIG1L`:
    return $(op\_a \ll 3) \oplus (op\_a \gg 6) \oplus (op\_a \gg 19) \oplus (op\_b \gg 29) \oplus (op\_b \ll 26) \oplus (op\_b \ll 13)$
  Case `SUM0R`:
    return $(op\_a \ll 25) \oplus (op\_a \ll 30) \oplus (op\_a \gg 28) \oplus (op\_b \gg 7) \oplus (op\_b \gg 2) \oplus (op\_b \ll 4)$
  Case `SUM1R`:
    return $(op\_a \ll 23) \oplus (op\_a \gg 14) \oplus (op\_a \gg 18) \oplus (op\_b \gg 9) \oplus (op\_b \ll 18) \oplus (op\_b \ll 14)$

4.2.3 ZBKB Extension

Following the integration of the ZKNH extension for SHA-2, the RS5 processor was further augmented with the ZBKB extension, which provides Bit Manipulation Instructions for Cryptography. While the previous section detailed specialized hardware acceleration for specific hash implementation (SHA2), the ZBKB extension introduces bitwise operations, as summarized in Table 4.3. These instructions are utilized within the SHA-3 (Keccak) workflow to manage the state permutations required by the sponge construction.

Table 4.3: Bit manipulation and scalar cryptography instructions supported by the RV32 architecture of the RS5 processor. [RISC-V, 2021].

Mnemonic	**Instruction Description**
`ror`	Rotate right (Register)
`rol`	Rotate left (Register)
`rori`	Rotate right (Immediate)
`andn`	AND with inverted operand
`orn`	OR with inverted operand
`xnor`	Exclusive NOR
`pack`	Pack low halves of registers
`packh`	Pack low bytes of registers
`brev8`	Reverse bits in bytes
`rev8`	Byte-reverse register
`zip`	Zip
`unzip`	Unzip

4.2.3.1 ROL, ROR

To minimize logic overhead and save silicon area, the rotation instructions (`rol` and `ror`) were designed to bypass the need for dedicated shifters. Instead, they reuse the existing logical shift-left (`sll`) and shift-right (`srl`) data paths already present in the RV32I instruction set.

With this architecture, the only required hardware adjustment is establishing the circular routing—ensuring that bits shifted out of one end of the register are systematically wrapped around and fed into the opposite end. As demonstrated in Algorithm 4.5, this circular feature is achieved through parallel combinational logic, merging the standard and complementary shifts via bitwise OR operations.

Algorithm 4.5: Hardware Execution of Rotate Right (ROR) and Rotate Left (ROL) Instructions.

Require: Data operand $rs1 \in \mathbb{Z}_{2^{32}}$, shift amount $N \in [0, 31]$, and control signal *is_rol*
Ensure: *shift_result* $\in \mathbb{Z}_{2^{32}}$

▷ Calculate complementary shift amount

$N_{comp} \leftarrow 32 - N$
sll_result $\leftarrow (rs1 \ll N)$
srl_result $\leftarrow (rs1 \gg N)$

▷ Parallel evaluation of rotation data paths

ror_result $\leftarrow$ *srl_result* $\vee (rs1 \ll N_{comp})$
rol_result $\leftarrow$ *sll_result* $\vee (rs1 \gg N_{comp})$

▷ Multiplexed output based on opcode

if *is_rol* **then**
    shift_result $\leftarrow$ *rol_result*
else
    shift_result $\leftarrow$ *ror_result*
end if
return *shift_result*

4.2.3.2 RORI

The `rori` (Rotate Right Immediate) instruction utilizes the exact same combinational data path and circular routing defined in Algorithm 4.5. From a hardware perspective, the core execution unit remains unchanged. The fundamental difference between the register-based (`ror`) and immediate-based (`rori`) operations lies entirely in the instruction decoding stage. For the standard `ror` instruction, the shift amount (N) is extracted from the lower 5 bits of the source register `rs2`. Conversely, for the `rori` instruction, the hardware multiplexers route the N value directly from the immediate `shamt` field encoded within the instruction itself.

4.2.3.3 ANDN, ORN, and XNOR

The Zbkb extension introduces three fundamental logic operations with inverted operands: `andn`, `orn`, and `xnor`. At the hardware level, the execution flow for these instructions is straightforward. Instead of deploying dedicated execution units, the architecture minimizes logic overhead by applying a bitwise NOT operation ($\neg$) to the second source register ($rs2$) before routing the data into the standard AND, OR, and XOR logic gates. This multiplexed evaluation approach is detailed in Algorithm 4.6.

Algorithm 4.6: Hardware Execution of Logic with Inverted Operand Instructions.

Require: Data operands $rs1, rs2 \in \mathbb{Z}_{2^{32}}$, and operation selector *logic_op*
Ensure: $result \in \mathbb{Z}_{2^{32}}$

▷ Invert the second operand

$rs2_{inv} \leftarrow \neg rs2$

▷ Multiplexed bitwise logic evaluation

Switch *logic_op* **do**
  Case `ANDN`:
    return $rs1 \wedge rs2_{inv}$
  Case `ORN`:
    return $rs1 \vee rs2_{inv}$
  Case `XNOR`:
    return $rs1 \oplus rs2_{inv}$

4.2.3.4 PACK, PACKH

The `pack` and `packh` instructions execute bitwise concatenations utilizing the lower segments of the source registers, `rs1` and `rs2`. Specifically, the `pack` operation extracts the 16 least significant bits (LSBs) from both source registers and concatenates them to form a complete 32-bit word in the destination register (`rd`), as illustrated in Figure 4.4.

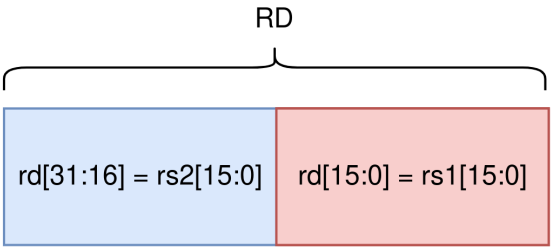


Figure 4.4: Bitwise concatenation for the `pack` instruction. Source: Author.

The `packh` instruction operates on the same principle but at the byte level. It concatenates the 8 LSBs of `rs1` with the 8 LSBs of `rs2`, placing the resulting 16-bit value into the lower half of `rd`, while the upper 16 bits are zero-extended, as shown in Figure 4.5.

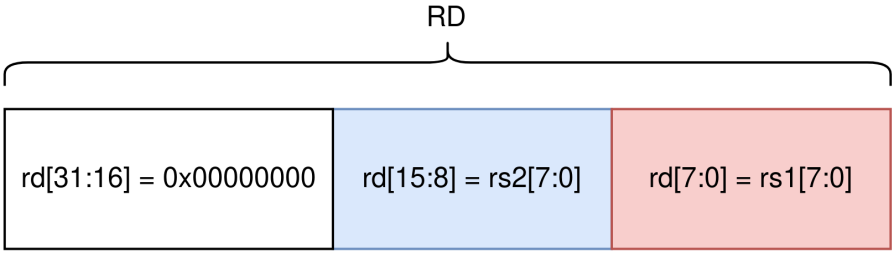


Figure 4.5: Byte-level concatenation and zero-extension for the `packh` instruction. Source: Author.

Ultimately, the physical hardware mirrors the data paths illustrated in the previous figures, resulting in a straightforward architectural design. As formalized in Algorithm 4.7, this operation requires no complex combinatorial logic, relying entirely on direct wire mapping.

Algorithm 4.7: Hardware Execution of PACK and PACKH Instructions.

Require: Data operands $rs1, rs2 \in \mathbb{Z}_{2^{32}}$, and control signal *is_packh*
Ensure: $pack\_result \in \mathbb{Z}_{2^{32}}$

▷ Multiplexed wire routing and concatenation
if *is_packh* **then**
    $pack\_result \leftarrow \{0x0000, rs2[7:0], rs1[7:0]\}$
else ▷ Default to standard PACK operation
    $pack\_result \leftarrow \{rs2[15:0], rs1[15:0]\}$
end if
return *pack_result*

4.2.3.5 BREV8 and REV8

The `brev8` and `rev8` instructions are designed to perform distinct structural reversals within a 32-bit register. It is crucial to differentiate their operational scope: one manipulates bit-level endianness, while the other manipulates byte-level endianness.

The `brev8` (Byte-Reverse Bits) instruction operates exclusively within the boundaries of each individual byte. It reverses the internal bit sequence of all four bytes in parallel, without altering the byte's physical position within the 32-bit word. As illustrated in Figure 4.6, the byte at `rs1[31:24]` remains mapped to `rd[31:24]`, but its internal bits are mirrored.

Conversely, the `rev8` (Reverse Bytes) instruction performs a complete structural endianness swap at the word level. It reverses the order of the bytes across the entire 32-bit register, while leaving the internal bit arrangement of each byte entirely intact. As shown in Figure 4.7, the most significant byte (`rs1[31:24]`) is routed directly to the least significant position (`rd[7:0]`), effectively flipping the byte order.

Algorithm algorithm 4.8 details the exact hardware implementation of these instructions mapped within the execution stage. The data path routing is governed directly by the instruction's encoded bits `[24:20]`. As detailed in table 4.4, this 5-bit segment acts as a hardwired control mask that dictates exactly which stages of the butterfly network are activated.

Table 4.4: Control mask encodings used in the implementation.

Instruction	**Instruction Field** `[24:20]`	**Activated Stages**
`brev8`	`00111`	Lower stages (Bits, pairs, and nibbles)
`rev8`	`11000`	Upper stages (Bytes and half-words)

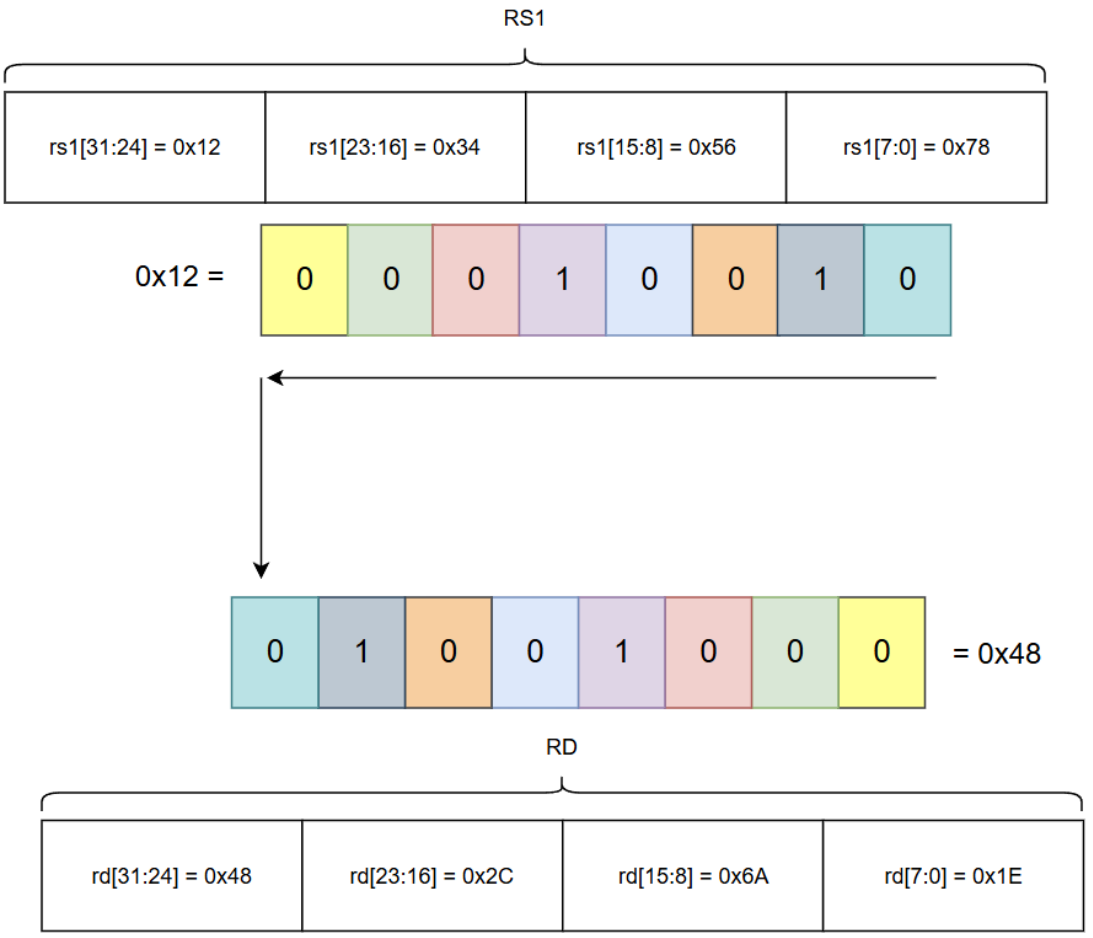


Figure 4.6: Execution of the `brev8` instruction, demonstrating internal bit reversal per byte. Source: Author.

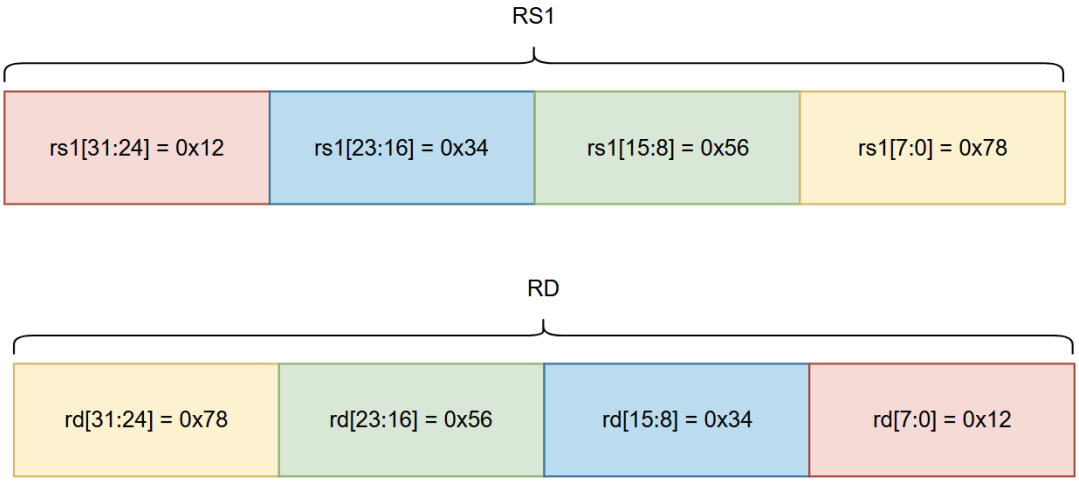


Figure 4.7: Execution of the `rev8` instruction, demonstrating byte-level endianness swapping. Source: Author.

4.2.3.6 ZIP and UNZIP

Concluding the ZBKB extension, the `zip` and `unzip` instructions perform specialized bit-level interleaving and de-interleaving on a single 32-bit register. The `zip` operation

Algorithm 4.8: Hardware Execution of the Generalized Reversal Network for BREV8 and REV8.

Require: Data operand $rs1 \in \mathbb{Z}_{2^{32}}$, and control mask $instruction[24:20] \in [0, 31]$
Ensure: $rev\_result \in \mathbb{Z}_{2^{32}}$
$rev\_result \leftarrow rs1$
▷ Stage 0: Swap adjacent single bits
if $instruction[20]$ **then**
    $rev\_result \leftarrow ((rev\_result \wedge 0x55555555) \ll 1) \vee ((rev\_result \wedge 0xAAAAAAAA) \gg 1)$
end if
▷ Stage 1: Swap adjacent 2-bit pairs
if $instruction[21]$ **then**
    $rev\_result \leftarrow ((rev\_result \wedge 0x33333333) \ll 2) \vee ((rev\_result \wedge 0xCCCCCCCC) \gg 2)$
end if
▷ Stage 2: Swap adjacent 4-bit nibbles
if $instruction[22]$ **then**
    $rev\_result \leftarrow ((rev\_result \wedge 0x0F0F0F0F) \ll 4) \vee ((rev\_result \wedge 0xF0F0F0F0) \gg 4)$
end if
▷ Stage 3: Swap adjacent 8-bit bytes
if $instruction[23]$ **then**
    $rev\_result \leftarrow ((rev\_result \wedge 0x00FF00FF) \ll 8) \vee ((rev\_result \wedge 0xFF00FF00) \gg 8)$
end if
▷ Stage 4: Swap adjacent 16-bit half-words
if $instruction[24]$ **then**
    $rev\_result \leftarrow ((rev\_result \wedge 0x0000FFFF) \ll 16) \vee ((rev\_result \wedge 0xFFFF0000) \gg 16)$
end if
return rev_result

systematically shuffles the source register by routing its lower half (`rs1[15:0]`) into the even bit positions of the destination register (`rd`), and its upper half (`rs1[31:16]`) into the odd bit positions, as illustrated in Figure 4.8. Conversely, the `unzip` instruction reverses this mapping, extracting the even bits of the source to fill the lower half of the result, while packing the odd bits into the upper half, which is visually demonstrated in Figure 4.9. As formalized in Algorithm 4.9, this physical execution requires no sequential logic, synthesizing directly into parallel wire crossings.

4.2.4 SHA-2 and SHA-3 Integration

The integration of the SHA-2 and SHA-3 families serves distinct architectural purposes within the processor design. The SHA-2 extension was implemented to satisfy the baseline cryptographic requirements of the broader project. In contrast, the SHA-3 datapath was developed as a strict prerequisite for the ML-KEM (Kyber) algorithm. Kyber relies entirely on Keccak-based primitives—specifically SHA-3 and the extendable-output functions

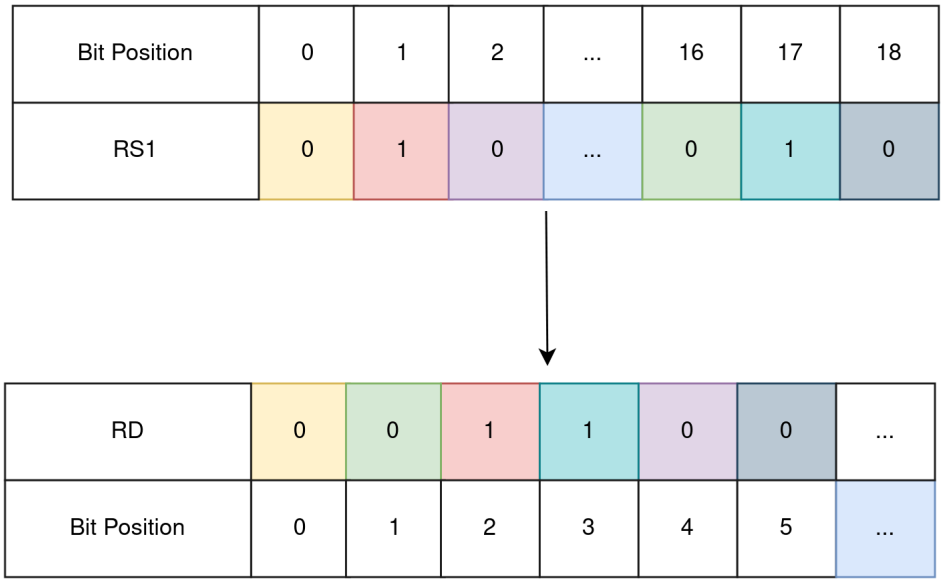


Figure 4.8: Bit-level interleaving execution for the `zip` instruction. Source: Author.

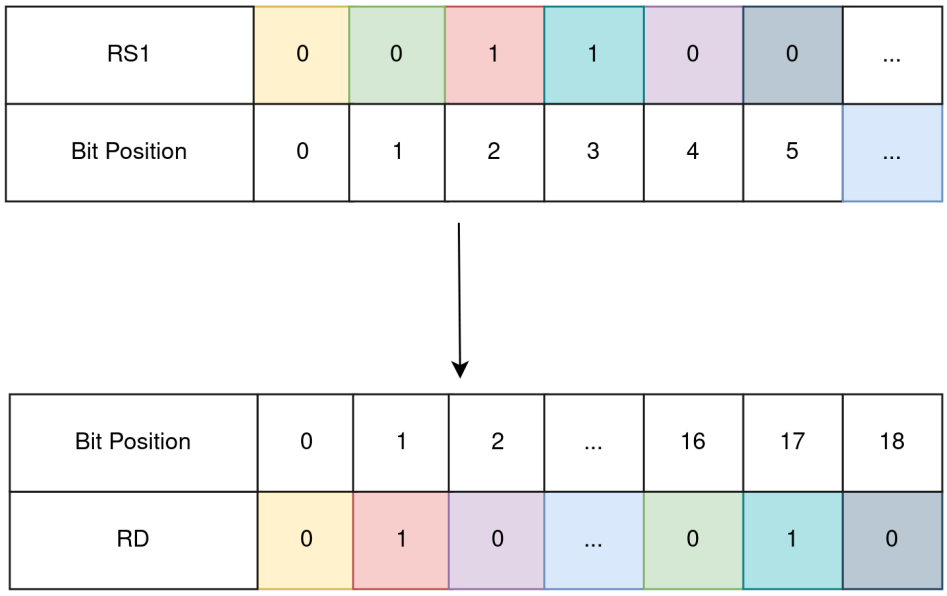


Figure 4.9: Bit-level de-interleaving execution for the `unzip` instruction. Source: Author.

SHAKE128 and SHAKE256—to execute its internal state hashing and matrix expansion. Furthermore, according to the FIPS 203 standard, the random number generator (RNG) remains the only symmetric component open to implementation flexibility, allowing the hardware to interface with any external or integrated entropy source, provided it strictly meets established NIST security requirements.

Algorithm 4.9: Hardware Execution of ZIP and UNZIP Instructions.

Require: Data operand $rs1 \in \mathbb{Z}_{2^{32}}$, and control signal *is_zip*
Ensure: *shuffle_result* $\in \mathbb{Z}_{2^{32}}$
if *is_zip* **then**
▷ Interleave lower and upper halves
  for $i = 0$ **to** 15 **do**
    $shuffle\_result[2i] \leftarrow rs1[i]$
    $shuffle\_result[2i + 1] \leftarrow rs1[i + 16]$
  end for
else
▷ De-interleave (unzip) to lower and upper halves
  for $i = 0$ **to** 15 **do**
    $shuffle\_result[i] \leftarrow rs1[2i]$
    $shuffle\_result[i + 16] \leftarrow rs1[2i + 1]$
  end for
end if
return *shuffle_result*

4.2.5 XKYBER Custom Extension

4.2.5.1 CBD2 and CBD3

While the formal mathematical definition of ML-KEM was introduced previously in Chapter 4, adapting these concepts to the RS5 processor requires optimizations. As defined by the FIPS 203 standard [NIST, 2024a], Algorithm 4.1 relies on an intensive loop to generate a polynomial of 256 coefficients within the range $\{0, \dots, q-1\}$.

Implementing this entire loop directly in hardware as a single multi-cycle instruction would severely degrade system performance, effectively stalling the processor pipeline for at least 256 clock cycles. To circumvent this latency bottleneck, the architectural strategy adopted from [Gewehr et al., 2024] avoids internal hardware loops entirely. Instead, it extracts only the core combinational logic found inside the loop and maps it into a dedicated, single-cycle custom instruction. This isolated operation, responsible for sampling the individual coefficients, is detailed in Algorithm 4.10.

Algorithm 4.10: CBD_η Sampling. [Gewehr et al., 2024].

Require: B is an array of 2η pseudo-random bits
Ensure: Returns a pseudo-random element $\in \{0, 1, \dots, \eta, q-\eta-1, \dots, q-1\}$
$a \leftarrow \sum_{i=0}^{\eta-1} B_i$ ▷ Sum η lower bits of B
$b \leftarrow \sum_{i=\eta}^{2\eta-1} B_i$ ▷ Sum η upper bits of B
return $a - b \pmod q$

The hardware realization of the CBD sampling is highly streamlined, avoiding complex sequential state machines. The execution datapath initiates the calculation through Algorithm 4.11, generating the intermediate `cbd_result_high` and `cbd_result_low` coefficients. Because the subtraction of the random bits can yield negative values, these intermediate results are processed as 13-bit signed integers to preserve the sign bit. To guarantee mathematical correctness, these 13-bit values are immediately routed into the shared modular adjustment block. As formalized in Algorithm 4.12, this final combinational stage evaluates the sign bit and applies the appropriate arithmetic correction, ensuring that every coefficient is correctly bounded within the valid $[0, 3328]$ polynomial ring.

Algorithm 4.11: Hardware Execution of the Single-Cycle CBD Datapath.

Require: Seed operand $rs1 \in \mathbb{B}^{32}$, and control signal *is_cbd3*
Ensure: Sampled coefficients *cbd_result_high* and *cbd_result_low*
if *is_cbd3* **then** ▷ Evaluate logic for $\eta = 3$ (`KYBER_CBD3`)
  $cbd\_result\_high \leftarrow (rs1[6] + rs1[7] + rs1[8]) - (rs1[9] + rs1[10] + rs1[11])$
  $cbd\_result\_low \leftarrow (rs1[0] + rs1[1] + rs1[2]) - (rs1[3] + rs1[4] + rs1[5])$
else ▷ Evaluate logic for $\eta = 2$ (`KYBER_CBD2`)
  $cbd\_result\_high \leftarrow (rs1[4] + rs1[5]) - (rs1[6] + rs1[7])$
  $cbd\_result\_low \leftarrow (rs1[0] + rs1[1]) - (rs1[2] + rs1[3])$
end if
return $\{cbd\_result\_high, cbd\_result\_low\}$

4.2.5.2 ALU Adder Reuse in the Execution Stage

To maximize hardware reuse within the execution stage, the primary ALU adder is shared across both the standard RISC-V instruction set and the new ML-KEM extensions. Rather than instantiating parallel arithmetic logic units, the architecture implements a multiplexing strategy at the inputs of the main adder, as formalized in Algorithm 4.13.

When a Kyber-specific operation is detected, the control logic isolates the main adder from the standard general-purpose register file. Instead, it routes the pre-processed operands (`mul_op_a` and `mul_op_b`) generated by the specialized Kyber datapath logic. For standard operations, the multiplexers default to the base registers (`rs1` and `rs2`), dynamically applying a two's complement inversion exclusively when a baseline subtraction (`SUB`) is decoded.

4.2.5.3 SUM and SUB

From an architectural standpoint, integrating the `kyberadd` and `kybersub` instructions into the RS5 pipeline is straightforward. Both operations function as SIMD (Single Instruction, Multiple Data) instructions, processing two coefficients in parallel. For the addition

Algorithm 4.12: Hardware for Modular Adjustment for KYBER_ADD, KYBER_CBD2, KYBER_CBD3, KYBER_SUB.

Require: Intermediate ALU result *alu_val*[12 : 0], and control signal *operator*
Ensure: Bounded coefficient *result* $\in [0, 3328]$

▷ Step 1: Configure adjustment adder operand (Reuse of hardware)
if *operator* $\in \{$KYBER_ADD, KYBER_MUL$\}$ **then**
  adj_operand[12 : 0] $\leftarrow -3329$
else ▷ KYBER_SUB, KYBER_CBD2, KYBER_CBD3, KYBER_COMPRESS
  adj_operand[12 : 0] $\leftarrow 3329$
end if

▷ Step 2: Compute adjustment beforehand
adj_val $\leftarrow$ *alu_val* + *adj_operand*

▷ Step 3: Multiplexed selection based on instruction
Switch *operator* **do**
  Case KYBER_ADD, KYBER_MUL:
    mux_sel $\leftarrow \neg$*adj_val*[12] ▷ Select if subtraction did not underflow
  Case KYBER_SUB:
    mux_sel $\leftarrow \neg$*alu_val*[12] ▷ Select if initial subtraction underflowed
  Case KYBER_CBD2, KYBER_CBD3:
    mux_sel $\leftarrow$ *alu_val*[*sign_bit*] ▷ Select if sampled value is strictly negative
  Case KYBER_COMPRESS:
    mux_sel $\leftarrow 1$
  Default:
    mux_sel $\leftarrow 0$

▷ Step 4: Route final bounded result
if *mux_sel* **then**
  return *adj_val*[11 : 0]
else
  return *alu_val*[11 : 0]
end if

operation, the hardware simply routes the unmodified values from the source registers directly into the ALU adders. To execute the subtraction without expanding the silicon area on a dedicated subtractor for xkyber operations, the datapath introduces a single multiplexing layer. As detailed in Algorithm 4.14, this layer dynamically converts the packed coefficients of the second operand (rs2) into their two's complement form, enabling the complete reuse of the standard adders during the execution stage.

Crucially, the datapath does not terminate at this initial arithmetic calculation. Because ML-KEM mathematics require all coefficients to remain within the finite field $\mathbb{Z}_q$, the intermediate outputs from these parallel adders are immediately routed into the shared modular adjustment block. As formalized in Algorithm 4.12, this final combinational logic evaluates the underflow or overflow conditions and applies the necessary ± 3329 correction within the exact same clock cycle. This guarantees that the final values written to the destination register (rd) are strictly bounded within the valid $[0, 3328]$ polynomial ring.

Algorithm 4.13: Hardware Execution of the Shared ALU Adder at execute stage.

Require: Base registers $rs1, rs2 \in \mathbb{Z}_{2^{32}}$, Kyber operands *mul_op_a*, *mul_op_b*, and control signal *operator*
Ensure: Addition result $sum\_result \in \mathbb{Z}_{2^{32}}$

▷ Step 1: Route first operand based on instruction family
if *operator* $\in$ {KYBER_ADD, KYBER_SUB, KYBER_CBD2, KYBER_CBD3, KYBER_MUL, KYBER_COMPRESS} **then**
    first_operand $\leftarrow$ *kyber_op_a*
else
    first_operand $\leftarrow$ *rs*1
end if

▷ Step 2: Route and pre-process second operand
Switch *operator* **do**
  Case KYBER_SUB, KYBER_MUL, KYBER_COMPRESS:
    sum2_opB $\leftarrow$ *kyber_op_b*
  Case SUB:
    sum2_opB $\leftarrow -rs2$ ▷ Standard two's complement subtraction
  Default: ▷ Standard additions and base instructions
    sum2_opB $\leftarrow$ *rs2*

▷ Step 3: Execute shared addition
sum_result $\leftarrow$ *first_operand* + *sum2_opB*
return *sum_result*

Algorithm 4.14: Pre-processing of Kyber inputs for ALU.

Require: Source registers $rs1, rs2 \in \mathbb{Z}_{2^{32}}$, multi-cycle operands *mul_op_a*, *mul_op_b*, and control signal *operator*
Ensure: Routed ALU operands *alu_op_a* and *alu_op_b*

▷ Step 1: Compute packed two's complement for vector subtraction
rs2_inv $\leftarrow \neg rs2$
sub_val_high $\leftarrow$ (*rs2_inv*[27 : 16] + 1) ▷ 13-bit arithmetic evaluation
sub_val_low $\leftarrow$ (*rs2_inv*[11 : 0] + 1) ▷ 13-bit arithmetic evaluation
packed_sub_val $\leftarrow$ {000, *sub_val_high*, 000, *sub_val_low*}

▷ Step 2: Multiplexed operand selection
Switch *operator* **do**
  Case KYBER_SUB:
    kyber_op_a $\leftarrow$ *rs*1
    kyber_op_b $\leftarrow$ *packed_sub_val*
  Case KYBER_MUL, KYBER_COMPRESS:
    kyber_op_a $\leftarrow$ *mul_op_a*
    kyber_op_b $\leftarrow$ *mul_op_b*
  Default: ▷ Standard routing for KYBER_ADD, etc.
    kyber_op_a $\leftarrow$ *rs*1
    kyber_op_b $\leftarrow$ *rs2*
return {*kyber_op_a*, *kyber_op_b*}

4.2.5.4 COMPRESS - FIRST VERSION

As introduced in Section 4.1.4, the `kybercompress` instruction requires dividing the polynomial coefficients by the prime modulus $q = 3329$. To circumvent the severe latency and area penalties of instantiating a dedicated hardware divider, the datapath translates this mathematical requirement into a optimized, two-cycle fixed-point sequence.

The first microarchitectural version of this instruction, derived from the implementation presented in [Gewehr et al., 2024], is formalized in Algorithm 4.15. During the initial clock cycle, the uncompressed coefficient x is routed simultaneously into two parallel multiplication paths. The integer scaling factor (5039) is processed through a dedicated combinational logic block. Concurrently, the fractional scaling factor (`16'hB760`) is routed into the processor's shared hardware multiplier.

In the subsequent clock cycle, the architecture resolves the radix alignment. Because the fractional constant was pre-scaled by 2^{16} to enable standard integer arithmetic, the resulting product from the multiplier must be divided by 2^{16} to recover its true magnitude. The datapath achieves this division with zero structural latency and no additional logic gates by simply discarding the lower 16 bits and extracting only the most significant bits (`[31:16]`). This aligned fractional component is then accumulated with the integer product, successfully completing the pseudo-division.

Finally, to conclude the instruction and guarantee that the correctly scaled coefficient is strictly bounded, this accumulated result is routed through the modular adjustment hardware. As detailed in Algorithm 4.12, this terminal step ensures the mathematically correct modulo reduction is applied before the final compressed coefficient is dispatched to the pipeline.

4.2.5.5 COMPRESS - SECOND VERSION

While the baseline architecture successfully achieves a two-cycle latency, it was originally tailored for the simpler Ibex processor pipeline. Integrating this parallel execution into the stricter RS5 core revealed critical timing violations during hardware synthesis in Xilinx Vivado. Specifically, the dedicated combinational logic previously used to multiply the integer constant (5039) created a deep critical path that failed to meet the target clock frequency constraints. This timing violation occurred because the unbuffered logic cascaded directly from the processor's ALU adder, propagated through the combinational shift-and-add structures of the multiplier, and culminated in a complex 33-bit, 4-operand addition, all within a single clock cycle.

To achieve timing closure and further reduce the overall silicon footprint, the datapath was redesigned. The dedicated combinational multiplier was completely removed. Instead, the architecture pivots to a fully serialized, time-multiplexed approach. To break the

Algorithm 4.15: Hardware-Optimized Fixed-Point Division for Kyber Compression.

Require: Uncompressed coefficient $x \in [0, 3328]$
Ensure: Integer approximation of $(x \cdot 2^{24})/3329$
▷ Shared hardware multiplier (Continuous Assignment)
$mac\_result \leftarrow OpA \cdot OpB$ ▷ Partial 16-bit multiplication
▷ Step 1: Load constants
$C_{int} \leftarrow 5039$ ▷ Integer part: $\lfloor 2^{24}/3329 \rfloor$
$C_{frac} \leftarrow$ 16'hB760 ▷ Fractional part: 0.7164×2^{16}
▷ Step 2: Execute parallel multiplications (first cycle)
$sum\_result \leftarrow (x \ll 2) + x$ ▷ ALU computes $5x$
$P_{int} \leftarrow (sum\_result \ll 10) - (x \ll 6) - (x \ll 4) - x$ ▷ Combinational tree for $x \cdot C_{int}$
$OpA \leftarrow x$
$OpB \leftarrow C_{frac}$
$P_{frac} \leftarrow mac\_result$
▷ Step 3: Accumulate partial products (second cycle)
$P_{frac\_higher} \leftarrow P_{frac}[31 : 16]$ ▷ Discard lower 16 bits (divide by 2^{16})
$R_{raw} \leftarrow P_{int} + P_{frac\_higher}$
$result \leftarrow \text{Adjust}(R_{raw})$ ▷ Modular reduction via Algorithm 4.12
return *result*

critical path delay, the shared multiplier module was modified to register its input operands (*OpA* and *OpB*). While registering these inputs successfully resolves the timing violations, it introduces an inherent multi-cycle latency, requiring the sequential computation of both the integer and fractional components.

This serialized execution is dynamically orchestrated by the Finite State Machine (FSM) detailed in Algorithm 4.16. Because the multiplier inputs are now registered, each partial product requires an additional pipeline stage to propagate. The FSM navigates these delays by loading the fractional constant in `STAGE_0`, storing the initial partial products in `STAGE_1` and `STAGE_2`, and selectively routing the upper byte of the operand for the subsequent partial multiplication in `STAGE_2`.

Once the sequential hardware accumulation concludes in `STAGE_3`, the datapath merges the aligned products in `STAGE_4`, successfully resolving the pseudo-division. Finally, identical to the parallel implementation, this raw accumulated result must be routed through the modular adjustment hardware. As detailed in Algorithm 4.12, this terminal step ensures the mathematically correct modulo reduction is applied, strictly bounding the coefficient to the $[0, 3328]$ range before asserting the valid signal.

4.2.5.6 MUL - FIRST VERSION

As introduced in Section 4.1.7, the `kyber_mul` instruction evaluates the product of two polynomial coefficients and must subsequently reduce this result modulo the prime $q = 3329$.

Algorithm 4.16: Finite State Machine (FSM) for Serialized KYBERCOMPRESS Execution.

Input: Operands $x \in [0, 3328]$, Current State S, Control Flag *is_compress*
Output: Bounded compressed integer *Result*, Valid Flag
▷ Shared hardware multiplier (Continuous Assignment)
$mac\_result \leftarrow OpA \cdot OpB$ ▷ Baseline 16-bit native multiplier
▷ Default parallel assignments (Cycle 0)
$OpA \leftarrow x[15:0]$ ▷ Default to lower 16 bits
$OpB \leftarrow 5039$ ▷ Default to integer constant
$Valid \leftarrow 0$
if S = `STAGE_0` **then**
    $OpB \leftarrow$ 16'hB760 ▷ Load fractional constant
    $S_{next} \leftarrow$ `STAGE_1`
else if S = `STAGE_1` **then**
    $temp_0 \leftarrow mac\_result$ ▷ Store fractional product
    $S_{next} \leftarrow$ `STAGE_2`
else if S = `STAGE_2` **then**
    $temp_1 \leftarrow mac\_result$ ▷ Store lower integer partial product
    $OpA \leftarrow x[23:16]$ ▷ Route upper byte for partial product
    $S_{next} \leftarrow$ `STAGE_3`
else if S = `STAGE_3` **then**
    $temp_1 \leftarrow (mac\_result[20:0] \ll 16) + temp_1$ ▷ Hardware accumulation
    $S_{next} \leftarrow$ `STAGE_4`
else if S = `STAGE_4` **then**
    $R_{raw} \leftarrow temp_0[31:16] + temp_1[31:0]$ ▷ Align fractional and add integer
    $Result \leftarrow \text{Adjust}(R_{raw})$ ▷ Modular reduction via Algorithm 4.12
    $Valid \leftarrow 1$ ▷ Signal completion to pipeline
    $S_{next} \leftarrow$ `IDLE`
end if
return *Result*

The first version of this instruction, derived from the implementation presented in [Gewehr et al., 2024], is designed to reuse the baseline 16-bit multiplier native to the processor pipeline. This time-multiplexed execution is formalized in Algorithm 4.17. During the initial clock cycle, the existing hardware computes the standard multiplication ($A \cdot B$).

In the subsequent cycle, the architecture must scale this intermediate 24-bit product by the constant 5039 to estimate the Barrett quotient. Because the 24-bit operand inherently exceeds the capacity of the standard 16-bit multiplier, this specific step cannot simply reuse the partial multiplier block. Instead of instantiating an expensive wide-operand multiplier, the datapath resolves this calculation combinationally through a dedicated, single-cycle shift-add tree.

Following the quotient estimation, the 16-bit hardware multiplier is once again available. In the third cycle, the FSM dynamically routes the newly estimated quotient and the prime modulus (3329) back into the multiplier. Finally, a sequential subtraction yields the raw remainder, which is immediately routed through the modular adjustment logic. As previously

formalized in Algorithm 4.12, this terminal phase strictly bounds the coefficient within the $\mathbb{Z}_q$ ring before the result is committed to the register file.

Algorithm 4.17: Finite State Machine for KYBER_MUL Execution.

Input: Operands $A, B \in [0, 3328]$, Current State S
Output: Reduced product $R \equiv (A \cdot B) \pmod{3329}$
▷ Shared hardware multiplier (Continuous Assignment)
$mac\_result \leftarrow OpA \cdot OpB$ ▷ Baseline 16-bit native multiplier
if $S =$ `STAGE_0` **then**
    $OpA \leftarrow A$
    $OpB \leftarrow B$
    $temp_0 \leftarrow mac\_result$ ▷ Store product $(A \cdot B)$
    $S_{next} \leftarrow$ `STAGE_1`
else if $S =$ `STAGE_1` **then**
    $alu\_base \leftarrow (temp_0 \ll 2) + temp_0$ ▷ ALU computes $5 \times temp_0$
    $P_{scaled} \leftarrow (alu\_base \ll 10) - (temp_0 \ll 6) - (temp_0 \ll 4) - temp_0$ ▷ Scale by 5039
    $temp_1 \leftarrow P_{scaled} \gg 24$ ▷ Estimate quotient
    $S_{next} \leftarrow$ `STAGE_2`
else if $S =$ `STAGE_2` **then**
    $OpA \leftarrow temp_1$ ▷ Route quotient to multiplier
    $OpB \leftarrow 3329$ ▷ Route modulus to multiplier
    $temp_1 \leftarrow mac\_result$ ▷ Overwrite with $(quotient \cdot 3329)$
    $S_{next} \leftarrow$ `STAGE_3`
else if $S =$ `STAGE_3` **then**
    $R_{raw} \leftarrow temp_0 - temp_1$ ▷ Subtract scaled modulus from original product
    $R \leftarrow \text{Adjust}(R_{raw})$ ▷ Modular reduction via Algorithm 4.12
    $S_{next} \leftarrow$ `IDLE`
end if
return R

4.2.5.7 MUL - SECOND VERSION

While the baseline architecture minimized cycle count, integrating its deep combinational logic into stricter processor pipelines exposed potential timing and area bottlenecks in Vivado synthesis. Prior to finalizing a fully serialized architecture, an intermediate exploration was evaluated to assess potential logic optimizations. This experimental iteration, formalized in Algorithm 4.18, sought to break down the large combinational scaling tree for the constant 5039 into discrete, cycle-by-cycle shifts.

The core engineering hypothesis behind this version was that isolating the shift-and-accumulate operations across sequential FSM states would compel the synthesis tool to infer resource sharing. Theoretically, the Genus tool could recognize the mathematical overlap—such as reusing the hardware of a 4-bit and 6-bit shift to synthesize the subsequent 10-bit and 12-bit shifts—thus reducing the overall logic footprint.

However, hardware synthesis results directly contradicted this theoretical assumption. Forcing the shift operations across sequential clock cycles inherently placed them behind register boundaries. This prevented the synthesis tool from performing holistic boolean optimizations. Instead of sharing shift networks, the Genus tool was forced to instantiate multiplexers to route the variously shifted operands into the shared ALU during each distinct state.

Consequently, this implementation suffered a area penalty. The post-synthesis reports revealed that the hardware footprint inflated drastically, surging from the 841 logic gates required by the baseline multiplier to 1737 logic gates. Because this experimental approach effectively doubled the area overhead while introducing multi-cycle latency, it was discarded in favor of more hardware-friendly execution strategies.

Algorithm 4.18: Sequential Execution of the Shift-Add Tree for Quotient Estimation.

Input: Base product $temp_0 \in [0, 2^{24} - 1]$, Current State S
Output: Accumulated scaled product $temp_1 = temp_0 \cdot 5039$
▷ Sequential Shift-Add accumulation reusing the 32-bit ALU
if S = `L1_STAGE0` **then**
    $temp_1 \leftarrow -temp_0$ ▷ Subtract base value ($temp_0 \ll 0$)
    $S_{next} \leftarrow$ `L1_STAGE1`
else if S = `L1_STAGE1` **then**
    $temp_1 \leftarrow temp_1 - (temp_0 \ll 4)$ ▷ Subtract 16x
    $S_{next} \leftarrow$ `L1_STAGE2`
else if S = `L1_STAGE2` **then**
    $temp_1 \leftarrow temp_1 - (temp_0 \ll 6)$ ▷ Subtract 64x
    $S_{next} \leftarrow$ `L1_STAGE3`
else if S = `L1_STAGE3` **then**
    $temp_1 \leftarrow temp_1 + (temp_0 \ll 10)$ ▷ Add 1024x
    $S_{next} \leftarrow$ `L1_STAGE4`
else if S = `L1_STAGE4` **then**
    $temp_1 \leftarrow temp_1 + (temp_0 \ll 12)$ ▷ Add 4096x
    $S_{next} \leftarrow$ `KYBER_BARRETT_L2`
end if
return $temp_1$ ▷ Resulting value inherently equals $temp_0 \cdot 5039$

4.2.5.8 MUL - THIRD VERSION

Because the experimental shift-add tree doubled the hardware footprint without resolving timing constraints, the architecture was fundamentally redesigned into a fully serialized execution. As formalized in Algorithm 4.19, this final architecture strictly enforces a one-cycle pipeline delay between operand dispatch and result capture. Because the multiplier inputs are registered to meet the timing constraints of the RS5 core on Vivado, any data routed to the operands (*OpA* and *OpB*) during a given state is only evaluated and available at the `mac_result` output in the subsequent clock cycle.

Consequently, the dedicated combinational scaling tree previously used for the 5039 multiplication is completely replaced by a sequential accumulation utilizing the baseline 16-bit hardware. While this serialization inherently introduces additional clock cycles to the instruction latency, the architectural trade-off is highly favorable. In practical deployment scenarios, Kyber cryptographic operations are typically invoked intermittently for session key encapsulation—often separated by intervals of several seconds. Therefore, this micro-level performance degradation is practically negligible relative to the macro-level active runtime of the system, justifying the substantial gains achieved in timing closure and area reduction.

Algorithm 4.19: Finite State Machine for Serialized Time-Multiplexed KYBER_MUL.

Input: Operands $A, B \in [0, 3328]$, Current State S
Output: Reduced product $R \equiv (A \cdot B) \pmod{3329}$
▷ Shared hardware multiplier (Continuous Assignment)
$mac\_result \leftarrow OpA \cdot OpB$ ▷ Baseline 16-bit native multiplier
if S = `STAGE_0` **then**
    $OpA \leftarrow A$
    $OpB \leftarrow B$
    $S_{next} \leftarrow$ `STAGE_1`
else if S = `STAGE_1` **then**
    $temp_0 \leftarrow mac\_result$ ▷ Store base product $(A \cdot B)$
    $OpA \leftarrow temp_0[15:0]$ ▷ Route lower half of product
    $OpB \leftarrow 5039$ ▷ Route integer constant
    $S_{next} \leftarrow$ `STAGE_2`
else if S = `STAGE_2` **then**
    $temp_1 \leftarrow mac\_result$ ▷ Store lower partial product
    $OpA \leftarrow temp_0[31:16]$ ▷ Route upper half of product
    $S_{next} \leftarrow$ `STAGE_3`
else if S = `STAGE_3` **then**
    $accum \leftarrow (mac\_result \ll 16) + temp_1$ ▷ Hardware accumulation
    $temp_1 \leftarrow accum$ ▷ Overwrite with full scaled product
    $S_{next} \leftarrow$ `STAGE_4`
else if S = `STAGE_4` **then**
    $OpA \leftarrow temp_1 \gg 24$ ▷ Route estimated quotient
    $OpB \leftarrow 3329$ ▷ Route modulus
    $S_{next} \leftarrow$ `STAGE_5`
else if S = `STAGE_5` **then**
    $temp_1 \leftarrow mac\_result$ ▷ Store $(quotient \cdot 3329)$
    $S_{next} \leftarrow$ `STAGE_6`
else if S = `STAGE_6` **then**
    $R_{raw} \leftarrow temp_0 - temp_1$ ▷ Sequential subtraction (ALU)
    $R \leftarrow \text{Adjust}(R_{raw})$ ▷ Modular reduction via Algorithm 4.12
    $S_{next} \leftarrow$ `IDLE`
end if
return R

4.3 Functional Verification and Evaluation

To guarantee the functional correctness of the integrated cryptographic accelerators, the verification methodology strictly relies on official Known Answer Tests (KATs) provided by the National Institute of Standards and Technology (NIST). Rather than developing a custom validation suite from scratch, this work leverages the software-based verification previously established by [Gewehr et al., 2024].

Implemented as a C application executing on the modified processor, this validation software is designed to automatically stimulate the custom hardware instructions and evaluate the processor's outputs against the official NIST cryptographic reference vectors. Specifically, it verifies the exact functional compliance of the underlying hash functions according to the SHA-2 (FIPS 180) and SHA-3 (FIPS 202) standards, alongside the full Module-Lattice-Based Key-Encapsulation Mechanism (ML-KEM/Kyber) algorithm. By successfully matching the hardware execution results with these standardized KATs, the functional integrity and mathematical accuracy of the entire custom datapath are validated.

4.4 Post-Synthesis and FPGA Analysis

Initial post-synthesis evaluations utilizing Cadence Genus confirmed that the custom RS5 successfully met all timing constraints, achieving timing closure at a target frequency of 250 MHz defined by the WIMED project for the standard-cell ASIC implementation.

However, mapping the design to an FPGA prototyping environment revealed significant physical implementation challenges. At a target operating frequency of 100 MHz, the underlying FPGA routing architecture introduced propagation delays, resulting in timing violations across the cryptographic datapaths. To resolve these critical path delays without altering the core functional logic, the hardware boundaries of the custom modules were modified to register all incoming operands and outgoing results.

While this architectural isolation brought the worst-case negative slack (WNS) to near zero, strict timing closure was finalized by applying synthesis and routing directives within the FPGA toolchain. Consequently, this physical strategy introduces a latency trade-off: the overall execution time of the cryptographic instructions is increased by two clock cycles due to the input and output staging registers.

5. RESULTS

This chapter presents the synthesis evaluation of the proposed post-quantum cryptographic accelerator integrated into the RS5 RISC-V core. To ensure a standardized comparative analysis across all evaluation metrics, a baseline operating frequency of 100 MHz was enforced as the primary timing constraint for Field-Programmable Gate Array (FPGA) prototyping, and 250 MHz for Application-Specific Integrated Circuit (ASIC) synthesis. Furthermore, all ASIC evaluations were conducted using the TSMC 28nm standard-cell library, provided as part of the WiMED project.

All evaluations considered the following scenarios:

- **Baseline RS5:** The unmodified processor core containing only the native scalar multiplier, evaluated prior to the integration of the post-quantum Kyber accelerators and the scalar cryptographic extensions (Zbkb and Zknh).
- **Kyber V1:** The initial integration featuring the combinational cryptographic multiplier (detailed in Section 4.2.5.6) and the baseline compression module (Section 4.2.5.4).
- **Kyber V2:** The resource-sharing experiment that attempted to multiplex the native ALU structure for cryptographic multiplication by 5039 (Section 4.2.5.7) alongside the baseline compression module (Section 4.2.5.4).
- **Kyber V3:** The serialized architecture utilizing the FSM-driven sequential multiplier (Section 4.2.5.8) and the fully registered compression module (Section 4.2.5.5). For the 28nm ASIC implementation, this architecture represents the optimal design point, successfully achieving timing closure with a minimized area footprint.
- **Kyber V4:** Optimization required for the FPGA prototyping phase. It utilizes the same sequential arithmetic modules as V3 but enforces explicit register boundaries on all input and output ports. While this insertion incurs a hardware area penalty, it is mandatory to prevent deep combinational paths from leaking into the slower FPGA routing fabric, a necessary overhead to secure timing closure at 100 MHz on the Artix-7 device.
- **Double Mul Experiment:** An experimental configuration representing a direct combination of the Baseline RS5 and Kyber V1 architectures. In this setup, both the native RS5 multiplier and the purely combinational cryptographic multiplier (Section 4.2.5.6) were physically instantiated to operate in parallel.

The subsequent sections detail the progressive resolution of timing, routing, and power bottlenecks as the architecture evolved from the unoptimized combinational baseline (V1) to the serial implementation (V4).

5.1 FPGA Prototyping using Xilinx Vivado

Before the silicon ASIC flow, physical prototyping on an FPGA is essential to validate the design's functional correctness and physical feasibility. The following subsections detail the implementation results generated by Xilinx Vivado targeting an Artix-7 (xc7a100t) FPGA.

This preliminary physical evaluation encompasses critical-path logic depth, routing success via Total Negative Slack (TNS) analysis, combinational versus sequential resource utilization (LUTs and Flip-Flops), and dynamic power dissipation within the configurable logic fabric. Analyzing these metrics provides a critical baseline, demonstrating how the transition from a combinational datapath to a fully serialized state machine resolves immediate hardware bottlenecks within the reconfigurable architecture.

5.1.1 Physical Implementation and Critical Path Analysis

Figure 5.1 and Table 5.1 illustrate the direct correlation between the critical path logic depth and the Worst Negative Slack (WNS). The baseline RS5 processor met timing constraints with a maximum logic depth of 11 levels. However, the combinational integration (Kyber V1) and pseudo-sequential multiplexing (Kyber V2) extended the logic depth to 20 and 27 levels, respectively, resulting in timing violations.

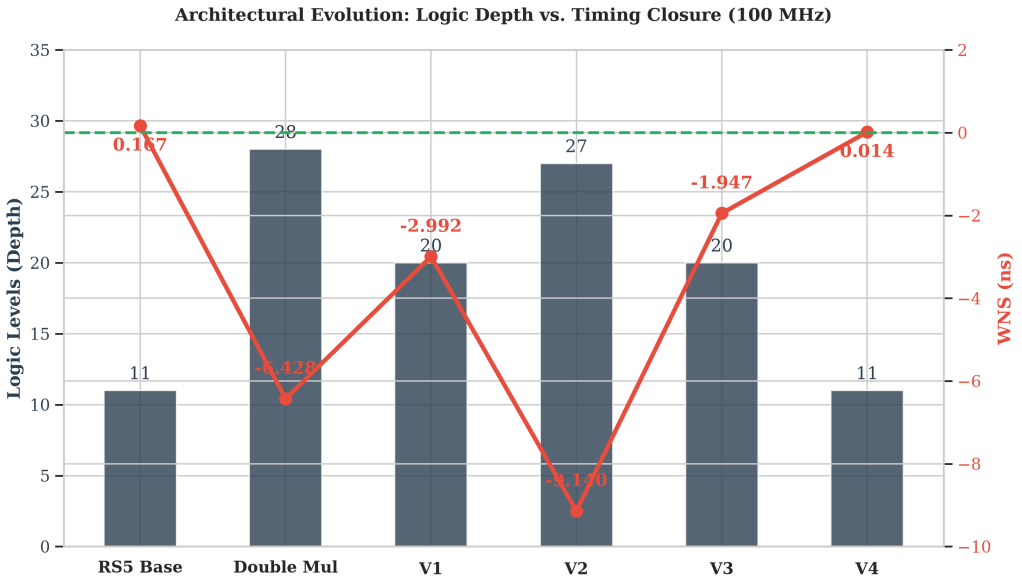


Figure 5.1: Evolution of critical path logic depth compared to Worst Negative Slack (WNS) across architectures.

Transitioning to the sequential Kyber V3 architecture reduced the slack, but the design still failed to meet timing, exhibiting 20 logic levels and a WNS of -1.947 ns. Finally,

Table 5.1: Evolution of Critical Path Logic Depth and Timing Violations (100 MHz).

Architecture Revision	Logic Levels (Depth)	Worst Negative Slack (WNS)
Baseline RS5 (No Kyber)	11	+0.167 ns
Double Mul Experiment	28	-6.428 ns
Kyber V1	20	-2.992 ns
Kyber V2	27	-9.140 ns
Kyber V3	20	-1.947 ns
Kyber V4	**11**	**+0.014 ns**

the insertion of explicit register boundaries in the Kyber V4 architecture effectively broke the remaining combinational chains. This optimization restored the logic depth to its native 11 levels, achieving timing closure with a positive WNS of +0.014 ns under the 100 MHz constraint.

5.1.2 Routing and Total Negative Slack

While the WNS identifies the single worst-failing path, evaluating the Total Negative Slack (TNS) and the number of failing endpoints exposes global routing degradation across the FPGA. As demonstrated in Table 5.2 and Figure 5.2, the multiplexing overhead in the Kyber V2 architecture caused a global timing failure, accumulating over 10,000 ns of negative slack and desynchronizing nearly half of the core's endpoints. This indicates that the proposed logic actively bottlenecks the system due to excessive combinational multiplexing.

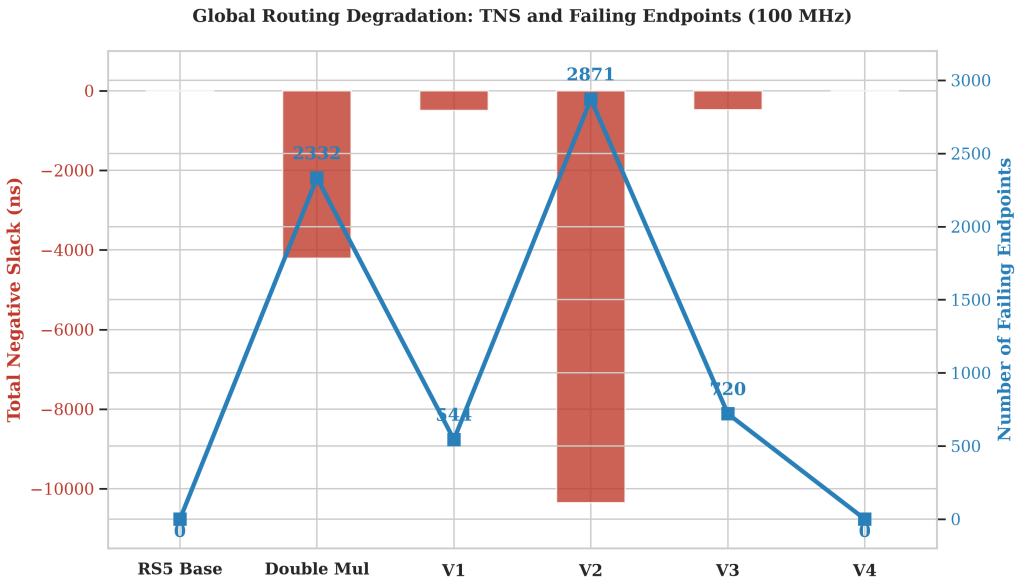


Figure 5.2: Global routing degradation, highlighting Total Negative Slack (TNS) and the number of failing endpoints.

Table 5.2: Global Timing Violations and Slack Analysis Across Revisions (100 MHz).

Architecture Revision	Worst Negative Slack (WNS)	Total Negative Slack (TNS)	Failing Endpoints
Baseline RS5 (No Kyber)	+0.167 ns	0.000 ns	0 / 6115
Double Mul Experiment	-6.428 ns	-4206.716 ns	2332 / 6534
Kyber V1	-2.992 ns	-497.504 ns	544 / 6086
Kyber V2	-9.140 ns	-10356.031 ns	2871 / 6295
Kyber V3	-1.947 ns	-474.557 ns	720 / 6340
Kyber V4	**+0.014 ns**	**0.000 ns**	**0 / 6481**

Although the serialized Kyber V3 architecture reduced this routing congestion, it was the enforcement of register boundaries in Kyber V4, combined with optimization directives during synthesis, that allowed the EDA tool to resolve placement and routing bottlenecks. This final structural isolation restored global stability, dropping the TNS to zero and ensuring that no endpoints failed across the entire processor.

5.1.3 FPGA Area

Table 5.3 and Figure 5.3 present resource utilization data obtained from hierarchical synthesis reports. The parallelization strategy explored in the Double Mul experiment increased the combinational footprint, raising the multiplier implementation to 826 LUTs and the total core utilization to 6443 LUTs. Likewise, the multiplexed V2 architecture incurred a combinational area penalty, increasing the multiplier utilization to 773 LUTs. The final V4 architecture achieves a more balanced area and performance trade-off. By extending ALU multiplexing across register boundaries, the multiplier's combinational footprint was reduced to 453 LUTs. This reduction came at the cost of a larger sequential footprint of 319 flip-flops

Table 5.3: Hardware Resource Utilization Across the designs (100 MHz).

Architecture Revision	Multiplier LUTs	Multiplier FFs	Core Total LUTs	Core Total FFs
Baseline RS5 (No Kyber)	72	90	4214	2486
Double Mul Experiment	826	277	6443	2376
Kyber V1	270	66	5454	2129
Kyber V2	773	173	6207	2241
Kyber V3	397	170	5835	2263
Kyber V4	**453**	**319**	**5556**	**2405**

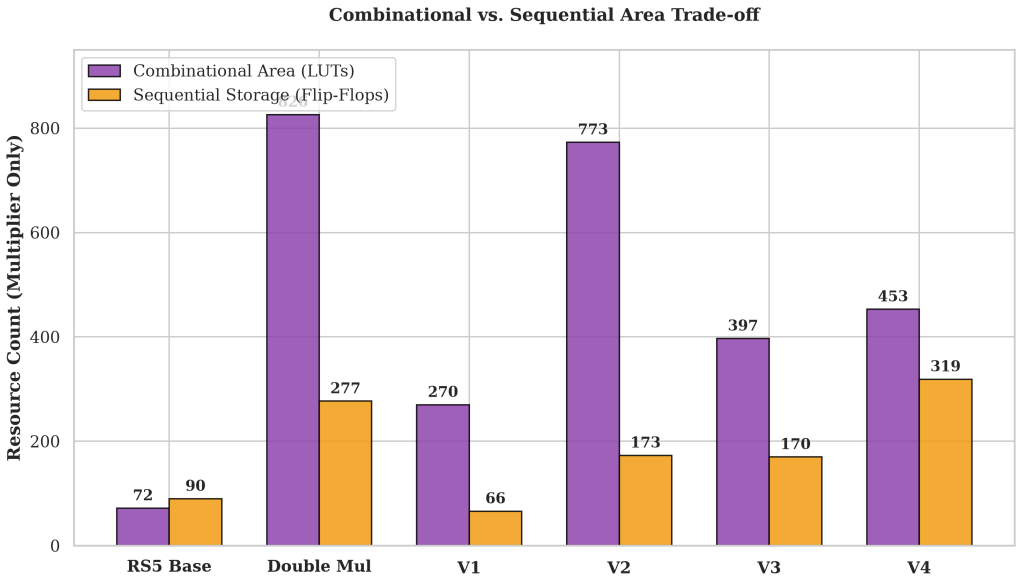


Figure 5.3: Combinational versus sequential logic utilization focusing exclusively on the multiplier module.

5.1.4 Dynamic Power Dissipation and Efficiency

Power dissipation on the FPGA platform is inherently composed of two components: the static baseline power of the silicon fabric and the dynamic power associated with switching activity in the synthesized logic. Because the static device power remains approximately constant at 98.0 mW across all implementations, architectural efficiency is assessed primarily through dynamic power consumption, as summarized in Table 5.4 and Figure 5.4. The unoptimized parallelization explored in the Double Mul experiment incurred a dynamic power penalty, increasing the total core dynamic power to 90.0 mW. In contrast, the sequential Version 3 architecture reduced glitching and combinational transitions by enforcing strict register boundaries. As a result, the standalone multiplier exhibited a dynamic power consumption of only 5.0 mW, while the complete RS5 core maintained a comparatively low dynamic power footprint of 62.0 mW.

Table 5.4: Dynamic Power Dissipation Across Revisions (xc7a100t FPGA) – 100 MHz.

Architecture Revision	Multiplier Power (mW)	Core Total Power (mW)	Total Chip Power (mW)
Baseline RS5 (No Kyber)	2.0	65.0	176.0
Double Mul Experiment	11.0	90.0	202.0
Kyber V1	4.0	83.0	193.0
Kyber V2	7.0	78.0	189.0
Kyber V3	5.0	65.0	177.0
Kyber V4	**5.0**	**62.0**	**173.0**

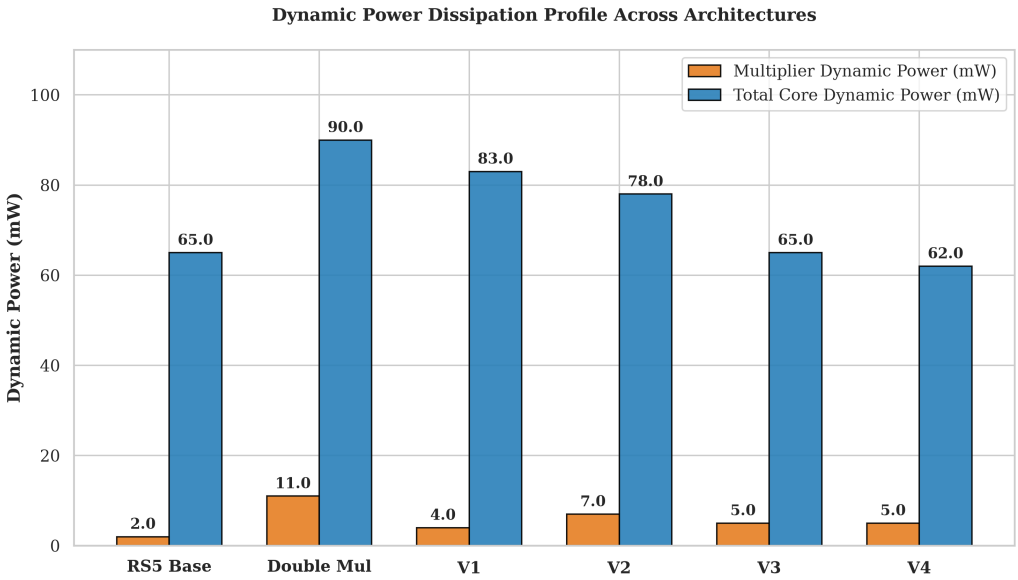


Figure 5.4: Dynamic power dissipation profile detailing the isolated multiplier and total core overhead.

5.2 ASIC Synthesis using Cadence Genus

While FPGA prototyping provides rapid architectural validation, evaluating the physical scalability, energy efficiency, and operational frequency limits of the post-quantum accelerator requires a dedicated Application-Specific Integrated Circuit (ASIC) synthesis flow.

The following subsections detail the implementation results generated by Cadence Genus. The hardware architectures were synthesized using a TSMC 28nm standard-cell library, with a 250 MHz frequency constraint across all synthesis runs.

It must be noted that the Kyber V4 architecture is excluded from the ASIC evaluations presented in this section. As established before, V4 represents a structural adaptation of the V3 architecture, incorporating register boundaries on all input and output ports to achieve 100 MHz timing closure on the FPGA routing fabric. Because this perimeter registration is an FPGA-specific optimization and provides no architectural benefit to the standard-cell design, where V3 already meets all constraints at the 250 MHz target, V4 is categorized as a prototyping artifact and is therefore omitted from the upcoming synthesis tables and graphs.

5.2.1 Area Evaluation

Table 5.5 and Figure 5.5 illustrate area results. While the baseline RS5 core occupies 21,807.20 μm^2, the final V3 architecture increases the total core area to 23,079.52 μm^2. This overhead is due to the isolated multiplier, whose area reached 2,103.15 μm^2.

The observed area overhead highlights the trade-off required to achieve timing closure at 250 MHz. The insertion of intermediate pipeline registers increases silicon utilization, but this cost is a necessary compromise to sustain the operating frequencies.

Table 5.5: ASIC Synthesis Area Estimation (Cadence Genus, 28nm@250 MHz).

Architecture Revision	**Multiplier Cells**	**Multiplier Area (μm^2)**	**Core Total Cells**	**Core Total Area (μm^2)**
Baseline RS5 (No Kyber)	740	1,301.08	13,845	21,807.20
Double Mul Experiment	2108	3,064.21	17,615	24,304.57
Kyber V1	1351	1,832.73	16,599	23,049.59
Kyber V2	1811	2,521.46	16,274	24,190.58
Kyber V3	1502	2,103.15	16,866	23,079.52

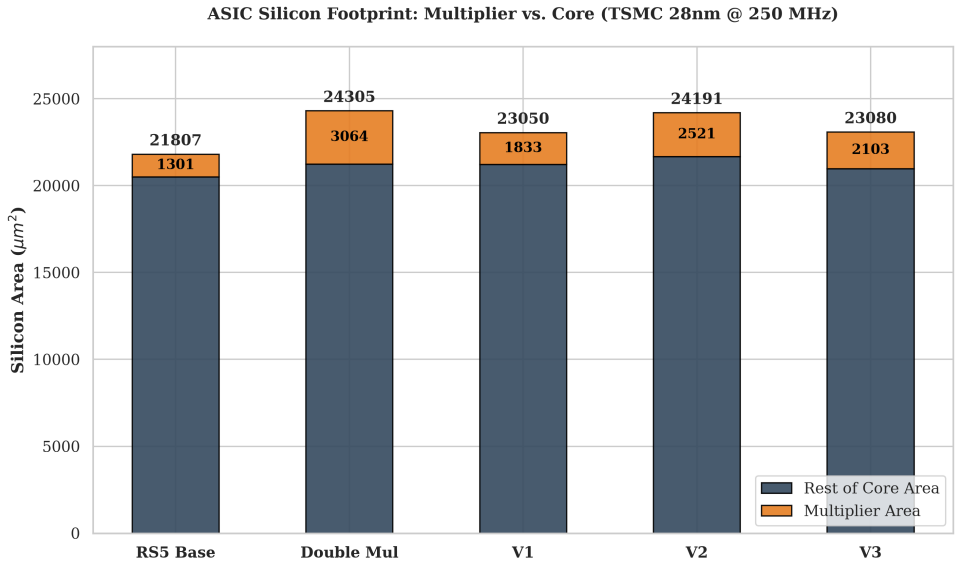


Figure 5.5: ASIC silicon footprint evolution comparing the isolated multiplier against the total core area in a 28nm technology node.

To quantify design complexity independently of vendor-specific cell dimensions, the physical area was normalized to NAND2 Gate Equivalents (kGE), as presented in Table 5.6 and Figure 5.6.

The V2 architecture required the synthesis tool to implement an extensive multiplexing network for signal routing, increasing the overall complexity to 38.70 kGE. In contrast, the final V3 architecture achieved a more compact implementation of 37.30 kGE through datapath separation, reducing the routing and control overhead associated with resource sharing. These structural refinements enabled effective datapath isolation and alleviated the routing congestion observed in earlier iterations, meeting the timing constraints.

Table 5.6: ASIC Gate Equivalent (GE) Complexity Analysis (TSMC 28nm@250MHz)

Architecture Revision	**Total Cell Instances**	**Total Area (μm^2)**	**Complexity (kGE)***
Baseline RS5 (No Kyber)	13,845	15,225.71	34.60 kGE
Double Mul Experiment	17,615	17,223.19	39.14 kGE
Kyber V1	16,599	16,322.67	37.10 kGE
Kyber V2	16,274	17,026.12	38.70 kGE
Kyber V3	16,866	16,410.49	37.30 kGE

**NAND2 Gate Equivalent (GE) normalized using standard 0.44μm^2 cell area.*

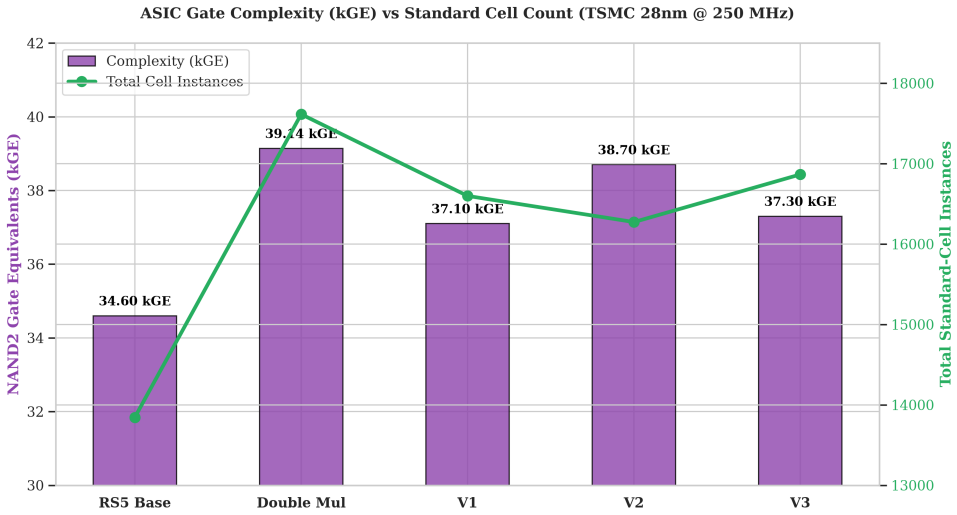


Figure 5.6: Standard-cell instance count versus normalized NAND2 gate equivalent (kGE) complexity.

5.2.2 Timing Evaluation

Table 5.7 and Figure 5.7 summarize the **sequential** profile and gating coverage across all architectural versions. The baseline RS5 core and the Kyber revisions (V1 through V3) demonstrated near-optimal sequential efficiency, maintaining a gating coverage well above 98%. These results indicate that the structural modifications required to serialize the datapath and close timing were implemented while maintaining clock gating. As a result, unnecessary switching activity was minimized, helping to constrain the associated increase in dynamic power consumption.

Table 5.8 and Figure 5.8 report the **critical path delay** and the remaining **setup slack** (timing margin) for each design iteration under the 4.0 ns period constraint. The multiplexing overhead introduced by the V2 architecture increased routing complexity, pushing the critical path delay to 3.998 ns. As a result, V2 used the entire timing budget, satisfying the constraint with a negligible setup slack of +1.8 ps. The subsequent architectural revisions mitigated this bottleneck by enforcing structural isolation and separating the datapaths. The

Table 5.7: ASIC Clock Gating Efficiency and Sequential Storage Analysis (250 MHz).

Architecture Revision	Total Flip-Flops	Gated FFs	Ungated FFs	Gating Coverage
Baseline RS5 (No Kyber)	3108	3085	23	99.26%
Double Mul Experiment	3179	3156	23	99.28%
Kyber V1	3088	3065	23	99.26%
Kyber V2	3195	3154	41	98.72%
Kyber V3	3190	3150	40	98.75%

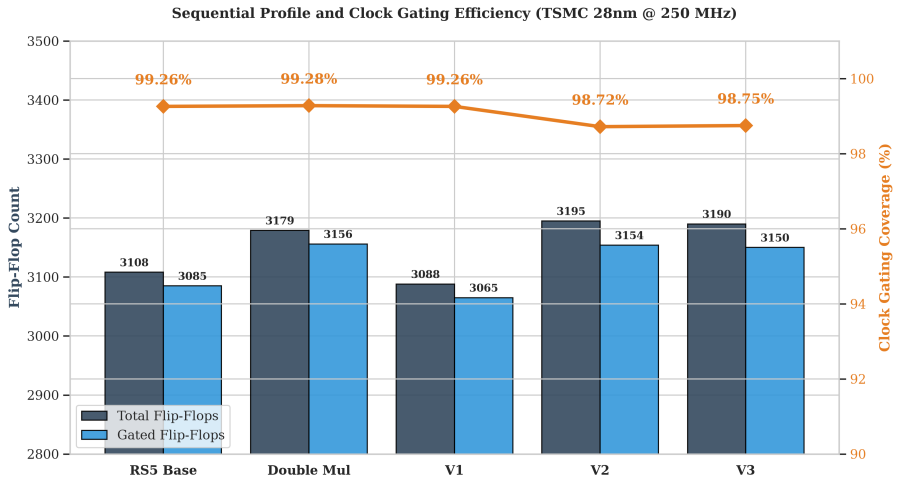


Figure 5.7: Sequential profile and clock gating efficiency scaling across revisions.

V3 architecture reduced routing congestion, reducing the critical path delay to 3.586 ns. This optimization granted the synthesis tool a +414.0 ps timing margin at the slowest and highest-temperature silicon corner (0.81 V and 125℃).

Table 5.8: Critical Path and Setup Margin (Worst-Case 0.81V, 125℃ @ 250 MHz).

Architecture Revision	Critical Path Delay (ns)	Setup Slack Margin (ps)
Baseline RS5 (No Kyber)	3.718	+282.0
Double Mul Experiment	3.761	+238.9
Kyber V1	3.809	+190.6
Kyber V2	3.998	+1.8
Kyber V3	3.586	+414.0

The **start** and end **points** of the worst-case setup paths at the slow corner, defined at 0.81 V and 125℃, provide additional insight into the internal routing behavior of the processor revisions. Table 5.9 and Figure 5.9 report the critical path delays observed across the evaluated architectures. In the baseline processor, the critical path lies in the control-routing logic between the Decoder and Fetch stages, with a signal propagation delay of 3684 ps. In the V2 architecture, the multiplexing logic increased routing complexity and extended this path to 3977 ps, while also shifting the critical endpoint into the Decoder. This result indi-

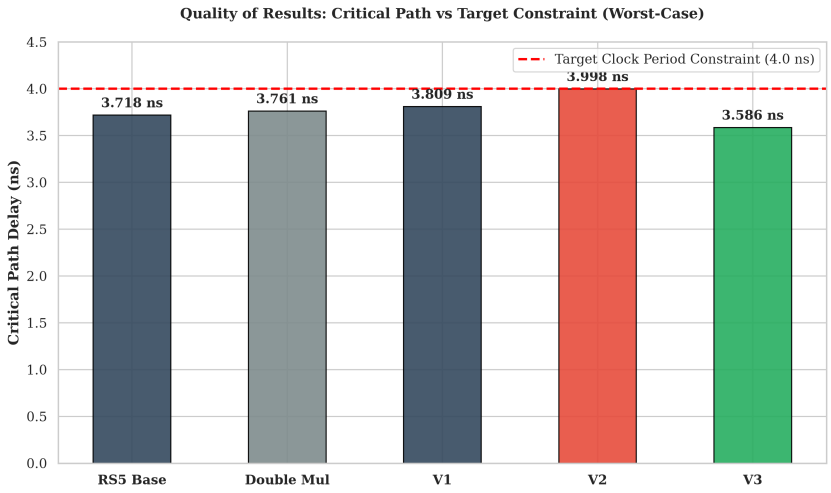


Figure 5.8: Evolution of the critical path delay approaching the 4.0 ns target constraint limit under worst-case conditions.

cates that the added multiplexing structures introduced local congestion and increased the timing pressure on the control path.

Table 5.9: Detailed Setup Timing Analysis and Critical Path Tracing (Worst-Case 0.81V, 125℃ @ 250 MHz)

Architecture	Critical Startpoint	Critical Endpoint	Data Path Delay (ps)
Baseline RS5	DECODER	FETCH	3684
Double Mul Exp.	DECODER	FETCH	3728
Kyber V1	DECODER	FETCH	3776
Kyber V2	DECODER	DECODER	3977
Kyber V3	DECODER	FETCH	3552

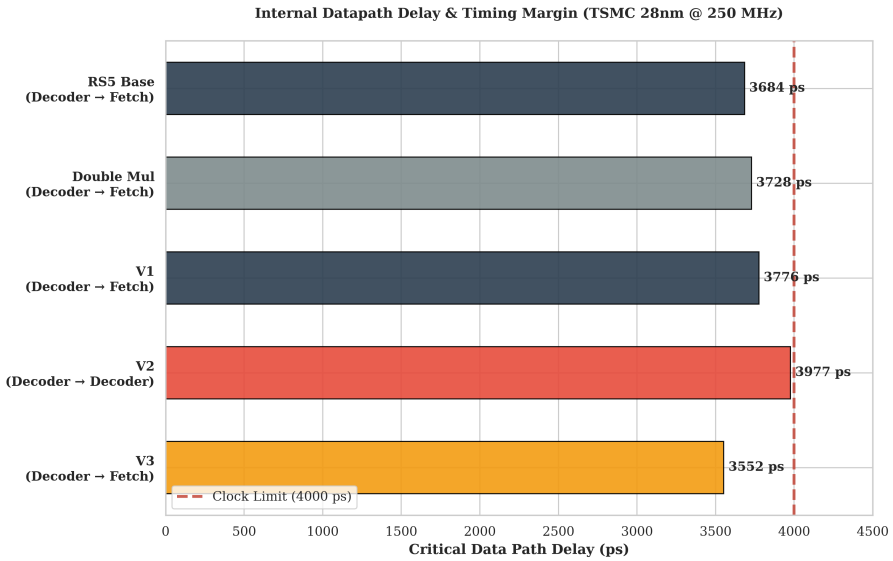


Figure 5.9: Critical data path delay from worst-case report timing.

The structural decoupling introduced in the V3 architecture reduced this routing pressure, isolating the cryptographic scaling logic from the global routing fabric and preventing the accelerator datapath from extending the processor critical path. As a result, V3 reports a critical path delay of 3552 ps, which remains below the 4.0 ns target period required for 250 MHz operation. These results indicate that the V3 architecture satisfies the target timing constraint while preserving the accelerator integration within the standard-cell implementation.

To validate the timing closure under nominal operating conditions, the critical paths were analyzed at the typical corner (0.90V, 25℃). As shown in Table 5.10 and Figure 5.10, the room-temperature environment accelerates standard-cell propagation. Under these conditions, the baseline RS5 delay drops to 2940 ps. It is crucial to note that, in this nominal scenario, the localized congestion of the V2 architecture is somewhat masked by the faster silicon, dropping to 3192 ps and returning its endpoint to the Fetch stage. However, the final V3 architecture continues to demonstrate superior routing stability, tracking the baseline with a 2827 ps delay. This confirms that the final serialized design operates with a timing margin (+1173 ps slack) during day-to-day typical workloads.

Table 5.10: Detailed Setup Timing Analysis and Critical Path Tracing (Typical Nominal 0.90V, 25℃ @ 250 MHz)

Architecture	Critical Startpoint	Critical Endpoint	Data Path Delay (ps)
Baseline RS5	DECODER	FETCH	2940
Double Mul Exp.	DECODER	FETCH	2993
Kyber V1	DECODER	FETCH	3053
Kyber V2	DECODER	FETCH	3192
Kyber V3	DECODER	FETCH	2827

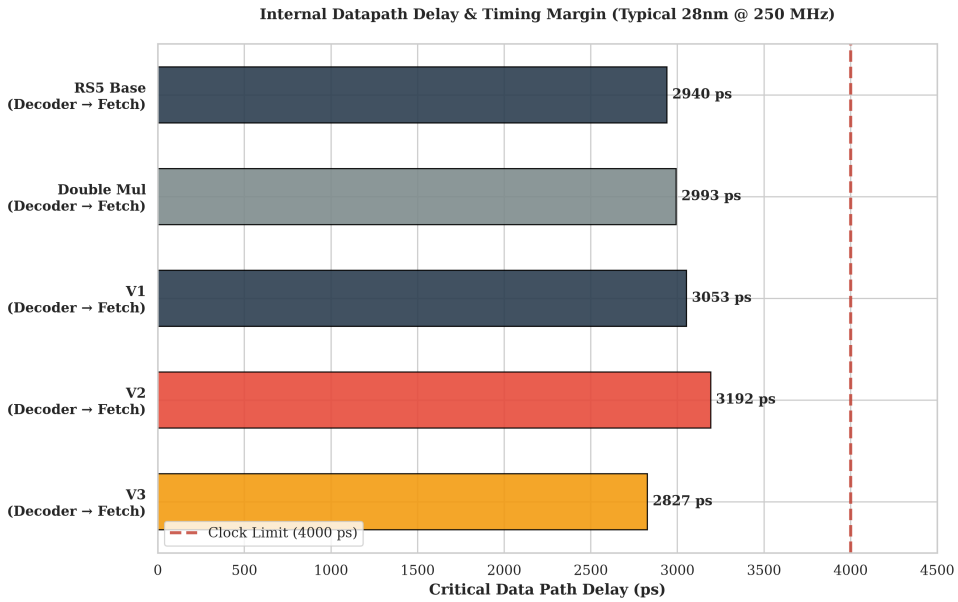


Figure 5.10: Critical data path delay from typical-case report timing.

To ensure that physical routing bottlenecks do not migrate under extreme environmental conditions, we re-evaluated the point-to-point critical paths at the best-case process corner of 0.99 V and (-40℃). Table 5.11 and Figure 5.11 summarize the resulting data path delays. The results show that reduced voltage and temperature reduce the core's baseline critical path to 2,299 ps. Despite these favorable silicon conditions, the multiplexed V2 architecture still exhibits an increased delay of 2,504 ps, indicating that routing complexity remains the dominant timing constraint. In contrast, the post-quantum arithmetic logic in the V3 architecture remains isolated by register boundaries across all PVT corners. With a critical path delay of 2,245 ps, V3 maintains consistent timing behavior and demonstrates resilience to process, voltage, and temperature variations.

Table 5.11: Best-Case Setup Timing Analysis and Critical Path Tracing (0.99V, -40℃ @ 250 MHz)

Architecture	Critical Startpoint	Critical Endpoint	Data Path Delay (ps)
Baseline RS5	DECODER	FETCH	2299
Double Mul Exp.	DECODER	FETCH	2385
Kyber V1	DECODER	FETCH	2401
Kyber V2	DECODER	FETCH	2504
Kyber V3	DECODER	FETCH	2245

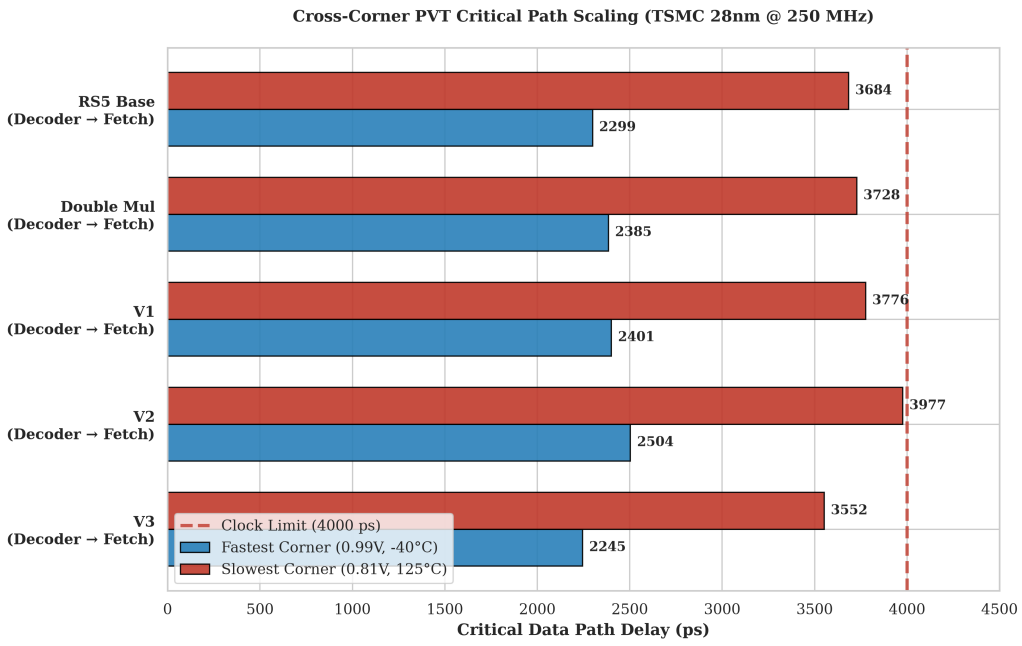


Figure 5.11: Critical data path delay from best-case report timing.

5.2.3 Power Dissipation

Power dissipation was evaluated across three PVT corners to characterize the behavior of the baseline RS5 core and the Kyber-enhanced architectural revisions under distinct voltage and temperature conditions. Tables 5.12, 5.13, and 5.14 report leakage, dynamic, and total power at 250 MHz for the worst-case slow corner, the typical corner, and the best-case fast corner, respectively.

Table 5.12: ASIC Power Dissipation (TSMC 28nm, **Worst**-Case Corner: 0.81V, 125℃, 250 MHz).

Architecture Revision	**Leakage Power (mW)**	**Dynamic Power (mW)**	**Total Power (mW)**
Baseline RS5 (No Kyber)	1.064	3.036	4.100
Double Mul Experiment	1.131	3.185	4.316
Kyber V1	1.090	2.972	4.062
Kyber V2	1.187	3.227	4.414
Kyber V3	1.084	3.149	4.233

Table 5.13: ASIC Power Dissipation (TSMC 28nm, **Typical** Corner: 0.90V, 25℃, 250 MHz).

Architecture Revision	**Leakage Power (mW)**	**Dynamic Power (mW)**	**Total Power (mW)**
Baseline RS5 (No Kyber)	0.073	3.715	3.789
Double Mul Experiment	0.078	3.892	3.970
Kyber V1	0.075	3.649	3.724
Kyber V2	0.082	3.953	4.035
Kyber V3	0.075	3.843	3.918

Table 5.14: ASIC Power Dissipation (TSMC 28nm, **Best**-Case Fast Corner: 0.99V, -40℃, 250 MHz.)

Architecture Revision	**Leakage Power (mW)**	**Dynamic Power (mW)**	**Total Power (mW)**
Baseline RS5 (No Kyber)	0.013	4.591	4.604
Double Mul Experiment	0.014	4.821	4.835
Kyber V1	0.013	4.497	4.510
Kyber V2	0.014	4.878	4.892
Kyber V3	0.013	4.756	4.769

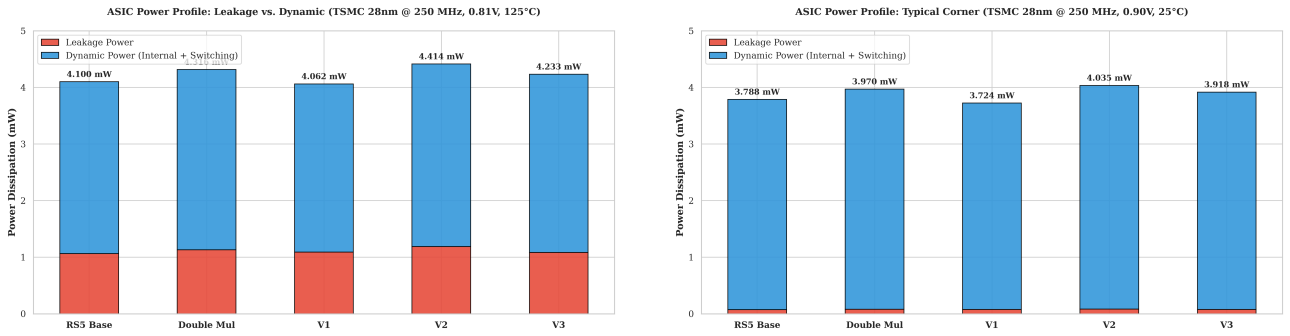
At the worst-case slow corner, defined at 0.81 V and 125℃, leakage power becomes a relevant component of the total dissipation due to the high operating temperature. As shown in Table 5.12, the baseline RS5 core dissipates 4.100 mW, with 1.064 mW from leakage and 3.036 mW from dynamic power. The Kyber V3 architecture increases the total

power to 4.233 mW, mainly due to the additional logic, pipeline registers, and clocked structures introduced by the Kyber datapath. Nevertheless, the total increase remains moderate, corresponding to 0.133 mW over the baseline design.

Table 5.13 reports the power results at the typical corner, defined at 0.90 V and 25 °C. Under nominal temperature, leakage power is substantially reduced and remains below 0.083 mW for all Kyber-enhanced versions. Consequently, total power is dominated by dynamic switching activity. The baseline RS5 core dissipates 3.789 mW, while Kyber V3 dissipates 3.918 mW. The increase observed in Kyber V3 is consistent with the larger clock tree, additional pipeline buffering, and expanded cryptographic datapath.

The best-case fast corner, defined at 0.99 V and -40 °C, is reported in Table 5.14. In this condition, leakage power becomes almost negligible due to the low temperature, reaching only 0.013 mW for Kyber V3. However, the increased supply voltage raises dynamic power, which dominates total dissipation across all architectural revisions. Kyber V3 dissipates 4.769 mW in this corner, remaining below 5 mW even under the highest evaluated supply voltage.

The graphical comparison in Figure 5.12 summarizes the leakage and dynamic power contributions for two representative operating conditions. Figure 5.12a shows the worst-case slow corner, where leakage is more visible due to the high temperature. Figure 5.12b shows the typical corner, where leakage is reduced, and total power is almost entirely determined by dynamic power. These results indicate that the Kyber extensions increase power primarily through additional switching activity, while leakage remains strongly dependent on the evaluated PVT corner.



(a) Worst-case slow corner.

(b) Typical corner.

Figure 5.12: ASIC power dissipation profiles showing leakage and dynamic power components for representative PVT conditions: (a) worst-case slow corner and (b) typical corner.

It is important to note that the power figures presented in this evaluation are static estimations generated directly by the synthesis tool (via the `report_power` command in Cadence Genus). These values rely on the tool's default statistical toggle rates across the netlist. A dynamic, post-synthesis simulation with back-annotated switching activity was not performed. Therefore, while these static reports provide a reliable metric to quantify the hardware overhead of the Kyber extensions across different architectural revisions, they

serve as an early-stage baseline rather than the absolute real-world power dissipation expected during physical execution.

5.2.4 Instruction Execution Latency

While ASIC guarantees higher operating frequencies and power efficiency, the actual throughput of a custom cryptographic accelerator is determined by its execution latency, measured in clock cycles per instruction. Table 5.15 and Figure 5.13 summarize the cycle counts required to retire the custom hardware instructions across the evaluated architectural revisions.

Table 5.15: Hardware Execution Latency Across Cryptographic Instruction Sets

Instruction Category	**Microarchitecture Revision**	**Execution Latency (Cycles)**
Standard Crypto (Zbkb, Zknh)	Native ALU Integration	1
Kyber Multiply (`kybermul`)	Double Mul Experiment	4
	Kyber V1	4
	Kyber V2	14
	Kyber V3	8
Kyber Compress (`kybercompress`)	Double Mul Experiment	2
	Kyber V1	2
	Kyber V2	2
	Kyber V3	6

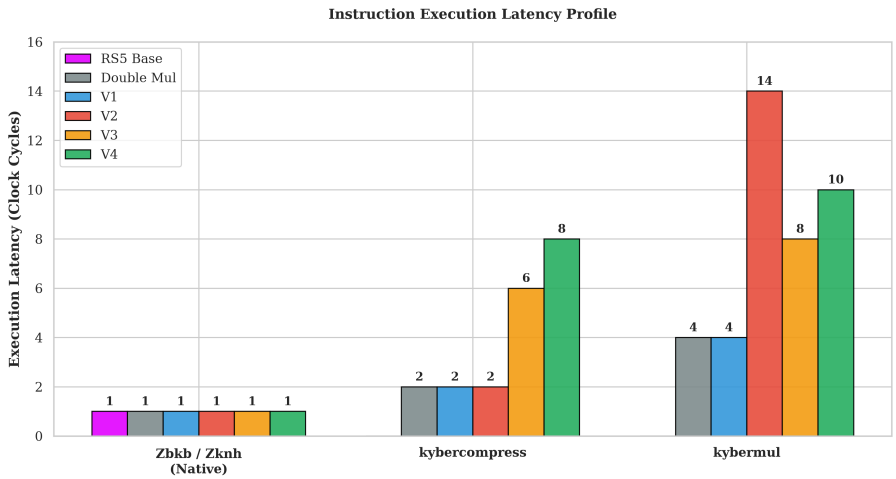


Figure 5.13: Execution latency profile detailing the clock cycles required for scalar cryptographic extensions and post-quantum Kyber instructions across architectural revisions.

The scalar cryptographic extensions (Zbkb and Zknh) inherently map directly into the native RISC-V execution stage. Because these operations rely on bit-manipulation logic rather than deep arithmetic trees, they successfully retire in a single clock cycle.

Conversely, the mathematical complexity of the post-quantum Kyber operations necessitates a multi-cycle execution model. The initial Version 1 architecture for `kybermul` forced the multiplication and modular reduction to occur within a 4-cycle window. However, as established in the logical synthesis evaluations, this combinational evaluation caused severe routing congestion and critical timing violations. The attempt to mitigate this by multiplexing the ALU in Version 2 resulted in a severe latency penalty, stalling the processor pipeline for 14 cycles due to inefficient state machine transitions.

The transition to the fully serialized Version 3 architecture established performance equilibrium. By dividing the `kybermul` operation across explicit register boundaries, the execution latency stabilized at 8 clock cycles. Similarly, the `kybercompress` instruction scaled from 2 cycles in its combinational form to a stable 6 cycles in its final sequential implementation. This deterministic latency increase represents an efficient trade-off: exchanging a marginal reduction in instruction-level parallelism for absolute logical timing closure and guaranteed operation at chosen frequencies inside the project.

5.2.5 Performance and Energy Efficiency

To isolate the performance gains, the ML-KEM reference software was selectively compiled using targeted ISA flags, enabling or disabling the custom `X-Kyber` instructions and the standard `Zbkb` extension.

Table 5.16 and Figure 5.14 detail the execution latency across the core ML-KEM operations. The baseline RV32I software implementation requires 1,837,178 cycles for decapsulation. Enabling only the standard `Zbkb` extension reduces this latency to 1,643,909 cycles by accelerating the Keccak permutations within the SHAKE functions. Conversely, enabling only the custom `X-Kyber` extension yields a higher performance gain, dropping the latency to 1,231,432 cycles by accelerating the NTT polynomial arithmetic.

Table 5.16: ML-KEM Execution Latency Analysis: Impact of Custom (X-Kyber) and Standard (Zbkb) Extensions

Hardware Configuration	**KeyGen (Cycles)**	**Encaps (Cycles)**	**Decaps (Cycles)**
Baseline (No X-Kyber, No Zbkb)	1,335,039	1,773,628	1,837,178
+ Zbkb Only (No X-Kyber)	1,115,385	1,504,659	1,643,909
+ X-Kyber Only (No Zbkb)	1,056,929	1,348,241	1,231,432
X-Kyber + Zbkb	**825,568**	**1,058,063**	**1,007,436**

Maximum performance is achieved when both extensions are enabled. The `X-Kyber + Zbkb` configuration completes the decapsulation in just 1,007,436 cycles. This represents a 45.1% reduction in total execution time compared to the baseline, demonstrating that the

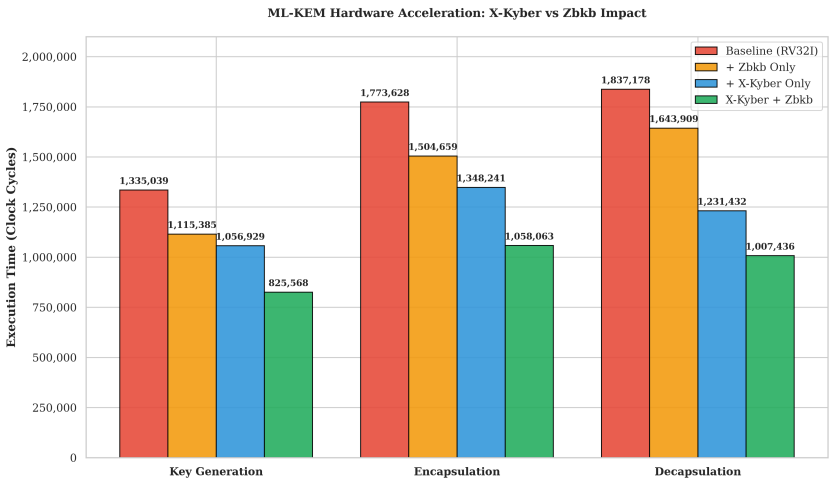


Figure 5.14: Execution latency of ML-KEM operations, demonstrating the cycle reductions achieved by the `X-Kyber` and `Zbkb` instruction sets.

standard `Zbkb` extension and the custom `X-Kyber` datapath provide complementary acceleration for different components of the ML-KEM workload.

The total energy dissipated during cryptographic operations is the product of the power and the execution time. As presented in Table 5.17 and Figure 5.15, executing the baseline software implementation demands 27.84 μJ to complete a single decapsulation. In contrast, the fully accelerated `X-Kyber + Zbkb` setup finishes the workload in a shorter timeframe, constraining the total energy consumption to just 15.79 μJ. This 42.3% reduction in energy consumption demonstrates that the custom instruction set and the optimized datapath inherently deliver an efficient physical implementation, validating the architectural choices made for the post-quantum accelerator.

Table 5.17: ML-KEM Energy Consumption (250 MHz).

Hardware/Software Configuration	**Active Power (mW)**	**KeyGen Energy (μJ)**	**Encaps Energy (μJ)**	**Decaps Energy (μJ)**
Baseline RV32I (No X-Kyber, No Zbkb)	3.789	20.23	26.88	27.84
+ Zbkb Only (No X-Kyber)	3.849	17.17	23.17	25.31
+ X-Kyber Only (No Zbkb)	3.776	15.96	20.36	18.60
X-Kyber + Zbkb	**3.918**	**12.94**	**16.58**	**15.79**

** Note: Energy (μJ) = Active Power (mW) $\times$ Total Cycles $\times$ Clock Period (4.0 ns).*

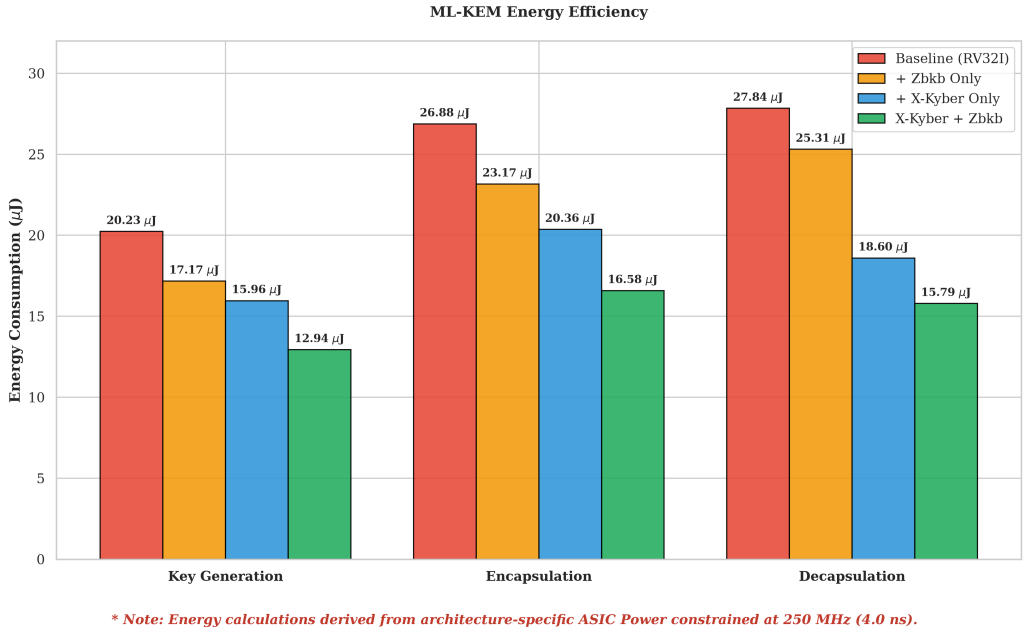


Figure 5.15: Energy consumption analysis for the hardware configurations at 250 MHz).

6. CONCLUSION

This chapter concludes the present work by summarizing its main contributions, evaluating the results against the objectives, and outlining directions for future research in the context of post-quantum cryptographic hardware integration.

This work addressed the need for hardware acceleration of post-quantum cryptographic algorithms in resource-constrained environments. Specifically, it tackles the challenges of integrating the ML-KEM algorithm into a RISC-V embedded processor. By developing a custom scalar accelerator as an ISA extension, this work demonstrates that computationally intensive operations, including the Number Theoretic Transform (NTT) for polynomial multiplication and data compression, can be implemented in a deep submicron ASIC while preserving the physical integrity of the host processor.

The evaluation, conducted through FPGA prototyping and 28 nm standard-cell ASIC synthesis, quantified the physical cost of cryptographic integration. Experimental results showed that the initial combinational implementations (Version 1) and the multiplexed architectures (Version 2) introduced routing congestion and timing violations. Consequently, these designs failed to scale and became the bottleneck limiting the processor's operating frequency. In contrast, the serialized architectures (Version 4 for FPGA and Version 3 for ASIC) established an active area-time tradeoff. By enforcing register boundaries and encapsulating the arithmetic logic within finite state machines, the proposed accelerator achieved timing closure independently of the processor routing fabric. As a result, the core sustained a worst-case theoretical maximum operating frequency of 278.9 MHz in a 28 nm technology node while maintaining a total power footprint of 4.233 mW under the slowest and highest-temperature silicon conditions.

From an architectural perspective, the execution latency analysis confirmed that the custom instructions translate physical serialization into absolute performance gains, establishing a quantified area-performance trade-off. The integration imposed overheads of a 5.8% increase in silicon footprint and a 3.4% rise in typical active power. In return, the combined `X-Kyber` and `Zbkb` extensions reduced the ML-KEM decapsulation execution time by 45.1%. Because the processor stalls the general-purpose fetch-decode pipeline during these local execution states, this latency reduction decreased the total energy consumption by 43.3%, dropping from 27.84 μJ in the software baseline to 15.79 μJ. The deterministic 8-cycle latency for polynomial multiplication and 6-cycle latency for data compression demonstrate that sacrificing single-cycle instruction-level parallelism is a necessary structural compromise to achieve timing closure and power efficiency.

Securing edge devices against the quantum threat is an ongoing pursuit, and several avenues remain open for future work:

- **Side-Channel Attack (SCA) Resistance:** While this work optimized the core for area, power, and frequency, it did not implement countermeasures against physical side-channel attacks, such as Differential Power Analysis (DPA). Future iterations must investigate the hardware overhead of integrating masking or hiding techniques into the custom accelerator.
- **Integration of ML-DSA (Dilithium):** This work focused on the ML-KEM key encapsulation mechanism; however, a post-quantum secure system also requires digital signatures. Future research should explore implementing the ML-DSA algorithm as a complementary ISA extension. Since both algorithms share polynomial arithmetic, particularly the NTT, evaluating hardware resource sharing between ML-KEM and ML-DSA accelerators presents an opportunity to optimize silicon area.
- **Hardware Acceleration of Polynomial Decompression:** The current microarchitecture implemented a custom instruction to accelerate data compression (`kybercompress`). Nevertheless, the inverse operation, decompression, remains a computational bottleneck during the ML-KEM decapsulation flow. Developing a dedicated `kyberdecompress` hardware module and integrating it into the RS5 execution stage would close this loop, providing a comprehensive cryptographic acceleration suite.

This work presented a validated approach to integrating post-quantum security into the RISC-V ecosystem, demonstrating that standard-cell serialization is a physical necessity rather than a cost-free optimization. Although this approach increases instruction latency and requires additional sequential resources, it improves routing and timing predictability. This architectural compromise is essential for achieving reliable cryptographic execution in resource-constrained environments.

REFERENCES

Alnaseri, O., Himeur, Y., Atalla, S., and Mansoor, W. (2025). Complexity of Post-Quantum Cryptography in Embedded Systems and Its Optimization Strategies. In *International Wireless Communications and Mobile Computing (IWCMC)*, pages 776–781. hhttps://doi.org/10.1109/IWCMC65282.2025.11059522.

Avanzi, R., Bos, J., Ducas, L., Kiltz, E., Lepoint, T., Lyubashevsky, V., Schanck, J. M., Schwabe, P., Seiler, G., and Stehlé, D. (2021). CRYSTALS-Kyber Algorithm Specifications and Supporting Documentation (Round 3). Technical report. https://pq-crystals.org/kyber/data/kyber-specification-round3-20210804.pdf.

Avižienis, A., Laprie, J.-C., Randell, B., and Landwehr, C. (2004). Basic concepts and taxonomy of dependable and secure computing. *IEEE Transactions on Dependable and Secure Computing*, 1(1):11–33. https://doi.org/10.1109/TDSC.2004.2.

Bellare, M., Desai, A., Pointcheval, D., and Rogaway, P. (1998). Relations among notions of security for public-key encryption schemes. In *Advances in Cryptology — CRYPTO '98*, pages 26–45. Springer.

Borowski, M. (2013). The sponge construction as a source of secure cryptographic primitives. In *2013 Military Communications and Information Systems Conference*, pages 1–5.

Cooley, J. W. and Tukey, J. W. (1965). An algorithm for the machine calculation of complex Fourier series. *Mathematics of Computation*, 19(90):297–301.

Damgård, I. B. (1989). A design principle for hash functions. In *Advances in Cryptology — CRYPTO '89*, pages 416–427. Springer.

Di Matteo, S. et al. (2024). CRYPHTOR: A memory-unified NTT-based hardware accelerator for post-quantum CRYSTALS algorithms. *Nome da Revista ou Conferência*.

Dinh, N. N., Pham, H. L., Le, V. T. D., Vu, T. H., Tran, V. D., and Nakashima, Y. (2024). Kyberator: A High-Efficiency FPGA-Based Multi-Mode CRYSTALS-Kyber Accelerator for Quantum-Resistant Security Applications. In *International SoC Design Conference (ISOCC)*, pages 308–309. https://doi.org/10.1109/ISOCC62682.2024.10762592.

Gentleman, W. M. and Sande, G. (1966). Fast Fourier transforms: for fun and profit. In *Proceedings of the November 7-10, 1966, Fall Joint Computer Conference*, pages 563–578. ACM.

Gewehr, C., Luza, L., and Moraes, F. G. (2024). Hardware Acceleration of Crystals-Kyber in Low-Complexity Embedded Systems With RISC-V Instruction Set Extensions. *IEEE Access*, 12:94477–94495. https://doi.org/10.1109/ACCESS.2024.3416812.

Goldwasser, S. and Micali, S. (1984). Probabilistic encryption. *Journal of Computer and System Sciences*, 28(2):270–299.

Kagai, F., Branch, P., But, J., and Allen, R. (2025). Harvest-now, decrypt-later: A temporal cybersecurity risk in the quantum transition. *Telecom*, 6(4).

Kannwischer, M. J., Rijneveld, J., Schwabe, P., and Stoffelen, K. (2019). pqm4: Testing and benchmarking NIST PQC on ARM Cortex-M4. In *Second PQC Standardization Conference*. National Institute of Standards and Technology (NIST).

Lippert, J. (2014). The Fujisaki-Okamoto Transformation. Master's thesis, Paderborn University. https://cs.uni-paderborn.de/fileadmin-eim/informatik/fg/cuk/Lehre/Abschlussarbeiten/Bachelorarbeiten/2014/BA_Lippert_FOT_final.pdf.

Merkle, R. C. (1989). One way hash functions and DES. In *Advances in Cryptology — CRYPTO '89*, pages 428–446. Springer.

NIST (2024a). FIPS 203: Module-Lattice-Based Key-Encapsulation Mechanism Standard. Technical report, U.S. Department of Commerce. https://nvlpubs.nist.gov/nistpubs/FIPS/NIST.FIPS.203.pdf.

NIST (2024b). NIST IR 8547: Transition to Post-Quantum Cryptography Standards. Technical report, U.S. Department of Commerce. https://nvlpubs.nist.gov/nistpubs/ir/2024/NIST.IR.8547.ipd.pdf.

Nunes, W., Dal Zotto, A., Borges, C., and Moraes, F. G. (2024). RS5: An Integrated Hardware and Software Ecosystem for RISC-V Embedded Systems. In *LASCAS*, pages 1–5. https://doi.org/10.1109/LASCAS60203.2024.10506171.

Paar, C., Pelzl, J., and Güneysu, T. (2024). *Understanding Cryptography: From Established Symmetric and Asymmetric Ciphers to Post-Quantum Algorithms*. Springer, 2nd edition.

Rackoff, C. and Simon, D. R. (1992). Non-interactive zero-knowledge proof of knowledge and chosen ciphertext attack. In *Advances in Cryptology — CRYPTO '91*, pages 433–444. Springer.

RISC-V (2021). *RISC-V Cryptography Extensions Volume I: Scalar & Entropy Source Instructions*. https://docs.riscv.org/reference/isa/extensions/crypto-scalar/_attachments/riscv-crypto-spec-scalar.pdf.

RISC-V (2024). *The RISC-V Instruction Set Manual, Volume I: Unprivileged ISA*. https://docs.riscv.org/reference/isa/_attachments/riscv-unprivileged.pdf.

Rivest, R. (1992). The MD4 Message-Digest Algorithm. Request for Comments 1320, Internet Engineering Task Force (IETF).

Satriawan, A. and Mareta, R. (2024). A Complete Beginner's Guide to the Number Theoretic Transform (NTT). Cryptology ePrint Archive, Paper 2024/585. https://eprint.iacr.org/2024/585.

Thamilarasi, V., Naik, P. K., Sharma, I., Porkodi, V., Sivaram, M., and Lawanyashri, M. (2024). Quantum Computing - Navigating the Frontier with Shor's Algorithm and Quantum Cryptography. In *International Conference on Trends in Quantum Computing and Emerging Business Technologies (TQCEBT)*, pages 1–5. https://doi.org/10.1109/TQCEBT59414.2024.10545283.

Tiwari, A., Chauhan, R., Joshi, N., Devliyal, S., Aluvala, S., and Kumar, A. (2024). The quantum threat: Implications for data security and the rise of post-quantum cryptography. In *2024 IEEE 9th International Conference for Convergence in Technology (I2CT)*, pages 1–7.

TopTechBoy (2026). Rotate left svg - arduino tutorial 45: Understanding circular shift. https://toptechboy.com/arduino-tutorial-45-understanding-circular-shift-left-and-circular-shift-right-with-the-74hc595/1280px-rotate_left-svg-2/.

Yaman, F., Mert, A., and Savaş, E. (2021). A Hardware Accelerator for Polynomial Multiplication Operation of CRYSTALS-KYBER PQC Scheme. In *Design, Automation and Test in Europe Conference (DATE)*, pages 1020–1025. https://doi.org/10.23919/DATE51398.2021.9474139.

Yılmaz, E. and Özbudak, E. K. (2026). Simulating quantum attacks on classical cryptography with cuda-q. In *2026 5th International Informatics and Software Engineering Conference (IISEC)*, pages 672–677.

Zhang, J., Yan, Y., Huang, J., and Koç, Ç. K. (2024). Optimized Software Implementation of Keccak, Kyber, and Dilithium on RV32,64IMBV. Cryptology ePrint Archive, Report 2024/1515. https://eprint.iacr.org/2024/1515.pdf.