POLITECNICO DI TORINO

Master's Degree in Electronic Engineering

Politecnico di Torino

Master's Degree Thesis

Hardware architecture for CRYSTALS-Kyber cryptographic primitives

Supervisors

Prof. Guido MASERA

Prof. Maurizio MARTINA

Kristjane KOLECI

Candidate

Alessandra DOLMETA

October 2022

Science can amuse and fascinate us all,
but it is engineering that changes the world.
Isaac Asimov

La scienza può divertire e affascinare tutti noi,
ma è l'ingegneria a cambiare il mondo.
Isaac Asimov

Summary

In the modern era, cryptography is essential for online communication security. It plays a crucial role for information reliability, and it is based on encryption and authentication algorithms that are constructed using hard mathematical and computationally infeasible problem.
However, once quantum computers become widespread, many of the regularly used cryptosystems will be completely useless, as they can be easily attacked and broken. Their growth represents a concrete threat to classical public-key protocols.
This is why researchers and scientists are developing post-quantum cryptography (PQC) algorithms, exploiting problems that can be invulnerable to quantum computer attacks.
Unlike quantum cryptography, which rely on quantum computing and quantum communication environments, PQC based cryptosystems run on classical computer, providing sufficient security.
In 2015, NIST -National Institute of Standards and Technology- launched a public evaluation process to standardize quantum-resistant public key algorithms.
After three round of solicitations, among the different finalists, there were four similar KEM algorithms. Here, cost and performance becomes the most crucial selection criteria. In NIST's current view, lattice-based algorithms are the most promising families, achieving a good balance in security.
In fact, in 2022, CRYSTALS-Kyber algorithm has been selected among the four KEM finalists to be finalized in about two years. It is characterized by comparatively small encryption keys (that two parties can exchange easily), as well as a good speed of operation.
One of the fundamental building blocks of CRYSTALS-Kyber, and more generally of any PQC algorithm, is the one relating to PQC primitives. PQC primitives guarantee the security of the algorithm, performing a specialized task with incredible accuracy and precision. Two types of primitives can be distinguished: security primitives (AES, SHA3 and Keccak) and computation primitives (Barrett reduction, Montgomery reduction and NTT).

Designing a dedicated architecture that can realize these primitives can significantly reduce hardware resource occupation, obtaining a component with higher performances and improving algorithm's performances.
The aim of this study is to provide a dedicated hardware-based implementation of the most consuming part of NIST-PQC-finalists Crystals-KYBER.
The architecture has been implemented for Kyber-768 (III-security level) and realize all the SHA-3 primitives used in the algorithm.

Keywords: Post-Quantum Cryptography, PQC Primitives, CRYSTALS-Kyber, Kyber hardware Design, SHA-3, Keccak

Table of Contents

List of Tables

List of Figures

Acronyms

AES

Advanced Encryption Standard

CPA

Chosen-Plaintext Attack

CPA-PKE

Chosen-Plaintext Attack-Public Key Encryption

FSM

Finite State Machine

FSMD

Finite State Machine with Data path

IND-CCA2

Indistinguishability under adaptive chosen ciphertext attack

IND-CPA

Indistinguishability under Chosen-Plaintext Attack

KEM

Key Encapsulation Mechanisms

LFSR

Linear Feedback Shift Register

LWE

Learning With Errors

LUT

Look-Up Table

MLWE

Module Learning With Errors

MVP

Multivariate Polynomial

NIST

National Institute of Standards and Technology

NTT

Number-Theoretic Transform

PQC

Post Quantum Cryptography

RC

Round Constant

RSA

Rivest–Shamir–Adleman

ROM

Read-Only Memory

SHA

Secure Hash Algorithm

SVP

Short Vector Problem

Chapter 1

Introduction

In the last years, there has been progress in building quantum computers.
Quantum computers are machines that use quantum physics properties to store data and perform computations. In fact, while classical computers encode information in binary bits that can either be 0s or 1s, in a quantum computer the basic unit of memory is a quantum bit (*qubit*). Its main properties is known as quantum superposition: a series of *qubits* can represent different things simultaneously.
For instance, sixteen bits are enough for a classical computer to represent any number between 0 and 65535. But sixteen *qubits* are enough for a quantum computer to represent every number between 0 and 65535 at the same time.
This is an extremely advantageous result. It is as a incredibly powerful calculator programmed with deep domain expertise. Quantum computers will be fast and effective, performing calculations in just a few seconds for which today's computers would requires instead decades.
However, like any incredible innovation, there are also dreadful consequences.
It is fair to assume that with the amount of research going on in the area of quantum computers, there are high chances they become a reality within a few decades. If large-scale quantum computers are realized, they would threaten the security of most commonly-used public-key cryptosystems. For external hackers, it could becomes simpler to access to *sensitive* information.
For decades, our data have been protected using the same cryptographic systems, as RSA or AES (Advanced Encryption Standard). These are hard mathematical problems that would keep a classical computer busy for thousands of years. No matter how powerful they are, they are not able to crack these problems and decrypt protected data.
In classical cryptography, both in the case of symmetric and asymmetric encryption, the security of communication depends on the secrecy of the of the key. Keeping keys save, it's almost impossible for data to be compromised, unless hackers get somehow their hands on the encryption keys.

Data are safe, or at least they are until powerful quantum computers arrive.
This bring us into the next cryptographic chapter: **post-quantum cryptography**.
PQC is an evolution of classical cryptography. It is based on math problems too, but they are much more difficult and complex, in order to be more robust to quantum computer attacks. To prevent a global collapse of infrastructure, designing quantum-safe cryptosystems has become crucial.
In response to this threat, the National Institute of Standards and Technology (**NIST**) started a competition in 2015 to encourage researchers to propose asymmetric cryptographic schemes that would be resistant to quantum computers.
The major classes of post quantum cryptography algorithm are:

- **Lattice-based cryptography** algorithms offer the best performances and are built on the hardness of SVP (Short Vector Problem). By definition, it asks to find a nonzero vector in a lattice;
- **Code-based cryptography** offers a more conservative approach using large keys. Researchers are trying to find a way to reduce the key size without compromising the security of these kinds of algorithm;
- **Multivariate polynomial cryptography** is based on multivariate polynomial (MVP) algorithms over finite fields. MVPs are preferred as signature schemes as they offer the shortest signatures;
- **Isogeny-based**: uses isogenies between elliptic curves over finite fields.
- **Hash-based digital signature** relies on cryptographic hash functions;
- others.

Each candidate in NIST competition belongs to one of these two functions:

1. **digital signature**: is based on the principle that the sender signs the message with a private key, and the receiver verifies this signature using the sender's public key.
2. **key encapsulation mechanism** (KEM): is one of the most common algorithms that can be used for key exchange. Traditional encryption-decryption protocols are used to encrypt a message using the sender's public key, which is then decrypted by the receiver using his private key.

In Table 1.1 is reported a summary of digital signature (**Sign**) and key encapsulation (**KEM**) submissions for Round 1 and Round 2 (within braces), summarizing their mathematical complexity. [1].

PQC	**The hard problem**	**KEM**	**Sign**	**Total**
Lattice	Find shortest vector, closest vector	5 (3)	23 (9)	28 (12)
Code	Decode random linear code	3 (0)	17 (7)	20 (7)
Multivariate	Solve multivariate quadratic equations	8 (4)	2 (0)	10 (4)
Hash	Second pre-image resistance of hash function	3 (2)	0 (0)	3 (2)
Isogeny	Find isogeny map between elliptic curves with same number of points	0 (0)	1 (1)	1 (1)
Other	-	2 (0)	5 (0)	7 (0)
Total	-	21 (9)	48 (17)	69 (26)

Table 1.1: NIST candidates

Figure 1.1 shows the evolution of the NIST competition in recent years.

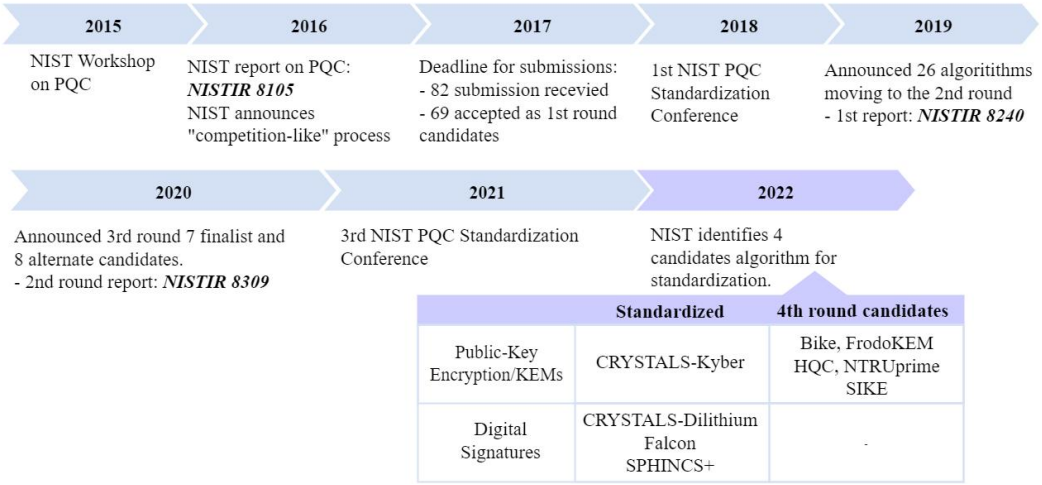

Figure 1.1: NIST announcements

Initially, 82 submissions were received. Between them, only 69 candidates were qualified enough for NIST competition's round 1. In 2018, there was the first NIST PQC standardization conference. Then, since January 2019, 26 candidates are competing in round 2.
In 2020, for the third round, 7 candidates were chosen as finalists and 8 candidates were considered as alternates.
In 2022, the U.S. department of commerce's NIST has chosen the first group of encryption tools: the four selected encryption algorithms will become part of **NIST's post-quantum cryptographic standard**, expected to be finalized in about two years. For general encryption, used when we access secure websites, NIST has selected the **CRYSTALS-Kyber algorithm**, which will be the algorithm analyzed in this thesis work.

1.1 Thesis objectives and structure

This thesis has been developed with the aim of exploring a dedicated digital hardware architecture to implement SHA3-algorithm for CRYSTALS-Kyber.
Using VHDL hardware description language, it has been designed one of the most computational-costly part of algorithm.
After this first introductory chapter 1, the thesis is structured as follow:

- **chapter 2** presents an introduction about cryptography. We discuss briefly symmetric key cryptography in subsection 2.1.2 and asymmetric key cryptography in subsection 2.1.3. In section 2.2, new cryptographic algorithm are discussed.
- **chapter 3** is dedicated to main modules of CRYSTAL-Kyber algorithm. In section 3.1 we give preliminaries, in section 3.2 we give a general description of `Kyber.CPAPKE` and `Kyber.CPAKEM`, and in section 3.3 we report a detailed profiling of the most expensive functions used (also see Appendix A).
- **chapter 4** is about SHA-3 algorithm. After an introduction about SHA3, in section 4.1 sponge functions are discussed, focusing on padding methodology and Keccak algorithm.
- **chapter 5** presents first the hardware implementation of the SHA3 core, and then is focused on the hardware implementation of the control unit, concentrating our attention on the specific primitives required by CRYSTALS-Kyber. In particular, the implemented dedicated hardware has been designed specifically for the third level of security (**Kyber768**).
- **chapter 6** implementation results are reported and compared to the counterparts, focusing on results reported in papers dealing with hardware implementations of the SHA3 core for PQC applications.
- **chapter 7** collects the conclusion about this thesis, with a brief summary of the different research activity possible.

Chapter 2

From Cryptography to PQC

The term cryptography derived its name from the Greek word *kryptos*, which means hidden. This is the practise and study of secure communications techniques that allow only the sender and intended receiver of a message to view its contents.
It is closely associated to encryption and decryption. In particular:

- **encryption** is the process of converting plain text into an unintelligible format (cipher text);

- **decryption** is the process of converting the cipher text into a plain text back again upon arrival.

An encryption process almost always involves both an **algorithm** and a **key**.
The key is another piece of information that defines how the plain text will be modified by the algorithm in order to be encrypted.
Any malicious third-parties, known as adversaries, should not be able to determine anything about a key given a large number of plain text/ciphertext combinations which used the key.
To achieve a secure protocol, there is a number of requirements that must be met and guaranteed. In particular:

- **authentication**, therefore proving one's identity and authenticate users across services and systems.

- **integrity**, ensure that the received message has not been intercepted and modified in any way with respect to the original message sent by the sender. This measure determines accuracy and completeness of data and must control whenever the information has been accessed. In the case system users alter information, it must be also ensure dare that they are legitimately authorized to do it.

- **privacy/confidentiality**, ensuring that no one can read the message except the desired receiver, protecting information from unauthorized access.
- **non-repudiation**, which is a mechanism to prove that the sender really sent this message. It must provide a proof of the integrity and origin of data, authenticating the genuineness of the message with high confidence.

Cryptography can protect communications that traverse untrusted networks. There are two main types of attacks that an adversary may attempt to carry out on a network: **passive attacks** and **active attacks**.
The firsts involve an attacker simply listening on a network segment and waiting to read sensitive information. It may be online or offline, in the case some time is needed to decrypt data. The later, involve an attacker impersonating a client-server, intercepting communications in transit, and modifying the contents before passing them on to their intended destination.

Before we move on to modern cryptography remember, as we said, that any cryptographic system involves both an algorithm and a key.
Kerckhoffs, a Dutch cryptographer of the 19th century, believed that:

> *A cryptographic system should be secure even if everything about the system, except the key, is public knowledge.*

The point is that while it would be nice to keep our cryptographic system a secret, our adversaries will eventually figure it out. What we do need to keep secret is the **cryptographic key**. Its length is normally expressed in bits. Obviously, have a longer key makes the encrypted data more difficult to be cracked, but also implies longer time periods to perform encryption and decryption processes.

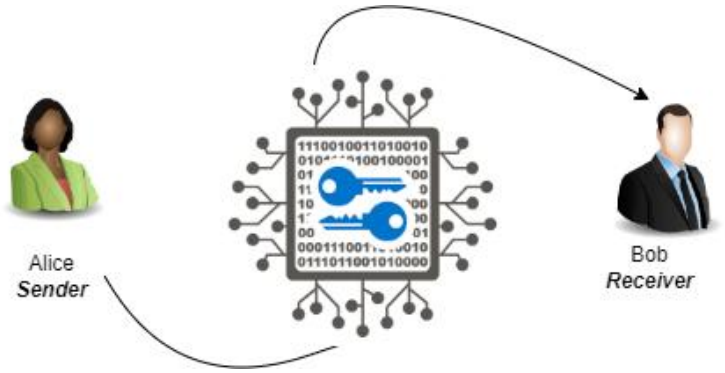

Figure 2.1: Generic Encryption

2.1 Cryptography

There are several ways of classifying cryptographic algorithms.
The most common one is to categorized them considering the number of keys that are employed for encryption and decryption.
The three types of algorithms that will be discussed are:

- **Hash Functions**: exploits mathematical transformation to irreversibly encrypt information, used for message integrity.
- **Symmetric Key Cryptography**: defined also as secret key cryptography, uses a single key for both encryption and decryption. It is primarily used for privacy and confidentiality. The most popular symmetric key methods are AES and ChaCha20.
- **Asymmetric Key Cryptography**: defined also as public key cryptography, uses one key for encryption and another for decryption. It is primarily used for authentication, non-repudiation, and key exchange.

2.1.1 Hash Function

Hashing methods take data and convert to a hash value, simply as reported in Figure 2.2. The most popular cryptographic hashing methods are: **MD5**, **SHA-1**, **SHA-2** and **SHA-3**.
Also Blake 3 is becoming popular, since it is one of the fastest hashing methods around. SHA-3 is based on Keccak, and it has been standardized by NIST. This is the hash module used in CRYSTALS-Kyber algorithm.
There are also non-cryptographic hashes, and which do not have the same security levels as the cryptographic hashes, but are often much faster in their operation. [2]

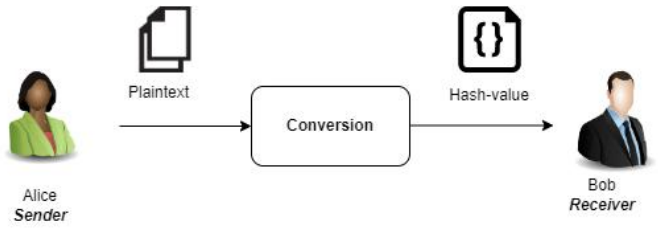

Figure 2.2: Hash Function

2.1.2 Symmetric Key Cryptography

The simplest method of encryption/decryption is to use the symmetric or "**secret key**" system. As shown in Figure 2.3, data is encrypted using a secret key, and then both the encoded message and secret key are sent to the recipient for decryption. Therefore, the same key is used for both parties.

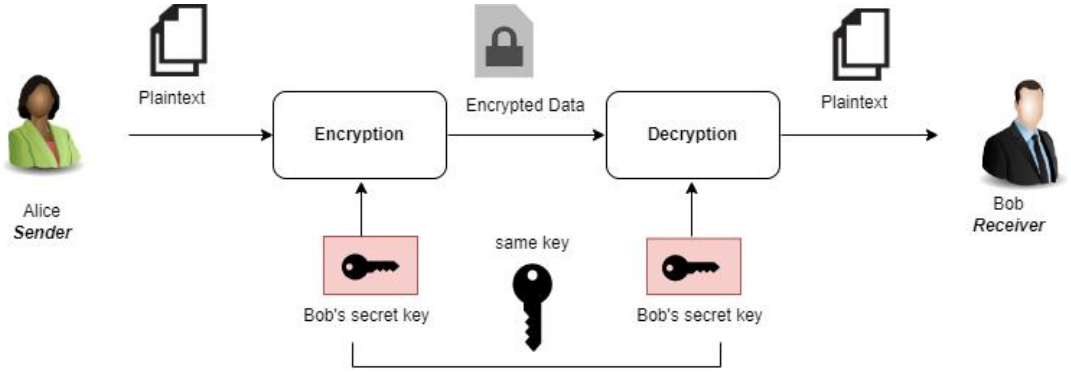

Figure 2.3: Secret-key encryption

This is simpler, faster and is widely used to keep data confidential.
It can be very useful for keeping a local hard drive private, where the secret key sharing is not a problem. In fact, the key cannot be send along with the message. Otherwise, if it is intercepted, a third party can decrypt and read it.
Symmetric-key encryption can use either **stream ciphers** or **block ciphers**.
Stream ciphers encrypt digits (usually bytes) or letters of a message one at a time. An example is given by substitution ciphers, which are well-known ciphers where the plain text is replaced with ciphertext, according to a fixed system. Here, the "*units*" may be single letters (the most common), pairs of letters, triplets of letters, etc., and being a fixed system, it can be easily decrypted using a frequency table. On the contrary, block ciphers take a number of bits and encrypt them as a single unit, padding the plain text so that it is a multiple of the block size.
Firstly, the most popular symmetric–key system was the Data Encryption Standard (**DES**). It was developed in the early 70's by IBM and used a 56 bit encryption key which can give around 2^{56} 256 combination to encrypt the plain text.[3]
In the last decades, it has been substitute by the Advanced Encryption Standard (**AES**) algorithm, approved by NIST in December 2001, which uses 128-bit blocks and a key size of 128, 192 or 256 bits. [4]

2.1.3 Asymmetric Key Cryptography

In order to increase the level of security, we need a way for communicating parties to establish a secure communications channel while only talking to each other across an inherently insecure network. To address this issue, cryptologists devised the asymmetric or "**public key**" system.
In this case, every user has **two keys**: one **public** and one **private**.
The public one is used to encrypt the message and it is sent to anyone the sender wishes to communicate with. Instead, the private key is shared with nobody, and it's necessary to decrypt those messages when they arrive.
Public key algorithms ensure confidentiality and authenticity in modern cryptosystems.

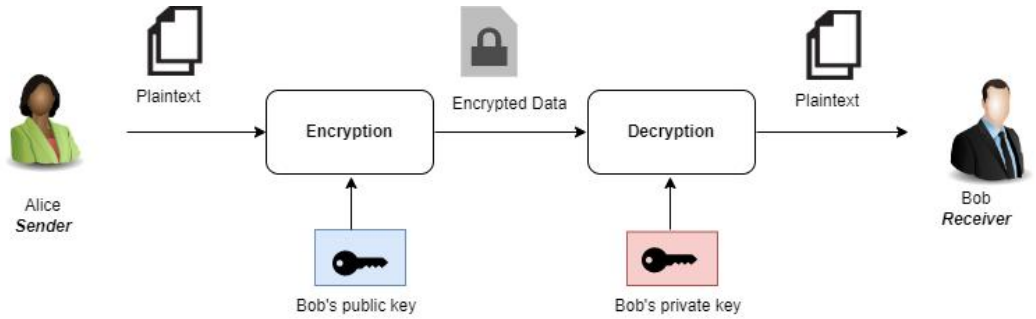

Figure 2.4: Public-key encryption

Public key encryption can be applied to encryption/decryption, digital signature and key exchange. The technique described in Figure 2.4 is the public key encryption/decryption, in which a message is encrypted with the intended recipient's public key, and decrypted with the private key.
This is one of the best-known applications of public key cryptography.
The other one is **digital signatures**. Here, a message is signed exploiting the sender's private key and can be verified by anyone who has access to the sender's public key. It is used to verify the authenticity of data.
The most widely used public-key cryptosystem is **RSA**, named for the three MIT mathematicians who developed it (Rivest–Shamir–Adleman). RSA uses a variable size encryption block and a variable size key. Its backbone is the difficulty of finding the prime factors of a composite number.

2.1.4 Asymmetric and symmetric cryptography differences

The main difference between asymmetric and symmetric cryptography is the number of keys used. In particular, as explained in subsection 2.1.3, asymmetric encryption algorithms use two different but related keys. Instead, symmetric encryption uses the same key to perform both functions.
Another difference between the two is the length of the keys: asymmetric keys need to be longer to offer the same level of security. In fact, in asymmetric encryption there is a mathematical link between the two keys. Since adversaries can potentially exploit this relation to crack the encryption, the keys involved must be consequently larger. As an example, a 2048-bit asymmetric key and a 128-bit symmetric key provide about an equivalent level of security.
In Table 2.1 is reported a small comparison between the two encryption techniques.

Symmetric Key Encryption	**Asymmetric Key Encryption**
Cipher text size is the same or smaller than the original plain text	Cipher text size is the same or larger than the original plain text
The encryption process is very fast	The encryption process is slow
Used to transport large amount of data	Used to transfer small amounts of data
Provides only confidentiality	Provides confidentiality, authenticity, and non-repudiation
Low resource utilization	High resource utilization
Security is less as only one key is used for both encryption and decryption purpose	It is more secure as two keys are used: one for encryption and the other for decryption

Table 2.1: Symmetric and asymmetric key encryption comparison

2.2 Post Quantum Cryptography

Post quantum cryptography indicates a class of cryptographic algorithms (generally public-key algorithms) that are thought enough to be secure against an attack by a quantum computer. Despite it is still early for quantum computing, migrating our extensive infrastructure from today's widely deployed algorithms to PQC alternatives is a complicated process, which can requires many years because of backward compatibility. [5]
Classic public-key cryptography algorithms are based on problems like factoring large integers (RSA) or discrete logarithm (ECC). However, these algorithms have been shown to be vulnerable to quantum attacks, since these problems would be extremely easy to be solved for a quantum computer.
We need to introduce advanced and secured cryptosystem.
Cryptographic algorithms are based on hard mathematical and computationally infeasible problems that are believed to be resistant to both conventional and quantum crypto-analysis. The most important quantum-safe approaches are the ones already discussed in Table 1.1. Some examples are:

- **hash-based**: Merkle signatures, Sphincs, Picnic;
- **code-based**: McEliece, Niederreiter;
- **lattice-based**: NTRU, learning with errors, ring-LWE, LWrounding;
- **multivariate cryptography**: multivariate quadratic;
- **isogeny-based cryptography**: super-singular elliptic curve isogenies.

A brief scheme of their main properties is reported in Figure 2.5. [6].

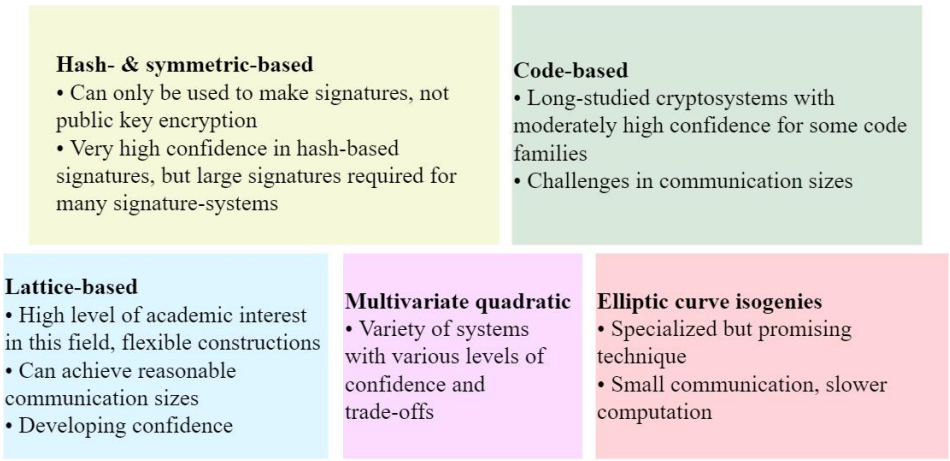

Figure 2.5: Most important quantum-safe approaches

These are some of the different families of math problems currently being investigated with each taken from a different branch of mathematics.
Over the last decade, cryptographers and mathematicians have been working over these problems and turning them into useful cryptographic schemes.
Considering NIST third round, between the seven finalists KEM primitives, two were code-based and the remaining five were lattice-based.
The latter is the most promising class of algorithms between the options considered.
Therefore, we will focus our attention on lattice-based cryptography, being CRYSTALS† Kyber algorithm based on the **hardness of solving the learning with-errors (LWE) problem over module lattices**.
In subsection 2.2.1 and in subsection 2.2.2, lattice-based cryptography and learning with-errors problems are briefly described, to better understand CRYSTALS-Kyber main characteristics.

2.2.1 Lattice-based cryptography

Lattices were first introduced in the 19th century as regular arrangements of points in n-dimensional space. For example, in Figure 2.6 there are 2 different 2-dimensional lattices.

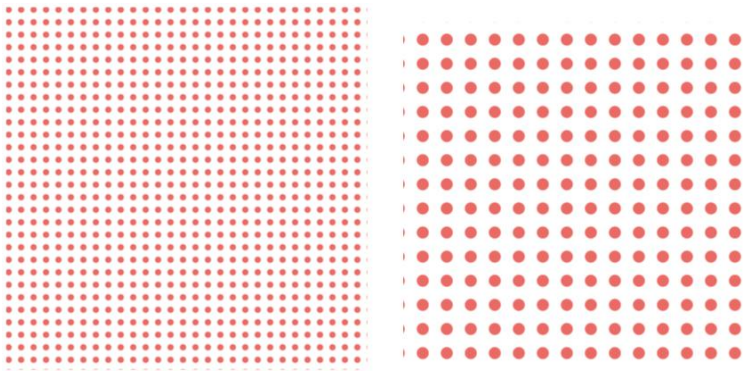

Figure 2.6: Lattice examples

Since the appearance of the lattice basis reduction algorithm, more than twenty years ago, lattices have had surprising applications in cryptography.
In the last decade their applications were only negative, since they were used to break various cryptographic schemes. Paradoxically, recently they have been selected to be one of the most promising solution in modern cryptography [7].
It is believed that lattice-based cryptography has potential to resist the attacks from quantum computers.
Obviously, lattice-based cryptography derived its name from the fact that algorithm's security is related to hard math problems around lattices.

A lattice can basically be thought of as any regularly spaced grid of points stretching out to infinity. An **integral lattice** $\mathcal{L}$ can be defined as the $\mathbb{Z}$-linear combination of n independent vectors $b_i \in \mathbb{Z}^n$.

$$\mathcal{L} = \{\sum a_i b_i \ : \ a_i \in \mathbb{Z}\}$$

Any linear combination of b_1 and b_2 , will be a point in the lattice. As an example, $\mathbb{Z}^n$ is a lattice, generated by the standard basis for $\mathbb{R}^n$.
In general, lattices would be infinitely large. However, computers does not have infinite memory to work with, and therefore we need a compact way to represent them when we use them in cryptography.
For this reason, we use what is called **a basis of a lattice**. A basis is a small collection of vectors that can be used to represent any point in the grid that forms the lattice. Crucially, **the basis for a lattice is not unique**. For instance, the vectors (8,8,8), (6,5,4) and (1,2,3) form an alternative basis for $\mathbb{R}^3$, since these points do not happen to lie on a single line going through the origin.
The idea is that by choosing a basis we have actually chosen an entire lattice and in this way, in contrast to an infinite grid of points, a basis is a simple finite element which we can represent in a memory.
Lattice-based cryptography includes the class of cryptosystems whose security is based on hard lattice problems such as Shortest Vector Problem (SVP), Shortest Independent Vectors Problem (SIVP), and the Closest Vector Problem (CVP). The most important one is the **Shortest Vector Problem**, which asks us to approximate the minimal Euclidean length of a non-zero lattice vector. In particular, given a linearly independent basis $B = \{\overrightarrow{b_1}, \overrightarrow{b_2}, ..., \overrightarrow{b_n}\} \in \mathbb{Z}^{m \times n}$, find a non-zero vector $\overrightarrow{v}$ such that:

$$\|\overrightarrow{v}\| = \min_{\overrightarrow{z} \in B} \|\overrightarrow{z}\|$$

The researches on solving the SVP play an important role in cryptography.
For instance, when a lattice-based cryptosystem is built, one of the most appropriate security parameters that can be derived is related to the time/space complexity of the best algorithm in solving the SVP. [8]
This problem is thought to be hard to solve efficiently, also with a quantum computer. At first glance, it might seem a relatively easy problem, but we need to remember that the basis we are given are made of long vectors. When it comes to cryptography, we are dealing with much higher dimension with respect to the example shown in Figure 2.6. Therefore, finding a combination of the basis vectors that simultaneously makes all coordinates small turns out to be quite hard, even with a quantum computer.

2.2.2 Learning with error problem

Learning with error problem (**LWE**) is a lattice-based problem that offer the better efficiency in terms of speed and key and ciphertext sizes. Its main claim to fame is being as hard as worst-case lattice problems, rendering all cryptographic constructions based on it secure under the assumption that worst-case lattice problems are hard. [9]
Learning with errors is a method defined by Oded Regev in 2005. It involves the difficulty of finding the values which solve:

$$A \cdot s + e = B$$

If not for the error e, finding s would be easy: after approximately n equations, s can be recovered in polynomial time using Gaussian elimination. Introducing the error makes the problem significantly more difficult. For instance, the Gaussian elimination algorithm takes linear combinations of n equations, thereby amplifying the error to unmanageable levels, leaving essentially no information in the resulting equations. The problem is described more precisely in [9].
Generally speaking, cryptosystem is parameterized by integers n (the security parameter), m (number of equations), q (modulus), and a real $\alpha > 0$ (noise parameter).
Now, let's see an example.
Imagine Alice and Bob want to communicate: they need to generate a secret key between them. Alice first compute $A \cdot s + e$. A will be random ($A \in \mathbb{Z}_q^{m \times n}$), and s and e will have small coefficients, between [-1,0,1] ($s \in \mathbb{Z}_q^{n \times 1}$ and $e \in \mathbb{Z}_q^{n \times 1}$).

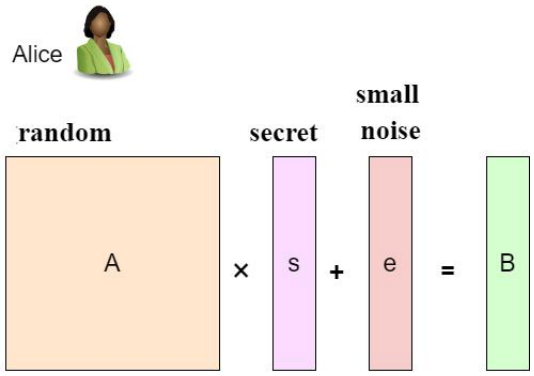

Figure 2.7: Public-key encryption scheme: key generation

(A, B) will be the public key for Alice, publishing it for other parties. s will be her own secret key, know only by her.
Then, Bob will ask for Alice public key. Imagine he has a single bit message M. He will compute:

$$u = \sum A_{samples}$$

$$v = \sum B_{samples} - \frac{q}{2}M$$

u and v are the encrypted values and the message M will becomes their shared secret key. Alice, to decrypt (u, v), must calculate:

$$Dec = v - su(mod\ q)$$

If Dec is less than $\frac{q}{2}$, the message is 0, while if is greater than $\frac{q}{2}$, the message is 1. If Alice used the cyphertext of u and v, and compute $v - su$, since s is her own secret key, she can decrypt the message M. This approach can be used to send the message from Bob to Alice, and once they have the same message, they can create the common secret key and communicate one with each other. [10]

Cryptographic schemes based on the LWE problems generally require large key sizes, of the order of n^2. Reducing this to almost linear size is highly desirable. One natural way to achieve this goal is to assume that there is some structure in the LWE samples. [9] This is defined as **Ring-Learning with Errors** problem (RLWE). More specifically, replacing the group $\mathbb{Z}_q^n$ with the ring:

$$\mathbb{Z}_q[x]/\langle x^n + 1\rangle$$

It has been demonstrated that RLWE is also at least as hard as worst-case lattice problems over special classes of ideal lattices and cryptographic applications of RLWE generally enjoy an increase in efficiency compared with those of LWE. [11] LWE is at least as hard as standard worst-case problems on Euclidean lattices, while RLWE is as hard as their restrictions to special classes of ideal lattices.

Finally, **Module Learning with Errors problem** (MLWE) has been proposed to address shortcomings in both plain LWE and RLWE by interpolating between the two, obtaining more complicated algebraic structures than ideal lattices. MLWE might be able to offer a better level of security than RLWE and still have performance advantages over plain LWE. It represents a trade-off between the two extremes. In the specific case of the Module-LWE parameters used in CRYSTALS-Kyber, there is a reduced structure compared to Ring-LWE, getting a much better scalability, and - when encrypting messages of a fixed size of 256 bits - performance very similar to Ring-LWE-based schemes.

Chapter 3

CRYSTALS-Kyber

In this chapter, we will provide a description of the main modules of CRYSTALS-Kyber algorithm. Before that, a general background on the object considered in the NIST competition is needed, focusing on the difference between KEX, PKE and KEMs.
As discussed in previous sections, NIST competition was initiated to find replacements for the public key primitives that are not quantum-resistant.
In the case of primitives that provide authenticity, this means digital signature schemes. But in the case of primitives that provide secrecy, the solution is not that clear. The discussed options were:

- **Key exchange** (KEX): is a protocol that runs between two parties. At the end of an execution, a key exchange protocol outputs the same session key at both parties.
- **Public Key Encryption** (PKE): it consists of three algorithms:
  - The **key generation algorithm** generates a key pair consisting of a public and a private key (*pk, sk*);
  - The **encryption algorithm** takes a message and a public key to compute a ciphertext;
  - The **decryption algorithm** takes a ciphertext and a private key to compute a plain-text.

  It is required that the decryption of an encryption returns the original message if the correct private key is used.

- **Key Encapsulation Mechanism** (KEM): it consists of three algorithms, similar to PKE.
  - the **key generation algorithm** generates a key pair consisting of a public and a private key (*pk, sk*);
  - the **encapsulation algorithm**, in contrast to PKE, takes a public key to compute a session key and a ciphertext;
  - the **decapsulation algorithm** takes a ciphertext and a private key to compute a session key.

  It is required that the decapsulation of a ciphertext returns the same session key as the encapsulation that generated it if the correct private key is used.

After long discussion processes, NIST decided that they will standardize KEMs. KEX has been discarded, since all the KEX candidates required interaction and thereby are less general than PKE and KEMs. [12]
A KEM is similar to a Public Key Encryption (PKE) scheme, since both of them exploit a combination of public and private keys. The main purpose of PKE is to encrypt session keys. A KEM does essentially the same thing, but with the difference that the scheme has control over choosing the session key. The public key is used to create an *encapsulation*, giving a randomly chosen shared key, and then using the private key this *encapsulation* is decrypted.
This small difference makes it easier to construct secure schemes.

CRYSTALS-Kyber is a **KEM** (or Key Encapsulation Mechanism). It is a probabilistic algorithm that produces a random symmetric key and an encryption of that key. Instead of processing the message directly using the public and the private keys, it adds a shared secret to the process, encapsulating and decapsulating it using the public and the secret keys.
Key Derivation Function, as hash function, is used, and it is composed by the three main algorithms described above.

3.1 Description

CRYSTALS-Kyber is a **IND-CCA2-secure KEM**, based on the hardness of solving the **learning with-errors** (**LWE**) problem over module lattices.
The construction of Kyber follows a two-stage approach: first, an INDCPA-secure public-key encryption scheme encrypting messages of a fixed length of 32 bytes is introduced, which is defined as `Kyber.CPAPKE` (Chosen-Cyphertext Attack Public Key Encryption). Then, a slightly **tweaked Fujisaki–Okamoto** (FO) transform is used to construct the IND-CCA2-secure KEM (Chosen-Cyphertext Attack Key Encapsulation Mechanism). [13]
CRYSTALS-Kyber includes **three parameter sets**, as reported in 3.1, corresponding to three security levels of NIST.

Kyber Length [bytes]			
Kyber model	SECRET-KEY	PUBLIC-KEY	CIPHERTEXT
Kyber512	1632	800	768
Kyber768	2400	1184	1088
Kyber1024	3168	1568	1568

Table 3.1: Kyber Crypto Length

`Kyber.CPAPKE` is similar to the LPR encryption scheme introduced at Eurocrypt 2010 in the paper of Lyubashevsky, Palacio and Segev. [14]
The roots of this scheme go back to the first LWE-based encryption scheme presented by Regev in [15], but here, the underlying ring is not Z_q and both the secret and the error vectors have small coefficients.
The idea of using a polynomial ring goes back to the NTRU cryptosystem presented by Hoffstein, Pipher, and Silverman in [16].
The main difference is to use Module-LWE instead of Ring-LWE.

3.1.1 Parameters

`Kyber.CPAPKE` is parameterized by integers n, k, q, η_1, η_2, d_u and d_v.
n and q characterized the ring R and the ring R_q, respectively indicated as:

$$\mathbb{Z}[X]/(X^n + 1) \qquad \mathbb{Z}_q[X]/(X^n + 1)$$

where $n = 2^{n'-1}$ such that $X^n + 1$ is the $2^{n'}$-th cyclotomic polynomial. [13]
These parameters are fixed. In particular, polynomials are of the same degree $n = 256$, and the polynomial coefficients are members of the prime field $\mathbb{Z}_q$, where $q = 3329$ for all security levels. In particular, n is set to 256 because the goal is to encapsulate keys with 256 bits of entropy: smaller values of n would require to

encode multiple key bits into one polynomial coefficient, which requires lower noise levels and therefore lower security. Larger values of n would reduce the capability to easily scale security via parameter k.
q is chosen as a small prime to satisfy $n|(q-1)$. This is a requirement for the fast NTT-based multiplication. There are two smaller primes for which this property holds, namely 257 and 769. However, 3329 has been selected since the lower prime numbers that satisfy the equation cannot assure the negligible failure probability necessary for CCA security.
Despite the fact that q and n are fixed, **for each security level**, **different numbers of polynomials are required**.
These polynomials are considered as a vector whose size is specified by a parameter k. k represents the dimension of the matrix of polynomials in R_q, and fixes the lattice dimensions as a multiple of n. The values of k are 2, 3, and 4, corresponding to the security levels 1, 3, and 5, respectively. The remaining parameters η_1, η_2, d_1, and d_2 are chosen to balance between security, ciphertext size, and failure probability. [17]

Algorithm	**NIST-Level**	**k**	(η_1, η_2)	$(\mathbf{d_u}, \mathbf{d_v})$
Kyber512	1 (AES-128)	2	(3,2)	(10,3)
Kyber768	3 (AES-192)	3	(2,2)	(10,4)
Kyber1024	5 (AES-256)	4	(2,2)	(11,5)

Table 3.2: Kyber Parameters

The parameter η_1 defines the noise of s and e in key generation algorithm and of r in encryption algorithm. The parameter η_2 defines the noise of e_1 and e_2 in encryption algorithm.

3.1.2 Symmetric Functions

Kyber makes a use of a pseudo-random function (**PRF**), an extendable output function (**XOF**), two hash functions **H**, and **G**, and a key-derivation function (**KDF**). All these primitives are instantiated with function from the FIPS-202 standard, as specified in Table 3.3.

Symmetric Primitives	**Kyber**
XOF	SHAKE-128
H and G	SHA3-256 and SHA3-512
PRF(s,b)	SHAKE-256(s||b)
KDF	SHAKE-256

Table 3.3: Symmetric Primitives in Kyber

3.2 Kyber-PKE and Kyber-KEM

Kyber-PKE is an IND-CPA-secure public-key encryption scheme. It encrypts messages of a fixed length of 32 bytes. It contains three algorithms: **Key Generation**, **Encryption**, and **Decryption**.
In `Kyber-PKE Key Generation`, the polynomial matrix $\mathbf{A}$ is randomly generated, and the polynomial vectors $\mathbf{s}$ and $\mathbf{e}$ are sampled according to B_{η_1}. Then, normally, the secret key is $\mathbf{s}$ and the public key is $\mathbf{As+e}$. However, for efficient implementation purposes, the multiplication $\mathbf{As}$ is performed in NTT domain by generating $\mathbf{A}$ in NTT domain (i.e. $\hat{\mathbf{A}}$) and transforming $\mathbf{s}$ to $\hat{\mathbf{s}} = \text{NTT(s)}$. To avoid NTT^{-1} operation, $\mathbf{e}$ is also transformed to $\hat{e}$ and added to $\hat{\mathbf{A}}$ o $\hat{\mathbf{s}}$.
Therefore, the values of secret and public keys are left in NTT domain and encoded to *sk* and *pk*, respectively. In addition, the seed for randomness is appended to the public key for letting the recipient generate the matrix $\mathbf{A}$.

Algorithm 1 KYBER.CPAPKE.*KeyGen*(): key generation

Output: Secret Key $sk \in \mathcal{B}^{12 \cdot k \cdot n/8}$
Output: Public Key $pk \in \mathcal{B}^{12 \cdot k \cdot n/8+32}$

d $\leftarrow \mathcal{B}^{32}$
$(\rho, \sigma) :=$ G(d)
N:=0
for i from 0 to k-1 **do** ▷ Generated matrix $\hat{\mathbf{A}} \in \mathcal{R}_q^{k \times k}$ in NTT domain
  for j from 0 to k-1 **do**
    $\hat{\mathbf{A}}[i][j]$:=Parse(XOF(ρ, j, i)
  end for
end for
for i from 0 to k-1 **do** ▷ Samples $\mathbf{s} \in \mathcal{R}_q^k$ from $\mathcal{B}_{\eta_1}$
  $\mathbf{s}[i] := \text{CBD}_{\eta_1}(\text{PRF}(\sigma,\text{N}))$
  N:= N+1
end for
for i from 0 to k-1 **do** ▷ Samples $\mathbf{e} \in \mathcal{R}_q^k$ from $\mathcal{B}_{\eta_1}$
  $\mathbf{e}[i] := \text{CBD}_{\eta_1}(\text{PRF}(\sigma,\text{N}))$
  N:= N+1
end for
$\hat{\mathbf{s}} := \text{NTT}(\mathbf{s})$
$\hat{\mathbf{e}} := \text{NTT}(\mathbf{e})$
$\hat{\mathbf{t}} := \hat{\mathbf{A}}$ o $\hat{\mathbf{s}} + \hat{\mathbf{e}}$
$pk := (\text{Encode}_{12}(\hat{\mathbf{t}} \bmod^+ q)||\rho)$ ▷ pk:= $\mathbf{As} + \mathbf{e}$
$sk := \text{Encode}_{12}(\hat{\mathbf{s}} \bmod^+ q)$ ▷ sk:= $\mathbf{s}$
return (pk,sk)

Further details about the different functions used in Algorithm 1 are reported in CRYSTALS-Kyber specifications. [13]
In `Kyber-PKE Encryption`, the message m is encrypted to the ciphertext $c = (c_1, c_2)$ by using the public key pk and random coins r. The polynomial vector t and the matrix $\mathbf{A}$ are obtained using the public key. The polynomial vector r is sampled according to B_{η_1} using r. The polynomial vector $\mathbf{e_1}$ and the polynomial e_2 are sampled according to B_{η_2} using r. Then, normally, the ciphertext $c = (c_1, c_2)$ is $(\mathbf{A^T r} + \mathbf{e_1}, \mathbf{t^T} r + e2 + m)$. However, multiplications are performed in NTT domain and then transformed to the normal domain by using NTT^{-1}. Moreover, the ciphertext is compressed and encoded.

Algorithm 2 KYBER.CPAPKE.*Enc*(pk,m,r): encryption

Input: Public Key $pk \in \mathcal{B}^{12 \cdot k \cdot n/8+32}$, Message $m \in \mathcal{B}^{32}$, Random coins $r \in \mathcal{B}^{32}$
Output: Cipher-text $c \in \mathcal{B}^{d_u \cdot k \cdot n/8 + d_v \cdot n/8}$

N:=0
$\hat{\mathbf{t}}$:= Decode_{12} (pk)
$\rho := pk + 12 \cdot k \cdot n/8$
for i from 0 to k-1 **do** ▷ Generated matrix $\hat{\mathbf{A}} \in \mathcal{R}_q^{k \times k}$ in NTT domain
  for j from 0 to k-1 **do**
    $\hat{\mathbf{A}}^T[i][j]$:=Parse(XOF(ρ, i, j)
  end for
end for
for i from 0 to k-1 **do** ▷ Samples $\mathbf{r} \in \mathcal{R}_q^k$ from $\mathcal{B}_{\eta_1}$
  $\mathbf{r}[i]$:= CBD_{η_1}(PRF(r,N))
  N:= N+1
end for
for i from 0 to k-1 **do** ▷ Samples $\mathbf{e}_1 \in \mathcal{R}_q^k$ from $\mathcal{B}_{\eta_2}$
  $\mathbf{e}_1[i]$:= CBD_{η_2}(PRF(r,N))
  N:= N+1
end for
e_2 := CBD_{η_2}(PRF(r,N)) ▷ Samples $e_2 \in \mathcal{R}_q^k$ from $\mathcal{B}_{\eta_2}$
$\hat{\mathbf{r}}$:= NTT($\mathbf{r}$)
$\mathbf{u}$:= NTT^{-1} ($\hat{\mathbf{A}}^T$ o $\hat{\mathbf{r}}$) + $\mathbf{e}_1$ ▷ $\mathbf{u} := \hat{\mathbf{A}}^T\ \mathbf{r} + \mathbf{e}_1$
v := NTT^{-1} ($\hat{\mathbf{t}}^T$ o $\hat{\mathbf{r}}$) + e_2 + Decompress_q($\text{Decode}_1(m)$,1)
c_1 := Encode_{d_u}(Compress_q($\mathbf{u}$,d_u))
c_2 := Encode_{d_v}(Compress_q(v,d_v))
return c= $c_1||c_2$

Further details about the different functions used in Algorithm 2 are reported in CRYSTALS-Kyber specifications. [13]

In `Kyber-PKE Decryption`, the polynomial vector $\mathbf{u}$ and the polynomial v are obtained from the ciphertext by decoding and decompressing.
The vector $\mathbf{s}$ is obtained from the secret key. Then, the message m is $v - \mathbf{s^T u}$. Again, the multiplications are performed in NTT domain and then transformed to the normal domain by using NTT^{-1}.

Algorithm 3 KYBER.CPAPKE.*Dec*(*sk*,*c*): decryption

Input: Secret Key $sk \in \mathcal{B}^{12 \cdot k \cdot n/8}$, Cipher-text $c \in \mathcal{B}^{d_u \cdot k \cdot n/8 + d_v \cdot n/8}$
Output: Message $m \in \mathcal{B}^{32}$

$\mathbf{u} := \text{Decompress}_q(\text{Decode}_{d_u}(c), \text{d}_u)$
$v := \text{Decompress}_q(\text{Decode}_{d_v}(c + d_u \cdot k \cdot n/8), \text{d}_v)$
$\hat{\mathbf{s}} := \text{Decode}_{12}(sk)$
$m := \text{Encode}_1(\text{Compress}_q(v - \text{NTT}^{-1}(\hat{\mathbf{s}}^T \circ \text{NTT}(\mathbf{u})), 1))$
return m

Further details about the different functions used in Algorithm 3 are reported in CRYSTALS-Kyber specifications. [13]

On the other hand, `Kyber.CCAKEM` is an IND-CCA2-secure KEM and it is built from the IND-CPA secure public-key encryption scheme, called `Kyber.CPAPKE`, by applying the Fujisaki-Okamoto transform.
It contains three steps: **Key Generation**, **Encapsulation**, and **Decapsulation**. In the first step, Alice generates the public and secret keys by using `Kyber-PKE Key Generation` algorithm, and shares her public key with Bob. In the second step, Bob encrypts the message to the ciphertext by using `Kyber-PKE Encryption` algorithm, and sends the ciphertext to Alice. He also computes the shared secret by using the message, Alice's public key, and the ciphertext. In the last step, Alice decrypts the ciphertext to the message by using `Kyber-PKE Decryption` algorithm, and then verifies whether it can be encrypted to the same ciphertext (sent by Bob) by following similar steps as Bob did by using `Kyber-PKE Encryption` algorithm. If ciphertexts match, Alice computes the shared secret by using the message, her public key, and the ciphertext. Otherwise, she computes the shared secret by using a random value and the ciphertext. [18].

A scheme of Kyber-CCA KEM is reported in Figure 3.1.

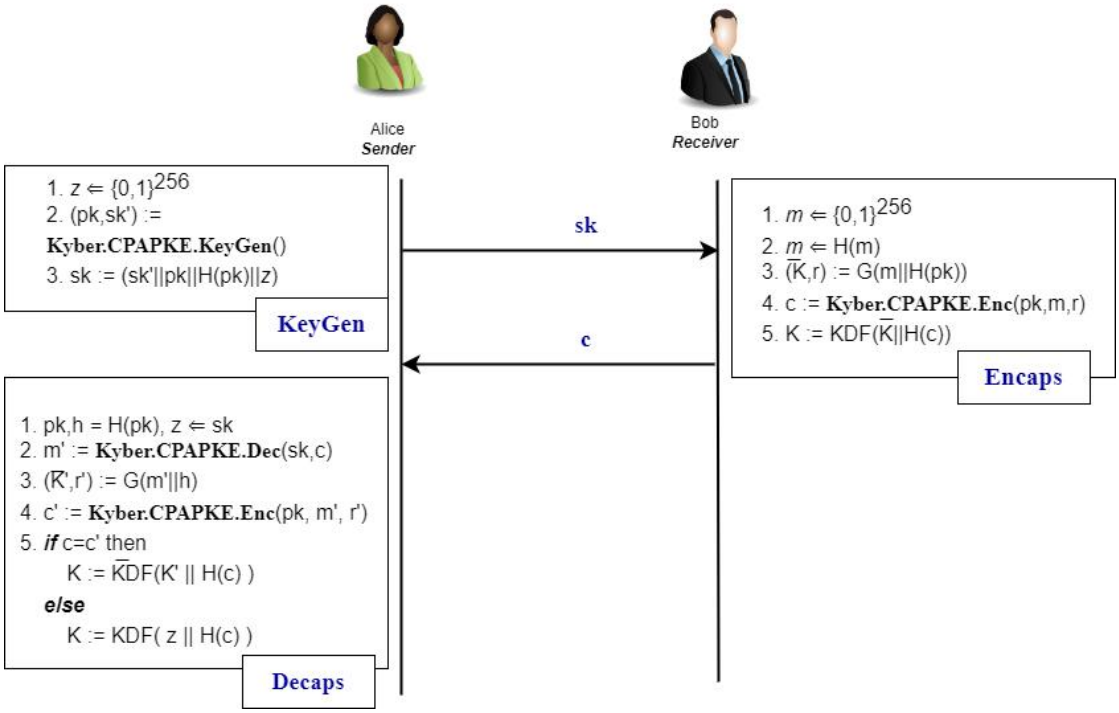

Figure 3.1: Kyber-CCA KEM

3.3 KYBER C-code Analysis

To understand which are the most expensive parts of the algorithm from the computational point of view, the C-code provided by CRYSTALS has been analyzed [1]. Compiling the test program on Linux, the executable files `test_kyber$ALG`, `test_kex$ALG` and `test_vectors$ALG` were obtained.
$ALG ranges over the parameter sets 512, 768, and 1024. The "*90s*" variant has not been considered, which is a version of CRYSTALS-Kyber where XOF and PRF functions are defined with AES-256 instead of SHAKE.
The attention is focused only to `test_kyber$ALG` test. Accordingly to the official CRYSTAL-KYBER repository, `test_kyber$ALG` tests 1000 times to generate keys, encapsulate a random key and correctly decapsulate it again. Also, the program tests that the keys cannot correctly be decapsulated using a random secret key or a ciphertext where a single random byte was randomly distorted in order to test for trivial failures of the CCA security. The program will abort with an error message and return 1 if there was an error. Otherwise it will output the key and ciphertext sizes and return 0.
The implementation in `ref/` has been considered, remembering that this is not optimized for any platform. Therefore, since it prioritises clean code, is significantly slower than a trivially optimized but still platform-independent implementation.
All the results obtained are reported in Appendix A.

Profiling `Test_Kyber`

Detailed results are reported in Appendix A.
Focusing on `test_kyber$ALG`, the algorithm has been analyzed more in details. Regardless of the level of security which is analyzed, the code is not changing. The only difference in the definition of the parameter involved, in the `param.h` file.
Obviously, the higher is the level of security considered, the higher are the dimension of the secret key, the private key and the ciphertext, as reported in Table 3.1. Therefore, the higher is the level of security, the higher is the number of calls that is done for the different main functions. This was surely predictable.
In Figure 3.2 are reported the percentages of the total running time of the program used by the main functions. The main difference that can be noticed between the three level of security, is the percentage of the total running of the program used by the different function. However, it can be clearly seen that the function computing Keccak permutation is one of the most expensive in terms of running time, despite the fact it is not the most frequently called.

[1] https://pq-crystals.org/kyber/index.shtml

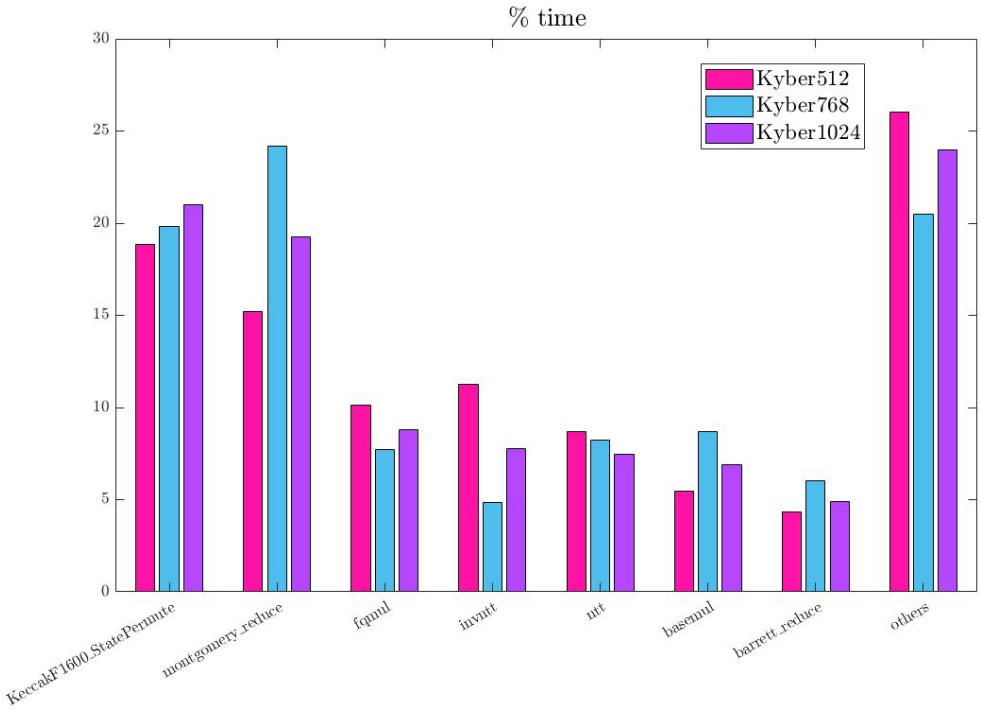

Figure 3.2: Percentage of the total running time of the program used

In particular:

- for Kyber512 and Kyber1024, `KeccakF1600_StatePermute` is the function which occupied the most of the running time. This is the core of Keccak algorithm, a family of sponge functions that has been standardized in the form of SHAKE128 and SHAKE256 extendable output functions and of SHA3-256 to SHA3-512 hash functions in FIPS-202.
- for Kyber768 the function which occupied the test most of the time is `Montgomery_reduction`. However, as for Kyber512 and Kyber1024, `KeccakF1600_StatePermute` is called a number of times which is two order of magnitude less with respect to `Montgomery_reduction`.

For all the three levels of security, **Keccak, NTT and INVNTT are the most consuming functions**. They are between the first position in terms of occupied time, but there are the ones that are called the lowest amount of time. This means the execution of these function is quite expensive.
In this thesis work, a dedicated hardware of SHA3 has been implementing, including a control part managing all the different primitives, and the Keccak core.

Chapter 4

SHA-3 Algorithm

The **Secure Hash Algorithms** (SHA) are a family of cryptographic hash functions published by the NIST as a U.S. Federal Information Processing Standard (FIPS). SHA-3 is the latest member of the Secure Hash Algorithm family of standards, released by NIST on August 5, 2015. SHA-3 is a subset of the broader cryptographic primitive family **Keccak**, designed by Guido Bertoni, Joan Daemen, Michaël Peeters, and Gilles Van Assche.
The SHA3-algorithm does not use the Merkle-Damgard construction, as in the case of MD5, SHA-1 or SHA-2, but rather uses the so-called sponge function. [17]
A sponge function is a simple iterated construction for building a function F with variable length input and arbitrary output length based on a fixed-length permutation f operating on a fixed number b of bits. SHA-3 is a family of sponge functions with members **Keccak[r,c]**, characterized by two parameters, **bit-rate** r and **capacity** c. Their sum determine the width of the Keccak-f permutation used in the sponge construction.

	r	c	Output length (bits)	Security level (bits)	Mbits	d
SHA3-224	1152	448	112	224	01	0x06
SHA3-256	1088	512	256	128	01	0x06
SHA3-384	832	768	384	192	01	0x06
SHA3-512	576	1024	512	256	01	0x06
SHAKE128	1344	256	unlimited	128	1111	0x1F
SHAKE256	1088	512	unlimited	256	1111	0x1F

Table 4.1: FIPS 202 standards' parameters

SHA-3 family is constituted by **four hash function** (SHA3-224, SHA3-256, SHA3-384 and SHA3-512) and two Extendable-Output Function (XOF) (SHAKE-128 and SHAKE-256). Keccak team proposed the **Keccak [1600]** with different r and c values for each desired length of hash output.

4.1 The sponge construction

First, all the bits of the state are initialized to zero. The input message is padded and cut into blocks of r bits. Then, sponge construction proceeds in two steps: the **absorbing phase** and the **squeezing phase**.

- In the absorbing phase, the r-bit input message blocks are XOR-ed into the first r bits of the state, interleaved with applications of the function f. When all message blocks are processed, the sponge construction switches to the squeezing phase.
- In the squeezing phase, the first r bits of the state are returned as output blocks, interleaved with applications of the function f. The number of output blocks is chosen at will by the user. The last c bits of the state are never directly affected by the input blocks and are never output during the squeezing phase

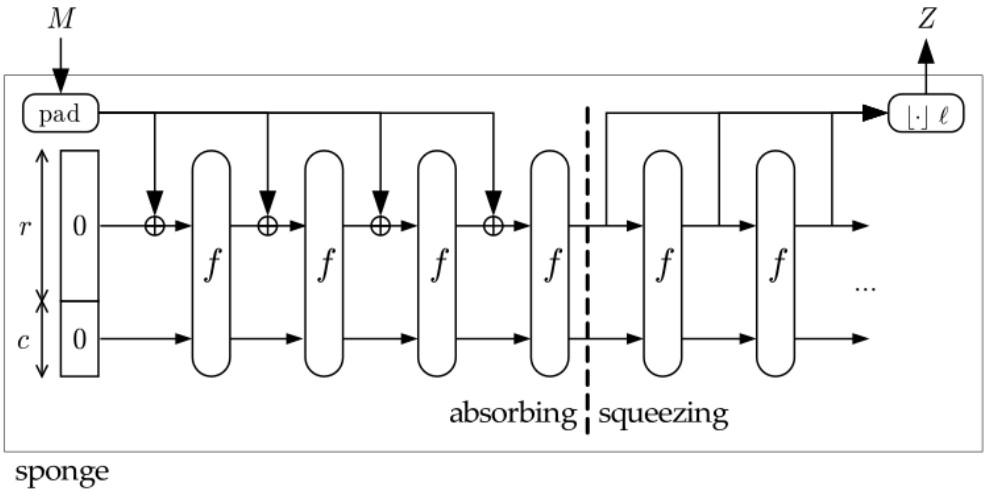

Figure 4.1: Sponge Function

In Algorithm 4 is presented the pseudo-code for the `Keccak[r,c]` sponge function, with parameters capacity c and bit-rate r. We assume for simplicity that r is a multiple of the lane size.
The description below assumes that the input `M` is represented as a string of bytes Mbytes followed by a number (possibly zero, at most 7) of trailing bits Mbits. The standard instances typically add a few trailing bits for domain separation. When made of bytes, the input of these functions then becomes Mbytes, while Mbits is solely determined by the instance used, see Table 4.1 [[1]].

Algorithm 4 Pseudo-code description of the sponge functions

```
    procedure KECCAK[R,C](M)
        ▷ Padding
3:      d = 2^|Mbits| + sum for i=0..|Mbits|-1 of 2^iMbits[i]
        P = Mbytes || d || 0x00 || ... || 0x00
        P = P ⊕ ( 0x00 || ... || 0x00 || 0x80)
6:      ▷ Initialization
        S[x,y] = 0                                  ▷ ∀ (x,y) in (0...4, 0...4)
        ▷ Absorbing phase
9:      for each block P_i in P do
            S[x,y] = S[x,y] xor P_i[x+5*y]          ▷ ∀ (x,y) such that x+5*y < r/w
            S = Keccak-f[r+c](S)
12:     end for
        ▷ Squeezing phase
        Z = empty string
15:     while output is requested
        Z = Z || S[x,y]                             ▷ ∀ (x,y) such that x+5*y < r/w
        S = Keccak-f[r+c](S)
18:     return Z
    end procedure
```

In the pseudo-code above, d is the delimited suffix, which encodes the trailing bits Mbits and its length. The padded message P is organised as an array of blocks P_i, themselves organized as arrays of lanes. The variable S denotes the state as an array of lanes. The || operator denotes the usual string concatenation.

[1] `https://keccak.team/keccak_specs_summary.html`

4.1.1 Padding

For this application, the message is byte-aligned, i.e. $len(M) = 8m$.
The total number of zero bytes that must be inserted, denoted by q, is determined as follow by m and the rate r:

$$q = \frac{r}{8} - \left(m \; mod \; \frac{r}{8}\right)$$

The padded messages that results are summarized in Table 4.2, where the notation $0x00...$ indicates the string that consists of $q - 2$ "zero" bytes.
These padding techniques are true only if $q > 2$. For each case in which a primitive is called in CRYSTALS-Kyber algorithm, this is always true.

Type of SHA-3 Function	Padded Message
Hash	M || 0x06 || 0x00 ... || 0x80
XOF	M || 0x1F || 0x00 ... || 0x80

Table 4.2: Hexadecimal form of SHA-3 padding for byte-aligned messages

4.1.2 Keccak-permutation

The 1600-bit state of Keccak [1600] consists of **5x5 matrix of 64-bit words**, as shown in Figure 4.2 ([17]). This is the width of the permutation.

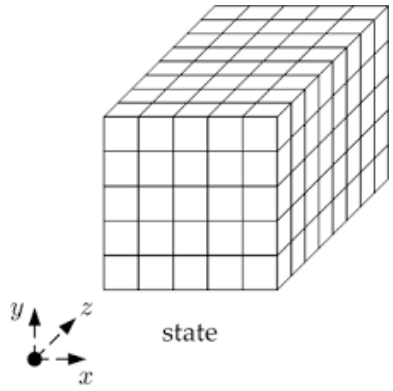

Figure 4.2: Keccak State Array

The number of round n_r depends on the permutation width, and is given by $n_r = 12 + 2l$, where $2^l = b/25$. This gives **24 rounds** for each compression step of the Keccak-f[1600].

Algorithm 5 Keccak-f[1600] (A)

```
1: procedure KECCAK-F[1600](A)
2:     ▷ A is the state matrix
3:     for i in 0 ... n_r − 1 do
4:         A=Round[1600] (A, RC [i])
5:     end for
6:     return A
7: end procedure
```

Each compression round consists of five different steps. These steps are denoted as θ, ρ, π, χ and ι. [19]. Each step of the algorithm takes a state array, denoted as A, and it returns an updated state array, denoted by A*. The different five steps can be briefly described as follow:

- θ computes the parity of each of the 5x64=320 colums, and XORs the result with two neighboring columns chosen in a regular pattern;
- ρ rotates the bits of each lane by a length, called the offset, which depends on the fixed x and y coordinates of the lane.
  Equivalently, for each bit in the lane, the z coordinate is modified by adding the offset, modulo the lane size (Table 4.3);

	X=3	X=4	X=0	X=1	X=0
Y=2	25	39	3	10	43
Y=1	55	20	36	44	6
Y=0	28	27	0	1	62
Y=4	56	14	18	2	61
Y=4	21	8	41	45	15

Table 4.3: Values r[i] constants

- π permutes the 5x5=25 64-bit words using a fixed pattern. It rearranges the positions of the lanes;
- χ combines along rows using bitwise XOR, NOT and AND operations;
- ι XORs a round constant (from Table 4.4) into one of 64-bit word of the state, in order to break the symmetry preserved by the previous steps.

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

Table 4.4: Values RC[i] constants

As explained in [20], in all the listed equations, all operations within indices are done modulo 5. `A` denotes the complete permutation state array and `A[x,y]` particular 64-bit word in that state.
`B[x,y]`, `C[x]` and `D[x]` are intermediate variables. The symbol $\oplus$ denotes the bitwise XOR, NOT the bitwise complement and AND the bitwise AND operation. Finally, ROT(W,r) the bitwise cyclic shift operation, moving the bit at position i into position i+r (modulo 64).

Algorithm 6 Round[b]

```
    procedure ROUND[B]((A,RC))
 2:     ▷ θ step
        C[x]= A[x,0] ⊕ A[x,1] ⊕ A[x,2] ⊕ A[x,3] ⊕ A[x,4] ⊕,        ▷ ∀x in 0...4
 4:     D[x]= C[x-1] ⊕ ROT(C[X+1],1),                              ▷ ∀x in 0...4
        A[x,y]=A[x,y]⊕ D[x]                          ▷ ∀ (x,y) in (0...4, 0...4)
 6:     ▷ ρ and π steps
        B[y,2x+3y]=ROT(A[x,y], r[x,y])               ▷ ∀ (x,y) in (0...4, 0...4)
 8:     ▷ χ step
        A[x,y]=B[x,y] ⊕ ((NOT B[x+1,y]) AND B[x+2,y]) ▷ ∀ (x,y) in (0...4, 0...4)
10:     ▷ ι step
        A[0,0]=A[0,0] ⊕ RC
12:     return A
    end procedure
```

The constants `RC[i]` and `r[x,y]` are cyclic shift offset and round constant respectively (Table 4.4 and Table 4.3):

Chapter 5

Hardware Implementation

This chapter presents the characteristics and architecture of SHA3 core and its control unit. It is an hardware accelerator designed for the implementation of hash and XOF functions compliant with SHA3-algorithm, but specifically dedicated to CRYSTALS-Kyber application.
With respect to Table 4.1, the primitives required by the algorithm are:

- **SHA3-256**;
- **SHA3-512**;
- **SHAKE128**;
- **SHAKE256**.

Originally, all symmetric primitives were done with only one function from the FIPS-202 standard. The authors decided to change this to different functions from the FIPS-202 family to avoid any domain-separation discussion. This modification increases code-size at most marginally, since all four functions can be obtained by a call to the same Keccak-core, with appropriate arguments and control signals.
Independenlty on the primitives selected, employing a dedicated hardware accelerators is the most efficient solution to the need for integration of cryptographic algorithm in common applications, and ensures the best performance.
SHA-3 module architecture aims to achieve the best trade-off between throughput and complexity. It has been **optimized for a single user at a time**: therefore, multiple requests are not possible, and they will be executed one by one.
Its architecture has been designed with a **top-down approach**: each block has been divided into smaller and simpler ones and then, once all of them has been tested and validated, they have been put together.
It was implemented in **VHDL language**, while testing procedure relies both on VHDL and Python code.

The architecture has been design considering parallelisms required for the 3rd level of security of CRYSTALS-Kyber: **Kyber 768**.
Table 5.1 shows all the reference parallelism used for the architecture construction.

Primitive	Function	Input Parallelism [bits]	Output Parallelism [bits]
SHA3-256	H(pk)	9472	256
SHA3-256	H(c)	8704	256
SHA3-256	H(m)	256	256
SHA3-512	G(d)	256	512
SHA3-512	G(m||H(pk))	512	512
SHA3-512	G(m'||h)	9216	512
SHAKE-128	XOF(ρ, i,j)	272	768
SHAKE-128	XOF(ρ,j,i)	272	768
SHAKE-256	PRF(σ, N)	264	1024
SHAKE-256	PRF(r, N)	264	1024
SHAKE-256	KDF(K||H(c))	512	256
SHAKE-256	KDF(z||H(c))	512	256

Table 5.1: SHA3 Inputs parallelism

As reported in Table 5.1, in the case of **SHA3-256**, the output is always fixed to 256, independently on the length of the input. The same is true for **SHA3-512**, with an output parallelism of 512.
They are respectively instantiating **H** and **G** functions.

$$H : \mathcal{B}^* \longrightarrow \mathcal{B}^{32} \qquad G : \mathcal{B}^* \longrightarrow \mathcal{B}^{32} \times \mathcal{B}^{32}$$

H inputs are taken from Table 3.1. `m` is defined as $m \leftarrow \mathcal{B}^{32}$.
By definition, $\mathcal{B}^k$ is used to denote a set of byte array of length k: therefore `m` has a parallelism of 256.
The same definition can be applied to `d`, one of the inputs of G functions.
`m||H(pk)` is the concatenation of `m`, previously commented, and `H(pk)`, whose length is fixed to 256: the final input parallelism is therefor 512.
`m'||h` is given by the concatenation of `m'`, which is the output message of `Kyber.CPAPKE.Dec` (Table 3.1), and `h`, which is defined as: $sk + 24 * k * n/8 + 32$. For the specific case of Kyber 768, this means starting from byte 2336 of 2400 of *sk*, therefore the last 64 bytes. Thus, we have the 8704 bit of the cypertext, plus 516, obtaining an input parallelism of 9216.
SHAKE128 and SHAKE256 are extendable output function (XOF) and will generate as many bits from its sponge as requested. Furthermore, SHAKE is design to work also as Pseudo Random Function (PRF) and Key Derivation Function (KDF).

In CRYSTALS-Kyber algorithm, **SHAKE128** is used as **XOF**.
With $\mathcal{B}$ the set 0,...,255 and $\mathcal{B}^*$ the set of byte arrays of arbitrary length,[13] a XOF is defined as:

$$\mathcal{B}^* \times \mathcal{B} \times \mathcal{B} \longrightarrow \mathcal{B}^*$$

XOF is always with three inputs: the seed ρ and the two indexes of the matrix, i and j. This function is called for the first time during the key generation process: here, being the seed half of the output of G, the input parallelism is 272 (being `i` and `j` both on one byte).
The time XOF is re-called, in the encryption step, ρ is defined instead as: $pk + 12 * k * n/8$. For the specific case of Kyber 768, this means starting from byte 1152 of 1184 of *pk*, thus the last 32 bytes. Therefore, we have the 256 bit of the public key, plus 16 of `i` and `j`, obtaining an input parallelism of 272.
On the other hand, **SHAKE256** is used both for **PRF** and **KDF**.

$$PRF: \ \mathcal{B}^{32} \times \mathcal{B} \longrightarrow \mathcal{B}^* \qquad KDF: \ \mathcal{B}^* \longrightarrow \mathcal{B}^*$$

As described in the NIST glossary, a **PRF** is a function that can be used to generate output from a random seed and a data variable, such that the output is computationally indistinguishable from truly random output. [21]
PRF is called first in key generation process, taking as inputs σ, which is half of G output (256 bit), and `N`. `N` is a variable used in the algorithm which is incremented from 0 a maximum amount of time equal to k-1. So, independently on the level of security of Kyber, `N` needs just an additional byte, and the overall input parallelism is 264. The output of `PRF(`σ`,N)` is the input of CBD_{η_1} function, which must be, by definition, $\mathcal{B}^{64*\eta_1}$. Being $\eta_1 = 2$, we obtain an output parallelism of 1024 bit.
The same holds true when PRF function is re-called in encryption process. Input `r` belongs by definition to $\mathcal{B}^{32}$, and `N` is the same described before. The output is again on 1024 bit, but this is due to the level of security chosen. In fact, in this case, the output of `PRF(r,N)` is the input of CBD_{η_2} function, which must be, by definition, $\mathcal{B}^{64*\eta_2}$. Being $\eta_1 = \eta_2$, the output parallelism is the same, but this is valid only for Kyber 768 and Kyber 1024.
KDF is called only when constructing `KYBER.CCAKEM`.
Twice taking in input 512 bit, given by the concatenation between the output of an H function (256 bit) and the output of a G function halved (256 bit).
The third times is re-called, in the decryption process, it takes in input the output of an H function (256 bit) and `z`. The latter one is defined as: $sk + 12 * k * n/8 + 64$. For the specific case of Kyber 768, this means starting from byte 2368 of 2400 of *sk*, therefore the last 32 bytes. Thus, we have the 256 bit of the secret key, plus 256, obtaining an input parallelism of 516.

5.1 SHA3 core

`SHA3 core` is the key element of our device.
This core can operate on both one-block and multi-block messages. A multi-block message is an input whose length is higher with respect to the maximum bit rate that can be processed by the specific primitive (Table 4.1).
The architecture consists mainly of:

- **Zero State**;
- **Version Selection and XORing (VSX) Module**;
- **Keccak core**;
- **Counter** and **comparators**;
- **Truncation unit**;
- **Output management** and **output divider unit**;
- **Control unit**.

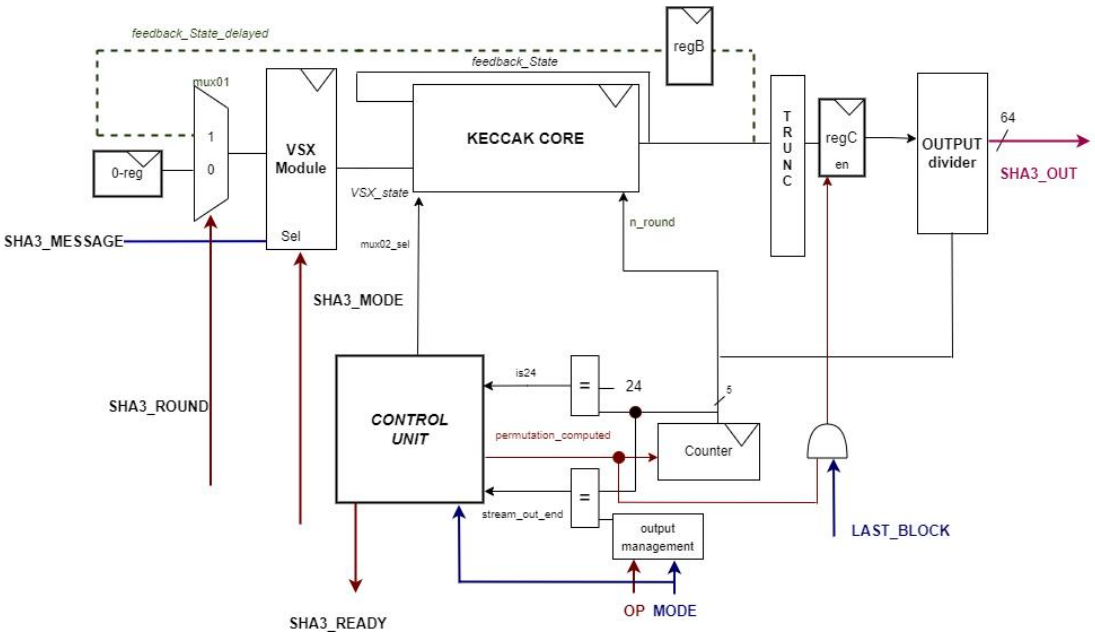

Figure 5.1: SHA3 core

The proposed architecture is gradually presented in the following subsections.

5.1.1 Zero State

The first iteration of the algorithm - independently on the primitives required - need the initial zero state, which is maintained in the component `0-reg`.
It is a 1600-bit-register and one of the input of `mux01`. This 2to1 multiplexer is used for feedback in the case a multi-block message is present. Its control signal, `SHA3_ROUND` is provided by the control unit of the top-level component.
This is equal to `0` in the case of single-block message and for the first iteration of the algorithm for multi-message block, otherwise is `1`.

5.1.2 VSX Module

This is the Version Selection and XORing Module [22].
It is made up of 1344-bit XOR for the initial storing and four concatenation blocks (`SHA3 256`, `SHA3 512`, `SHAKE128` and `SHAKE 256`) which are in charge of constructing the proper state per primitive.

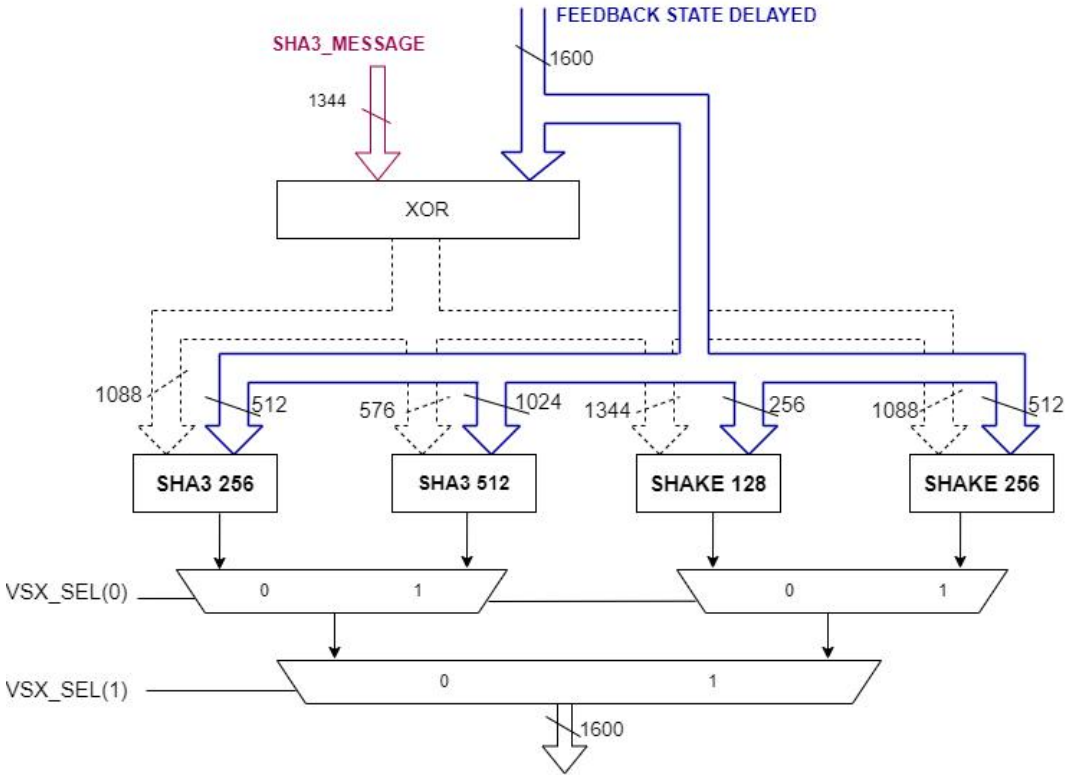

Figure 5.2: VSX Module

XOR inputs are:

- `SHA3_MESSAGE`: the message given in input to the SHA3-core. This is collected and re-organized properly by the control unit described in section 5.2. The number of XOR-operation that are actually needed depends on the primitive required: 1088 for SHA3 256 and SHAKE256, 576 for SHA3 512 and 1344 for SHAKE128;
- `FEEDBACK_STATE_DELAYED`: this is equal to the zero-state for the first iteration, and to the feedback-state delayed of one clock cycle and derived by Keccak-core in the case of multi-block messages.

Version selection is performed only after the XORing operation. Independently on the primitive required, 1344 evaluation will be processed, but this allow to instantiate a lower number of XOR-gate with respect to the case in which the selection is performed first.
The four different blocks shown in Figure 5.2 have been added just to simplify the comprehension of `VSX` module. Despite they are represented as different units, they all have the same task. There are no hardware resources that can be shared between them, because actually none of them is performing any operation at all. These block are simply taking some of the bits at the output of the XOR gate and concatenating them with some bits from the other input, `FEEDBACK STATE DELAYED`. How the state is combined depends on the primitive required, as shown in table Table 5.2. First, r-bit from XOR-gate output are taken, and then c-bit from `FEEDBACK STATE DELAYED` are concatenated.

Primitive	**rate**	**capacity**
SHA3-256	1088	512
SHA3-512	576	1024
SHAKE128	1344	256
SHAKE256	1088	512

Table 5.2: Keccak-input state construction

Then, to pass the right state to the Keccak core, a cascade of multiplexer is used. `VSX_sel` is the selector of the three multiplexers and it is equal to `SHA3_MODE`, as reported later in Table 5.7.

5.1.3 Keccak Core

To design a high-performance core, the core of Keccak provided by Keccak team has been used. [23]. As the algorithm imposes, there are five modules realizing the five algorithm's functions, i.e. Theta (θ), Rho (ρ), Pi (π), Chi (χ) and Iota (ι). The transformation rounds to be performed have been described in detail in subsection 4.1.2.

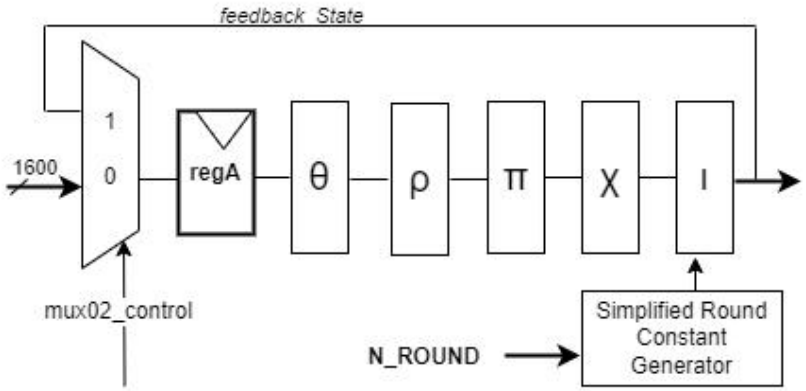

Figure 5.3: Keccak Core

The input of this unit is the output of the `VSX_Module`.
`mux02` is the input multiplexer. In the first permutation round, its selector is set to zero, so that new input state is selected. In the following rounds, it is instead set to 1. This control signal is handled by the control unit of the SHA3-core, and received by Keccak-core as input signal. The same holds true for `N_ROUNDS` signals, which is given by a **5-bit counter** outside Keccak core, described in subsection 5.1.4.

Pipelined-Keccak core

To reduce the critical path, a register (`regA` in Figure 5.3) is added.
Subpipeling has been exploited in many of the recent works about SHA-3 hash. In [24], pipeline registers are inserted after θ-operation. In this way, the longest delay in the first half of transformation rounds is made of 5 XORs, while the second part includes 2 XORs, 1 AND and 1 XOR.
In [22], to apply pipeline technique, two registers have been inserted: one between π and χ blocks, and the second at the end of the branch. By doing so, both the critical path of transformation rounds and feedback path are cut in almost half.

In our design, all the different possibilities have been analysed with **Xilinx Vivado 2022.1**, synthesizing the architecture **Xilinx Artix XC7A75-3**.
Results have been investigated in terms achievable frequency, in order to understand where is more convenient to insert the sub-pipeline register.

Placement	f [MHz]
Before θ	350
After θ	344
After ρ	328
After ρ	322
After χ	302

Table 5.3: Frequency results changing sub-pipeline register placement

The pipeline registers are inserted before θ operation.
As it can be clearly seen from the results in Table 5.3, the higher is the distance between the input of Keccak core and the sub-pipeline register, the greater the critical path and consequently the lower the maximum achievable frequency are.
The critical path has been found to be the one connecting the output of the counter shown in Figure 5.1 and the input of the inserted register, `regA`. Therefore, the nearest is the sub-pipeline register to the input, the better it is.
The reason why it has not been place in the middle of the transformation rounds is that we wanted to split the critical path not only of the Keccak core, but of the overall SHA3 core.
The results reported in Table 5.3 are related to simulations performed with the complete SHA3 structure described in Figure 5.1.
Placing the register where it is, having `regB` in the feedback and `regC` at the output, reduce the delay between the output of the feedback register and the input of Keccak core, avoiding `VSX module` from being an obstacle.

Simplified RC generator

`RC generator` is used to provide the right round constant.
As stated in subsection 4.1.2, ι-round XORs a round constant (from Table 4.4) into one of 64-bit word of the state.

Algorithm 7 ι-round

▷ ι **step**
A[0,0]=A[0,0] $\oplus$ RC

The mapping ι adds round constants in the process, with the aim of disrupting symmetry. Without this addition, round function may be subject to attacks exploiting symmetry as slide attacks, because no translation in z-direction will be made. The bits of RC are different from one round to another and are only added in a single lane of state `A`, as shown in Algorithm 7.
After a single round, disruption diffuses to all the other lanes through θ and χ.
There are different plausible implementations.
One possible implementation is having all the 24 pre-calculated round constants of 64-bits stored in a memory and transfer them to ι-module exploiting `N_ROUND` signal as multiplexers' selector. Another one is constructing a circuit using LFSR, as in [25], to perform on-the-fly generation of the RC values.
In this work, the size of the round constant generator has been reduced from being 64-bit to one byte size, simplifying the memory structure.
The simplified bits round constant values are tabulated in Table 5.4.

RC[0] 0x01	RC[12] 0x7b
RC[1] 0x32	RC[13] 0x9b
RC[2] 0xba	RC[14] 0xb9
RC[3] 0xe0	RC[15] 0xa3
RC[4] 0x3b	RC[16] 0xa2
RC[5] 0x41	RC[17] 0x90
RC[6] 0xf1	RC[18] 0x2a
RC[7] 0xa9	RC[19] 0xca
RC[8] 0x1a	RC[20] 0xf1
RC[9] 0x18	RC[21] 0x90
RC[10] 0x69	RC[22] 0xf1
RC[11] 0x4a	RC[23] 0e8

Table 5.4: Simplified round constants

This is obtained by storing only the non-zero bits in each of the round constant value. Based on the SHA-3 specification, there are only maximum number of 8 non-zero bits. This also simplifies the computation in ι, where number of logical XOR needed is reduced from 64 to 8. The bitwise XOR operation will be performed on bit 0,1,3,7,15,31 and 63 of s(0,0) respectively. [26].
With this simplification, the round constant generator requires less memory storage and less operations. The comparison between the two design has been made, synthesizing them with **Xilinx Vivado 2022.1-Xilinx Artix XC7A75-3**.

Model	LUTs	Bounded IOB	Cells
RC memory	4	69	76
Simplified RC memory	4	13	26

Table 5.5: RC[i] memory area occupation

As shown in Table 5.5, with the simplified version a lower number of IOB (Input/Output Buffer) and cells is used.

5.1.4 Counter and comparators

`Counter` is an 5-bit counter instantiated in SHA3-core. It is a synchronous counter, which counts sequentially on every clock pulse.
This counter has three different purposes:

- distinguish the different rounds of Keccak permutations. It must count from 0 to 24 to determine the flow of the different permutation stages. In the case of multi-block messages, each time a cycle ends, the counter must reset to zero and start again.
- address to the `RC constant generator`.
- manage the `output divider` unit, so that no other signal must be added to the architecture.

The two comparators have been implemented making N-xors between the two inputs that must be compared. `COMPARATOR_EQ_OUT` goes to `1` if all bits of `DIFF_BITS` are `0`: therefore, if all input bits are equal.
The first comparator is used in order to compare counter-output with 24, to determine when one cycle of Keccak-algorithm is terminated.
The second comparator is used instead together with the output of the `output management` unit, described in subsection 5.1.6.

5.1.5 Truncation Module

The `truncation module` is used to re-organized properly the hash value at the output of Keccak core. This unit receives as input a 3 dimensional matrix of size $5 \times 5 \times 64$. Independently on the required output length, in order to obtain the proper result, this matrix is first reorganized, and then send to the output as a unique stream of bit. Since the maximum output length required is 1024 bit for SHAKE256 primitive (Table 5.1), only 1024 out of 1600 are sent to `regC`.
In this way, 576 registers are saved, since there is no need of saving that values. Moreover, in order to save power, this last register is enabled only at the end of all the permutations required by the specific primitive called.
To do that, its enable signal is given by AND-ing `permutation_computed` (which is high each time a permutation cycle is concluded) and `SHA3_last_block`. The latest signal is received as input by SHA3-core from the stream control unit described in section 5.2, and is high only when the last stream bit has been transformed by Keccak core.

5.1.6 Output management and output divider unit

The `output management` and `output divider` units are the last block used by SHA3 core and are used to unpack the result into stream of 64-bit.
In fact, as happens for the input values, output parallelism would be unbearable for the synthesis on a FPGA without the proper unpacking.
For each of the primitive required we need some extra clock cycles to prepare the output. Dividing the different output parallelism reported in Table 5.1 by the output bus width (64-bit), we obtain the results shown in Table 5.6.
Since the counter starts from zero, the value that are saved in the four registers in the `output management` unit are the ones shown in Table 5.6, minus one.

Primitive	Function	Extra Clock Cycles
SHA3-256	H(pk) - H(c) - H(m)	4
SHA3-512	G(d) - G(m||H(pk)) - G(m'||h)	8
SHAKE-128	XOF(ρ, i,j) - XOF(ρ,j,i)	12
SHAKE-256	PRF	16
SHAKE-256	KDF	4

Table 5.6: Extra Clock Cycles for output buffer

In order to properly manage this unit, optimizing area resources, we have exploited components already present in the architecture.

Thus, we are able to save some area, improving the efficiency of SHA3 core.
As introduced in subsection 5.1.4, the 5-bit counter used to keep track of Keccak permutations is also used to scan the clock cycles needed for buffering the result. To determine when the entire hash value has been transmitted, a comparator is used. As shown in Table 5.6, we have four different possibilities. Thus, four registers are used to save these values, while a set of multiplexers is implemented to properly choose between them.
Multiplexers' selectors are found starting from `OP` and `MODE` signals (Table 5.7), exploiting Karnaugh maps. The three selectors of **??** are obtained as:

$$a = \texttt{(not OP(1) and not M(1) and M(0)) or (not OP(0) and not M(1) and M(0))}$$

$$b = \texttt{not OP(1)) and (not OP(0)) and MOD(0) and MOD(1)}$$

$$x = \texttt{not MOD(1)) or OP(1) or OP(0)}$$

When `stream_out_end` is high, all the 64-bit stream of the result have been send correctly to the output. Therefore, SHA3-core has terminated its work.

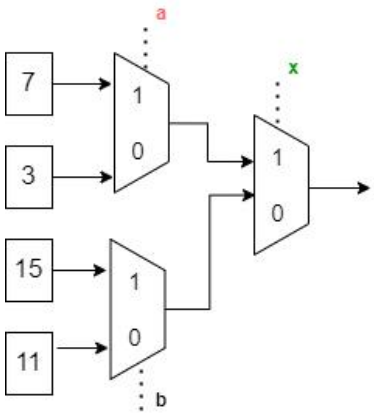

Figure 5.4: Output Management Unit

5.1.7 SHA3-core Control Unit

SHA3 core is controlled by its own control unit, realized using a Finite State Machine (FSM).

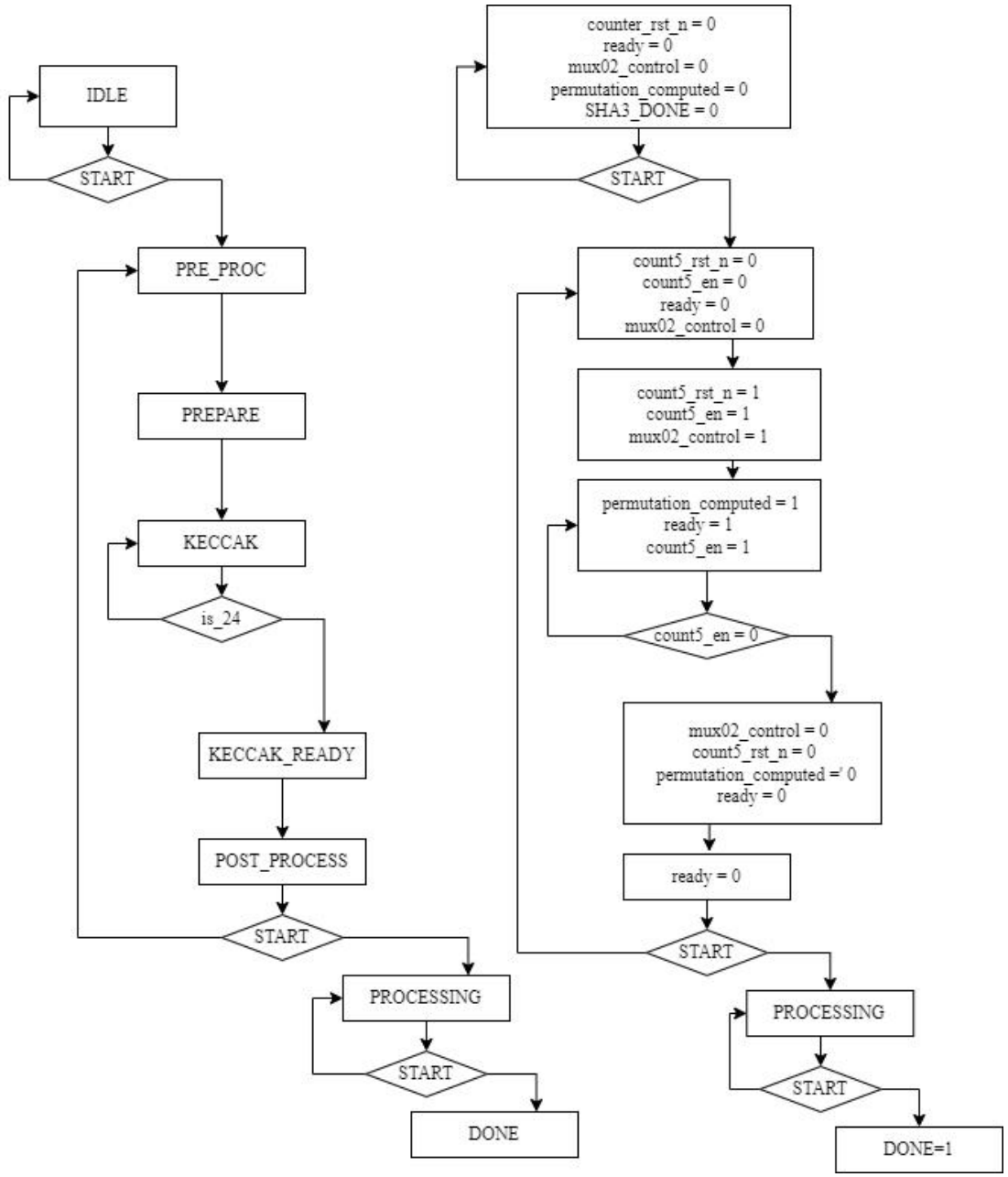

Figure 5.5: SHA3 FSM

The FSM includes 7 states, namely:
IDLE, **PRE_PROC**, **PREPARE**, **KECCAK**, **KECCAK_READY**, **POST_PROCESS**, **PROCESSING** and **DONE**.
IDLE is the starting point of the machine. Nothing happens in this state, it is just the condition in which the FSM is stalled until the start signal rises.
PRE_PROC and PREPARE states are used for version selection and initial XORing. Two states are necessary to properly synchronize the core with data coming from components of the control unit (defined in section 5.2).
Moreover, when multi-block message are required, a register has been added in the feedback path in order to decrease the critical path. An additional state before starting Keccak permutation is therefore needed to be sure the input state is the one coming from the feedback.
KECCAK state perform the main computations. It last 23 clock cycles, incrementing each time counter output.
KECCAK_READY performs the last permutation required by Keccak algorithm is performed. Here, the first hash value is finally ready.
POST_PROCESS state is an intermediate state, needed in the case a multi-block message is required. If so, the start signals will be high, and the FSM will start again a new permutation cycle. If not, or if we have processed all the stream of the multi-block message, the machine will going in PROCESSING state.
This state has been inserted to handle unpacking procedure.
Finally, once all the final hash result stream are sent to the output, DONE state is reached, and the machine stops working. `SHA3_DONE` signals goes high.

In Figure 5.6 timing diagram of the architecture is shown. In the example shown, SHA3-256 primitive is executed, with an input on 9472 bits. This implies 9 streams to be analyzed, and therefore 9 permutations to be executed. The acquisition unit takes less than 20 clock cycles to acquire one stream, while SHA3-core takes 27 clock cycles to process the message in input. However, SHA3-core aim is only to perform Keccak algorithm. The CU described in section 5.2 will provide at the input of the core the proper stream of bit each time one permutation cycle is over (in the case of multi-block message).
For this example, being the hash output value on 256-bit, four cycles are needed in order to unpack it.

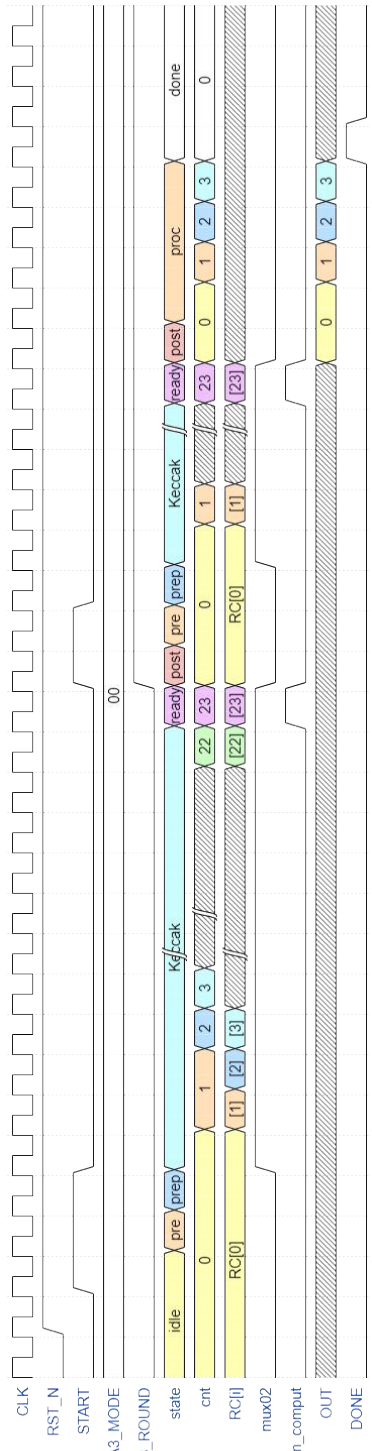

Figure 5.6: SHA-3 core timing diagram

5.2 Control Unit

The control unit is developed to synchronize the flow of data in the architecture and data communication between input and output.
In addition to the SHA3 core described in section 5.1, there are:

- `Acquisition Unit`: the first unit, whose aim is to properly handle the data in input.
- `Stream Control Unit`: used to store data until the core is ready to process a new message again. Therefore, it synchronizes the different streams of bit when there is a multi-message input to SHA3-core.
- `Padding Unit` implements the padding operations.

Each main block of the structure adopts the classical Finite State Machine with Data path (FSMD) model, with:

- a **controller**, which is in charge of organize the operations in each clock cycle.
- a **data-path**, which includes functional blocks and registers.

In Table 5.7 are reported the control signals used to select the primitive desired. `SHA3_MODE` is used to distinguish between the different primitives, while `SHA3_OP` characterize the distinct functions.

Primitive	Function	SHA3_MODE	SHA3_OP
SHA3_256	H(pk)	00	00
SHA3_256	H(c)	00	01
SHA3_256	H(m)	00	10
SHA3_512	G(d)	01	00
SHA3_512	G(m||H(pk))	01	01
SHA3_512	G(m'||h)	01	10
SHAKE_128	XOF(ρ, i,j)	10	00
SHAKE_128	XOF(ρ,j,i)	10	00
SHAKE_256	PRF(σ, N)	11	00
SHAKE_256	PRF(r, N)	11	00
SHAKE_256	KDF(K||H(c))	11	01
SHAKE_256	KDF(z||H(c))	11	01

Table 5.7: SHA3 Control Signals

They are mainly used to address the different ROMs present in the architecture. Their aim is to handle data acquisition and maintenance all over the structure, determining when:

- the acquisition of single stream is over (`Rate memory`);
- the acquisition of the input data is over (`Input buffer memory`);
- all the stream has been collected by the `stream control` unit;

One of the memories is instead used to handle the asynchrony present between the control and execution parts. This is the case of `multiplexer selector memory` in the stream control unit.
In fact, SHA3 core takes more clock cycles to perform Keccak permutation than the ones needed by the control unit to collect the different data streams. Therefore, for multi-block message, a ROM is exploited to send the output of the correct register to the padding unit only when SHA3 core is ready to operate again.
All the main architecture of SHA-3 are working one concurrently to the other. There is only an initial latency due to the reading of the first message in r-bit blocks. Then, for multi-message blocks, data is read in while the current data is being processed by the hash function, so that input latency only affects the first message block

In the following sections, we describe the functioning of the architecture more in details.

5.2.1 Acquisition Unit

The `acquisition unit` is the first unit to elaborate the data in input.
Nowadays, 64 bits is the common word size of computer architecture, buses, memory, and CPUs and, by extension, the software that runs on them. Taking this into account, the `acquisition unit` has been designed to receive a 64-bit input.
However, as shown in Table 5.1, input parallelisms are many times higher (even two order of magnitude more). The aim of this unit is to properly manage the different streams of bit in input and correctly save them in a buffer.
Remembering Table 4.1, the maximum rate we need to handle for each Keccak iteration is the one related to SHAKE128 primitive.
Consequently, the output buffer parallelism is set to be 1344 bit.

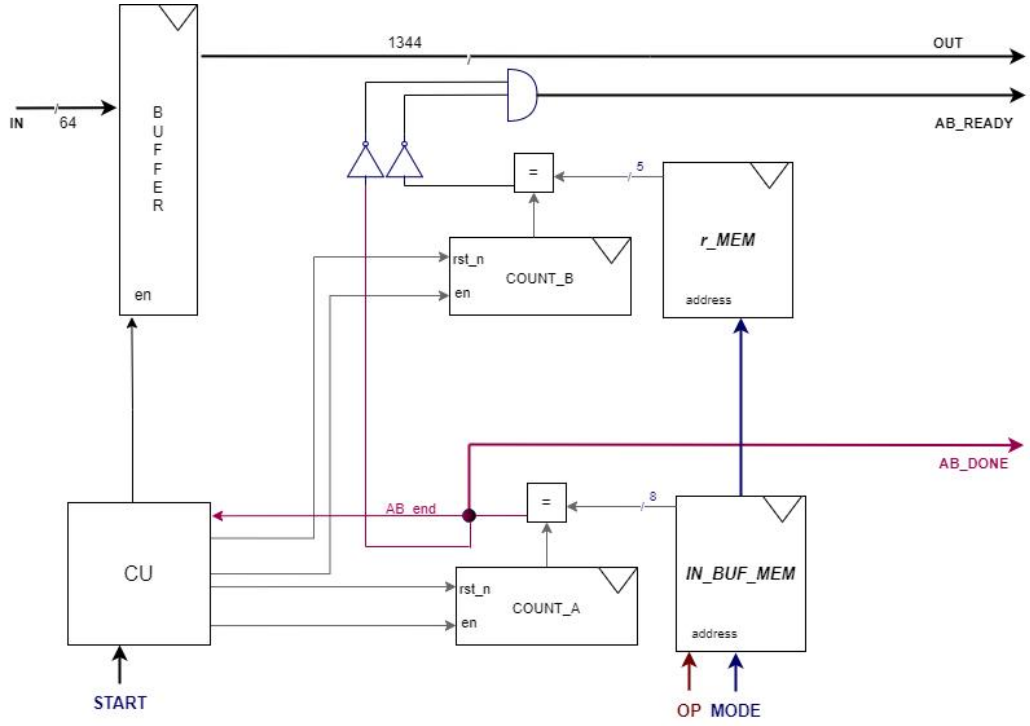

Figure 5.7: Acquisition Block

The `acquisition unit` has been design in order to work in parallel with the other units. In fact, there is no need to wait for the whole input to be collected, since accordingly to SHA3 permutation algorithm, we can process only r-bit at a time. When the buffer has collected r-bit (where r depends on the primitive that has been selected), `AB_READY` signal is set to `1`, and the r-bit stream is send to the following unit in order to be saved or processed.
When all the streams have been collected, also `AB_DONE` goes to `1`.

As well as all other major components, the `acquisition unit` exploits the FSMD model. Its FSM is reported in Figure 5.8.
The main components that handle all the signals involved are the two counters:

- `Counter A` is never reset, and it handles the input in its entirety. Its enable signal is turned off cyclically whenever counter B resets, giving time for the machine to refresh the output buffer.
- `Counter B` resets whenever the number of acquired streams equals the maximum number of streams that can be processed for the specific primitive requested. Its enable signal is never turned off; it is the reset signal that manages the component. For the last bit stream, in the case of some primitives (such as the one shown in Figure 5.9), the number of streams that must be acquired is less than the maximum number of streams that can be acquired by definition. Here, is `counter A` to manage the game, and it stops the acquisition when the last stream has been collected.

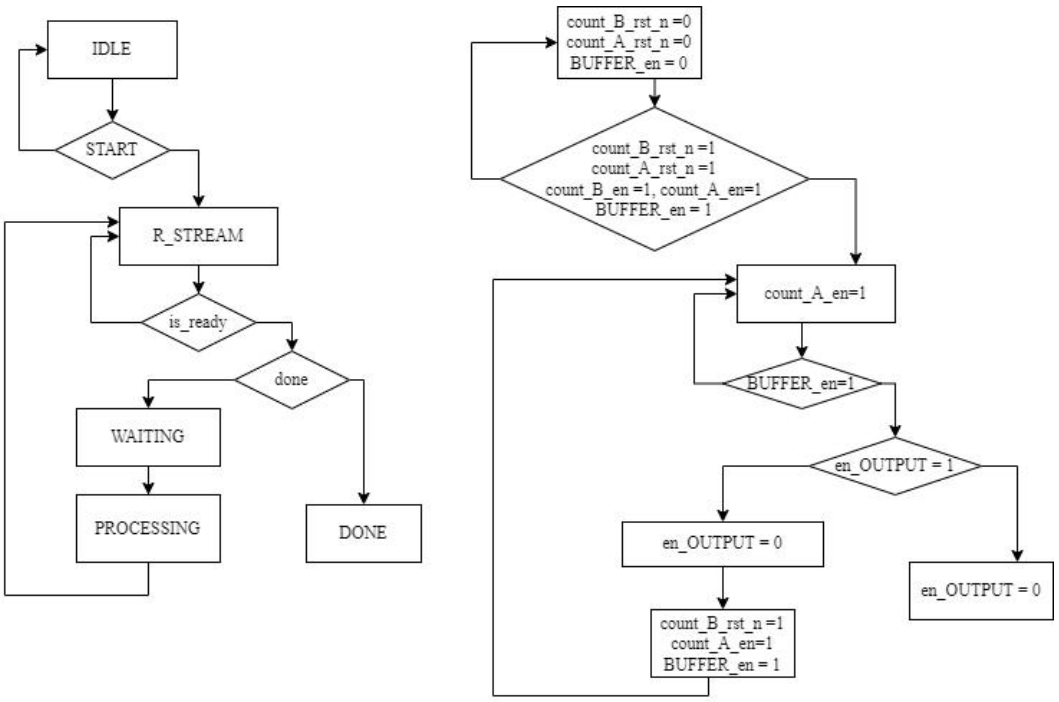

Figure 5.8: Acquisition Block FSM

In Figure 5.9 the timing diagram of the architecture is shown.

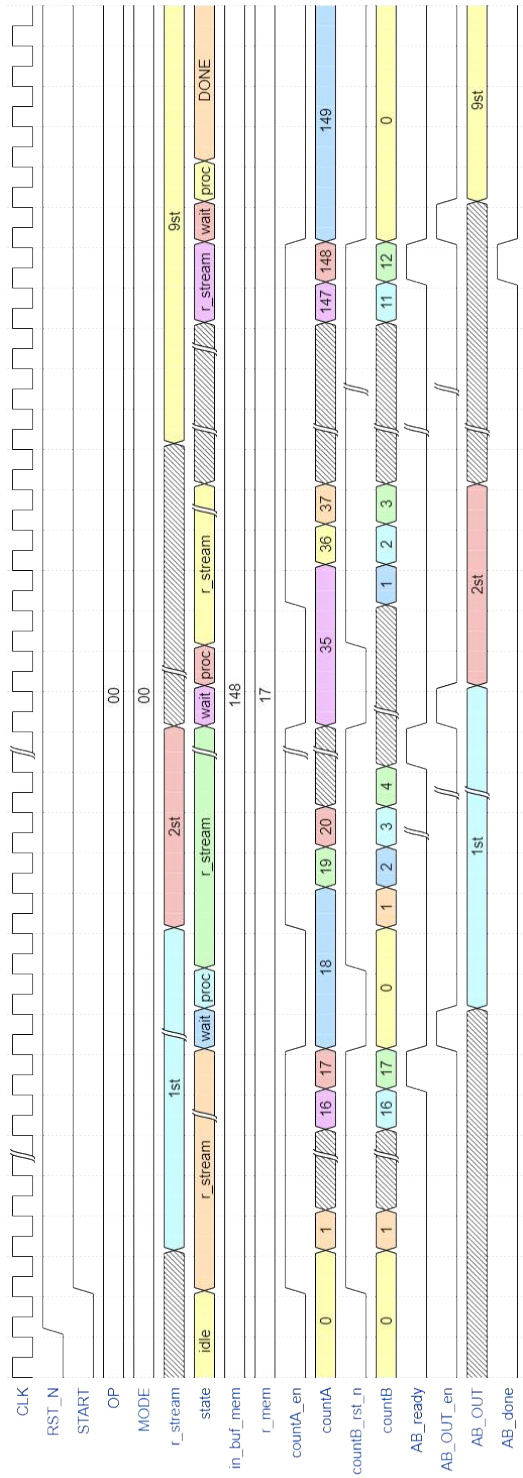

Figure 5.9: Acquisition Block timing diagram

In the example shown, where the SHA3-256 primitive is executed, the length of the analyzable stream is 1088 bits. In the last loop, however, since the input is on 9472 bits, only 768 bits remain to be acquired and grouped on the output buffer (thus only 12 streams). In this case, counter B would still continue to count up to 17 (maximum number of streams that can be acquired by the SHA3-256 primitive) but is interrupted by counter A, which signals the end of the acquisition. At this point, the signal `AB_done` goes to one and the component has finished its task.
`AB_ready` and `AB_done` control signals are both connected to output pins of the major component. Both are indispensable for managing the acquisition of input data outside the architecture. In fact, `AB_ready` lets the user know when a stream has been acquired, and allow him to momentarily interrupt sending of data. `AB_done` clarifies when it is no longer necessary to send any bit-string, as the number of streams required for the particular operation requested has been reached.

In the following subsections, a brief description of the main architecture's components is reported.

Buffer

A buffer is needed in order to properly acquire the input. We chose an input bus width, w, 64-bit, as previously stated. This has been selected in order to model a realistic communication system, showing the bandwidth limitations of any hash function. Having a large message in input m, depending on which operation and primitive are required, theoretically we would need at least m/w registers to acquire properly the input message. However, while we are acquiring the input message, we can process it in streams of r-bits (Table 4.1).
Therefore, with the proper control signals, in the buffer we can instantiate **21 64-bit registers**, being the longest stream on 1344 bit (for `SHAKE_128`).

Counter

`COUNT_A` and `COUNT_B` are two counters: the first is on 8-bit, while the second on 5-bit. These are synchronous counters: they will count sequentially on every clock pulse, and the resulting outputs count upwards from 0 to $2^N - 1$.
`COUNT_A` output is compared with `IN_BUFF_MEM` output to determine when the input acquisition is terminated, while `COUNT_B` output is compared with `r_MEM` output to understand whenever a r-bit stream is ready to be processed.

Comparator

This component has been implemented making N-xors between the two inputs that must be compared. `COMPARATOR_EQ_OUT` goes to `1` if all bits of `DIFF_BITS` are `0`: therefore, if all input bits are equal.

Rate Memory (R-Memory)

`r_mem` is a memory block which determines the number of clock cycles needed to collect r-bit. r is the rate of the SHA3-primitive, as shown in Table 4.1.
This memory is only required for the operation which need multiple permutation cycles. In all the other cases, `AB_mem` is enough, since it will stop the acquisition process once all the streams have been collected.

Primitive	Function	SHA3_MODE	SHA3_OP	Value
SHA3_256	H(pk)	00	00	17
SHA3_256	H(c)	00	01	17
SHA3_512	G(m'||h)	01	10	9

Table 5.8: Rate Memory

Input Buffer Memory

`IN_BUF_mem` is a memory block which determines the number of clock cycles needed to collect input data, accordingly to the SHA3-primitive and the corresponding operation required.

Primitive	Function	SHA3_MODE	SHA3_OP	Value
SHA3_256	H(pk)	00	00	148
SHA3_256	H(c)	00	01	137
SHA3_256	H(m)	00	10	4
SHA3_512	G(d)	01	00	4
SHA3_512	G(m||H(pk))	01	01	8
SHA3_512	G(m'||h)	01	10	145
SHAKE_128	XOF	10	00	5
SHAKE_256	PRF	11	00	5
SHAKE_256	KDF	11	01	8

Table 5.9: Acquisition Buffer Memory

The different values to be stored in the memory are found as:

$$value = m/64$$

where m is the input message length.
Considering a specific primitive, in the case the input parallelism is an integer multiple of the corresponding r, one clock cycle more must be considered.
In this case, accordingly to the algorithm definition, the last stream of bit to be processed will not contain any bit from the input, but it will be relative to the padding strings only. This happens for SHA3-256, when H(c) function is called, and for SHA3-512 in the case of G(m'||h).
For all other cases, except for SHA3-256 H(pk), the number of streams to be collected is less than the maximum number of streams that can be collected.
In these occurrences, counter B has no relevance, and the acquisition is entirely managed by counter A. For these functions, as there is only one permutation cycle to be handled, the unit of `stream control` is not necessary.
To distinguish these cases, a control signal, defined as **`one_stream_OP`**, is used. This is evaluated in the top-entity and determine when there is a one block message which can be passed directly to the padding unit.

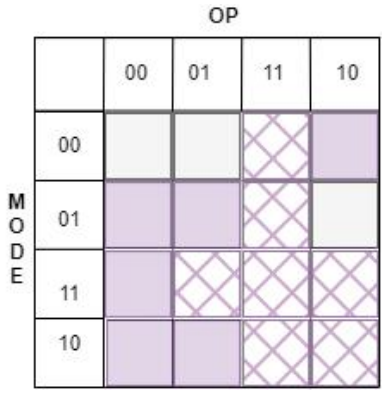

Figure 5.10: `one_stream_OP` Karnaugh map

A Karnaugh map is used. The dark cells are the one representing one-stream functions, while the striped boxes are not relevant. Different Boolean expressions can be used. The most convenient one, in terms of resource occupation, is obtained by covering all the cells (dark and striped-ones), getting a simplified expression for `one_stream_OP` control signal equal to:

$$one\ stream\ op = C + AB + \overline{A}D + A\overline{D}$$

having: A=`OP(1)`, B=`OP(0)`, C=`MODE(1)` and D=`MODE(0)`.

5.2.2 Stream Control Unit

The `stream control unit` is used in order to store the data until the SHA3 core is ready to process a new data again. Therefore, it is exploited only in the case of a multi-block messages.
As described in subsection 5.2.1, when the number of streams to be collected is less than the maximum number of stream that can be collected by definition, one permutation cycle is enough. Thus, there is no need of storing anything, because data can be send directly to the padding unit in order to be processed.
Under these circumstances, the output data from the `acquisition unit` does not pass through any registers, but through one or more multiplexers (depending on the primitive that has been requested) and then goes directly to the `padding unit`. The selectors of the different multiplexers involved in the process are obtained from `sel_mum_MEM`, described later in detail.

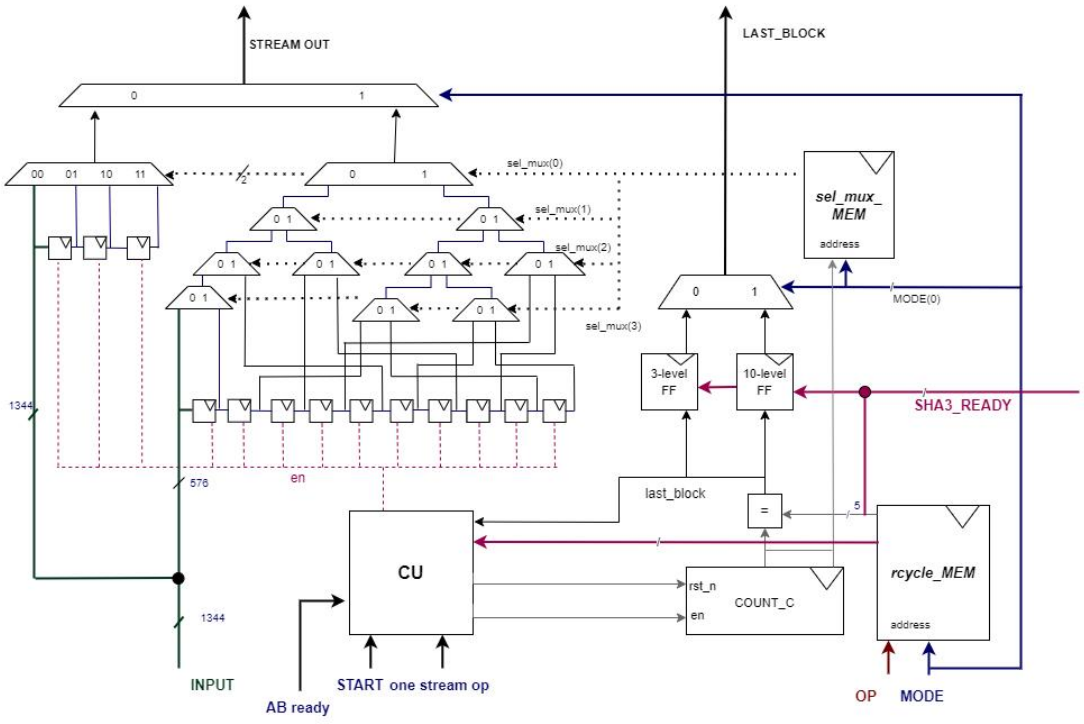

Figure 5.11: Stream Control block

The state machine that manages the component is activated at the command of `START`, which is common to each of the architectures. The unit remains in the *WAITING* state until the acquisition block has completed its first absorption.
When the `AB_ready` signal is active, then the first data stream can be saved within

the unit, and consequently the enable signal of the various registers present is also activated. When a function requires only one permutation cycle, (`one_stream_OP=1`), then the unit will simply process the data through the various multiplexers present, and conclude its work by moving into the *DONE* state.
On the other hand, when a function requires more than one permutation cycle, the machine returns to the wait state and continues to absorb and save incoming data streams from the acquisition unit. When the streams are finished, the acquisition unit stops working, having completed its task, while the stream control unit must remain active.
At this point of the simulation `last_block` is active, and we move into the second procession state. Here, each time the SHA3-core finishes a processing, the machine checks whether there are still streams to be sent, or whether all memorized streams have already been examined. In the first case, it returns to the processing state, sending the next stream to the following units. The selection of the correct stream to be sent is done by using the memory mentioned above, the output of the counter `countC` and the LSB of the signal `MODE`. If there are no more streams to be examined, the delayed version of last block signal (`last_block_del`) is active, and the `stream control` unit can finally switch off.

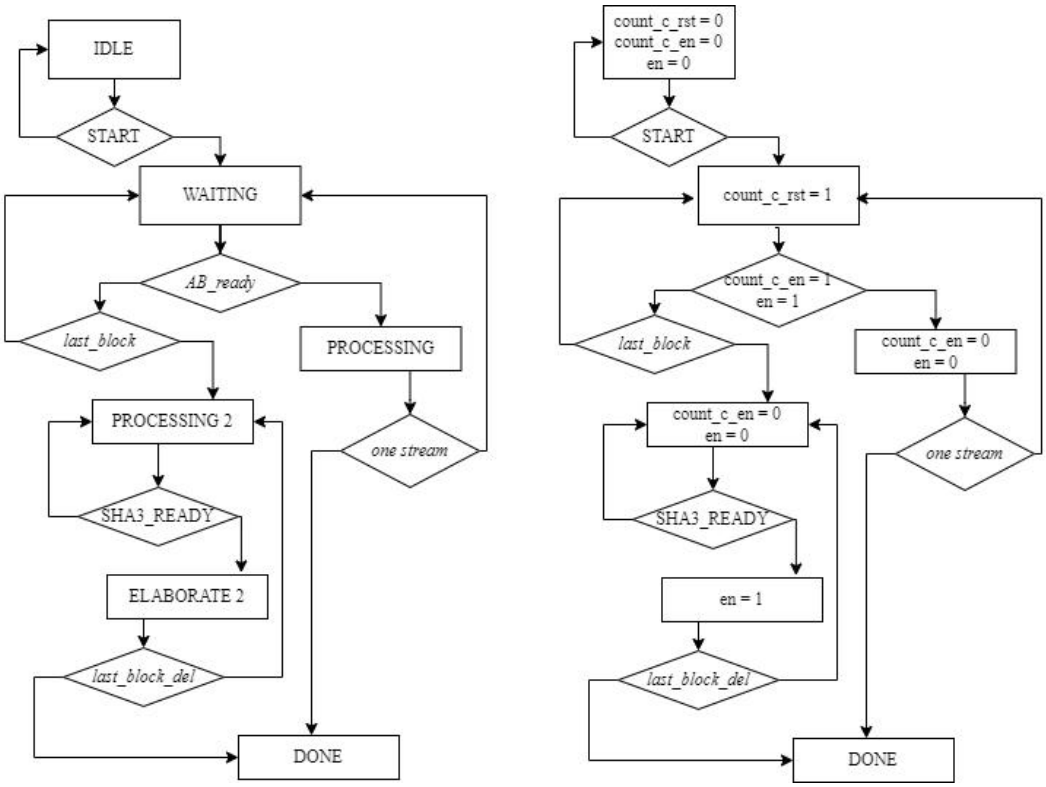

Figure 5.12: Stream Control FSM

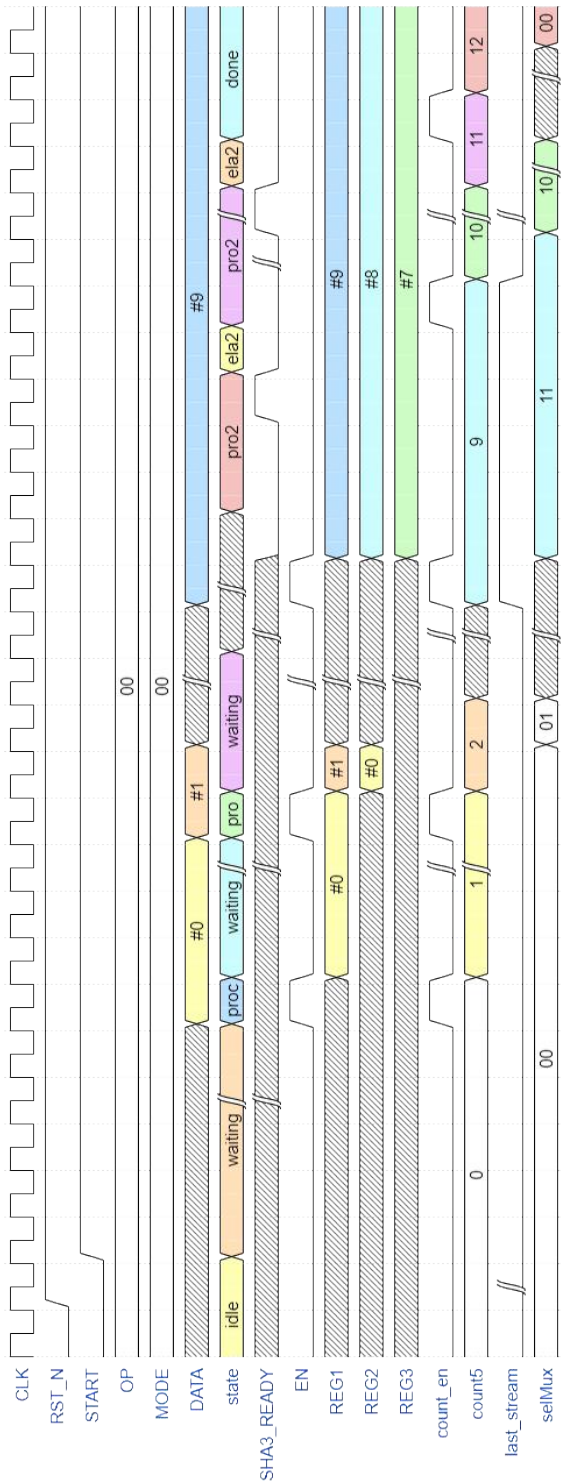

Figure 5.13: Stream Control timing diagram

In Figure 5.13 the timing diagram of the architecture is shown. In the example reported, where the SHA3-256 primitive is executed, the input length is 9472. This implies 9 streams to be analyzed: 8 of 1088 bits and the last one of 768 bits. The acquisition unit takes less than 20 clock cycles to acquire one stream, while SHA3-core takes 27 clock cycles to process the message in input.
When a function requires more than one permutation cycle, like in this case, the unit continues to absorb and save incoming data streams from the acquisition unit. When the streams are finished, all the stream of data that are needed are already saved in some of the registers present. Therefore, there is no more need to shift data from one register to the following one. The enable signal which manage all the registers remains low for the rest of the simulation. What changes is the value of the counter, `count_C`. Based on this value, each time the SHA3 core is again available to perform a new operation, multiplexers' selectors change, allowing the right stream to reach the padding unit.

Multiplexer Selector Memory (sel_mux_MEM)

`sel_mux_MEM` is a read only memory block which determines which is the next stream to be processed. The address is given by the concatenation of the LSB of `MODE` control signal and the output of `count_C`.
As shown in Figure 5.11, the memory output is connected to the different multiplexers, and guide the right stream towards the output.

ADD	Mux Sel	#reg	ADD	Mux Sel	#reg	ADD	Mux Sel	#reg
0	0000	#0	10	0010	#2	49	1101	#10
1	0000	#0	11	0001	#1	50	1001	#9
2	0001	#1	32	0000	#0	51	0111	#8
3	0001	#1	35	0001	#2	52	0110	#7
4	0010	#2	37	0010	#3	53	0101	#6
5	0010	#2	39	0011	#4	54	0100	#5
6	0010	#2	41	0100	#5	55	0011	#4
7	0010	#2	42	0101	#6	56	0010	#3
8	0011	#3	44	0110	#7	57	1000	#1
9	0011	#3	46	0111	#8	58	0110	#6

Table 5.10: Multiplexer Selector Memory

In Table 5.10 is shown the truth table which decide the content of each address of the ROM. `ADD` is the address, expressed in decimal format, and `mux_sel` is the output, which is then connected to the different multiplexers. #reg represents the register whose output is selected.
When the LSB of `MODE` control signal is `0`, we are executing either SHA3-256 or SHAKE128 primitives. SHAKE128 in CRYSTALS-Kyber applications does not require more than one permutation cycle, so in this case the data will not pass through any register. In contrast, two out of three functions of the SHA3-256 primitives handle multi-block messages, and it is therefore necessary to store some of the streams in the registers of the stream control unit. The registers used for these functions must have a parallelism of 1088. Fortunately, only three registers are needed. The outputs of the latter are collected in a 4to1 multiplexer, together with the non-delayed input, and the right data is selected thanks to the multiplexer selector memory output.
When the LSB of `MODE` control signal is `1`, we are executing either SHA3-512 or SHAKE256 primitive. For the same reason of SHAKE128 functions, also SHAKE256 will not pass through any register. On the contrary, for one of the functions of SHA3-512, ten registers are needed. The registers used for this function must have a parallelism of 576 (which is the value of r, as reported in Table 4.1).

Stream Memory (rcycle_MEM)

`rcycle_MEM` is a memory block which determines the number of stream that need to be processed. Just as with `r_mem` (Table 5.8), this is also a small memory block, since there are only three functions among those described that require more than one permutation cycle.

Primitive	Function	SHA3_MODE	SHA3_OP	Value
SHA3_256	H(pk)	00	00	9
SHA3_256	H(c)	00	01	9
SHA3_512	G(m'||h)	01	10	17

Table 5.11: Stream Memory

Once the output of `countC` is equal to the output of `rcycle_MEM`, all the streams have been properly saved. The output of the comparator is `last_block` signals, previously mentioned.

5.2.3 Padding Unit

The `padding unit` is the last unit needed to properly prepare the input message for SHA3-core. First of all, independently on the fact of having single-block or multi-block messages in input, the bits are re-mapped.
Then, we need to distinguish the two different cases.

- **single-block message**: padding is added to the first and only stream that must be analyzed.
- **multi-block message**: all the streams before the last one are simply padded with zeros to reach Keccak state parallelism (1600-bit) and are passed to SHA3-core as they are. Then, padded is added to the last stream.

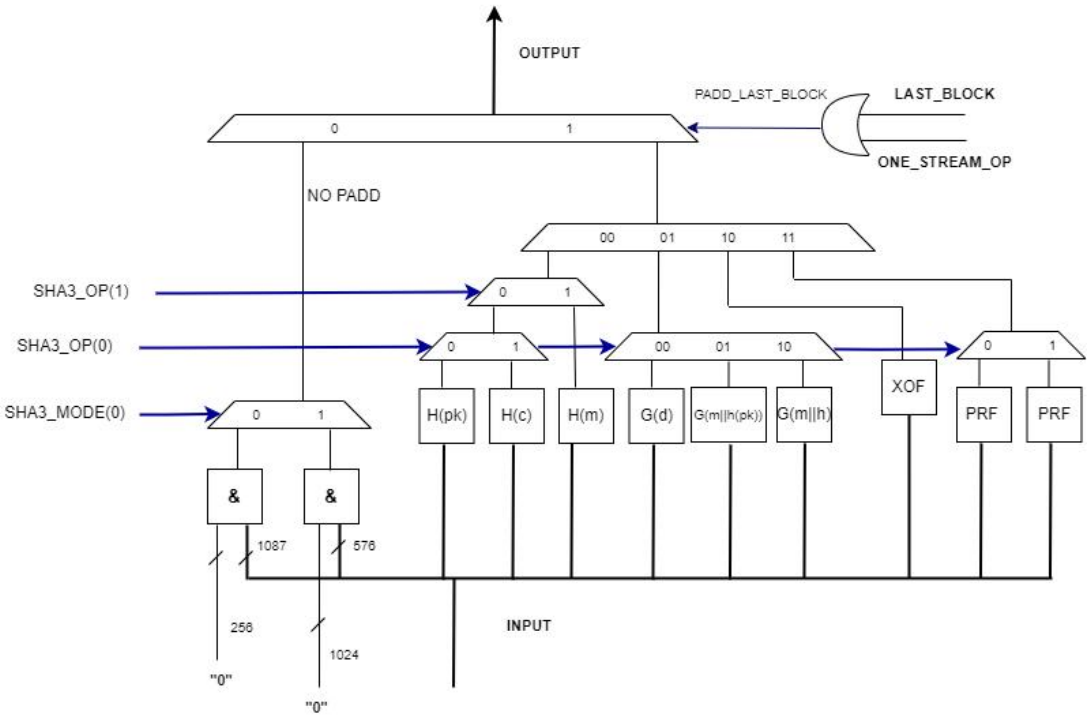

Figure 5.14: Padding Unit block

The architecture is composed by different sub-blocks. Each of these block is in charge of computing the specific padding required by the function called.
Then, exploiting a series of multiplexers, the proper padded output is selected and taken to SHA3 core.

In Figure 5.15 is reported an example of how padding operation is performed.

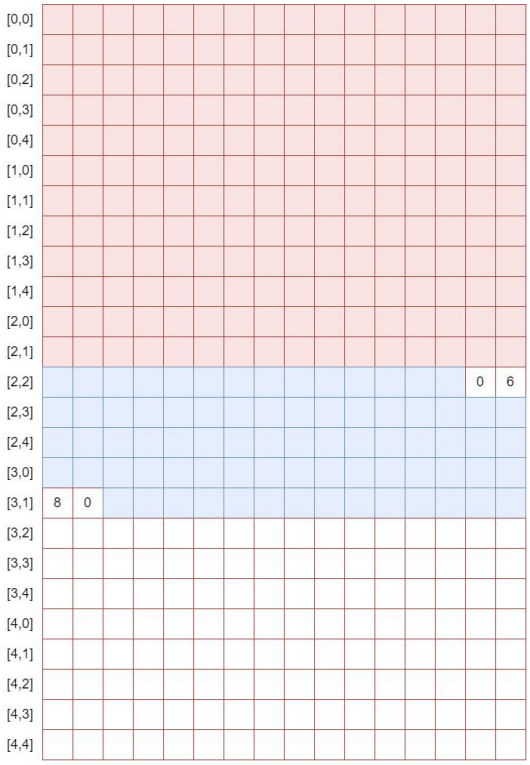

Figure 5.15: Padding first example

Here, SHA3-256 primitive is executed, with an input length of 9472 bit.
This implies 9 streams to be analyzed: 8 of 1088 bits and the last one of 768 bits.
For the first 8 rounds, no padding is performed. The padding unit's input is simply re-mapped and pass to SHA3 core.
For the last round instead, padding is performed, as shown in Figure 5.15.
The red boxes represent the 768 bits remained. Each box represent a byte. Therefore, since each lane is on 64-bit, one lane contains 16 boxes. The last stream of the input message will occupy $768 \div 64$ lanes, so from lane [0,0] to [2,1].
Then, accordingly to SHA3 specifications, `0×06` hexadecimal string must be inserted, being the function under examination an hash function. For XOFs, `0×1F` should be put, as explained in Table 4.2.
The light blue boxes represent the total number of zero bytes that must be inserted. This is denoted by q and is determined as follow:

$$q = \frac{r}{8} - \left(m \; mod \; \frac{r}{8}\right)$$

with m derived from $len(M) = 8m$, M the input length and r the bit rate.

In this example, we obtain:

$$q = \frac{1088}{8} - \left(\frac{9472}{8} \; mod \; \frac{1088}{8}\right) = 40 \rightarrow 38 \; zeros \; bytes$$

Once $q-2$ zeros bytes has been added, padding procedure is terminated adding `0×08` hexadecimal string. In a more compact form, output will be:

$$M \; || \; 0 \times 06 \; || \; 0 \times 00... \; || \; 0 \times 80$$

where the notation $0 \times 00...$ indicates the string that consists of $q-2$ "zero" bytes. Another interesting example is the one reported in Figure 5.16.

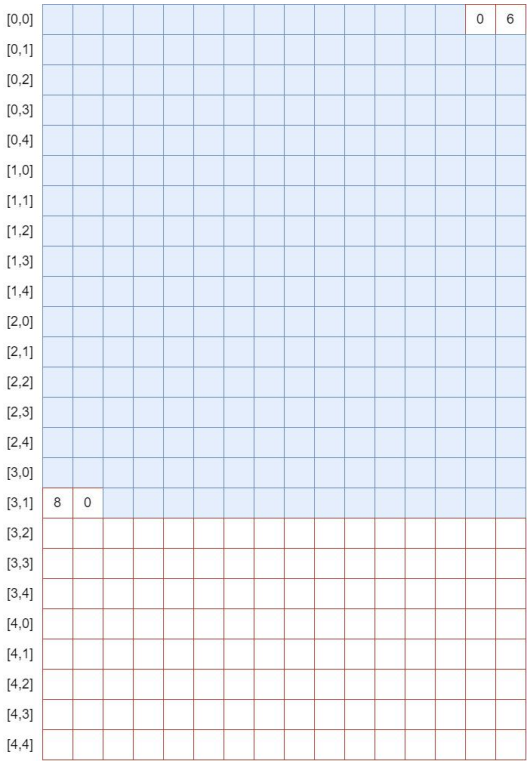

Figure 5.16: Padding second example

Here, SHA3-256 primitive is executed, with an input length of 8704 bit.
This is one of the case in which the input parallelism is an integer multiple of r.
There should be 8 streams to be analyzed, but we must consider 9.
In fact, accordingly to the algorithm definition, the last stream of bit to be processed will not contain any bit from the input, but it will be relative to the padding strings only.
Therefore, for the first 8 rounds, no padding is performed.

Then, `0×06` hexadecimal string must be inserted, followed by a number of zeros bytes equal to:

$$q = \frac{1088}{8} - \left(\frac{8704}{8}\ mod\ \frac{1088}{8}\right) = 136 \rightarrow 134\ zeros\ bytes$$

`0×08` hexadecimal string is then put. In this case, exploiting again a more compact form, output will be:

$$0 \times 06\ ||\ 0 \times 00...\ ||\ 0 \times 80$$

M is not present.

Accordingly to the rules explained in subsection 4.1.1, in Table 5.12 are reported the compact form of all the different padding procedures.

Primitive	Function	Padded expression	$q-2$ values
SHA3-256	H(pk)	$M\ ||\ 0 \times 06\ ||\ 0 \times 00...\ ||\ 0 \times 80$	38
SHA3-256	H(c)	$0 \times 06\ ||\ 0 \times 00...\ ||\ 0 \times 80$	134
SHA3-256	H(m)	$M\ ||\ 0 \times 06\ ||\ 0 \times 00...\ ||\ 0 \times 80$	102
SHA3-512	G(d)	$M\ ||\ 0 \times 06\ ||\ 0 \times 00...\ ||\ 0 \times 80$	38
SHA3-512	G(m||H(pk))	$M\ ||\ 0 \times 06\ ||\ 0 \times 00...\ ||\ 0 \times 80$	8
SHA3-512	G(m'||h)	$0 \times 06\ ||\ 0 \times 00...\ ||\ 0 \times 80$	70
SHAKE-128	XOF(ρ, i,j)	$M\ ||\ 0 \times 1F\ ||\ 0 \times 00...\ ||\ 0 \times 80$	132
SHAKE-128	XOF(ρ,j,i)	$M\ ||\ 0 \times 1F\ ||\ 0 \times 00...\ ||\ 0 \times 80$	132
SHAKE-256	PRF(σ, N)	$M\ ||\ 0 \times 1F\ ||\ 0 \times 00...\ ||\ 0 \times 80$	101
SHAKE-256	PRF(r, N)	$M\ ||\ 0 \times 1F\ ||\ 0 \times 00...\ ||\ 0 \times 80$	101
SHAKE-256	KDF(K||H(c))	$M\ ||\ 0 \times 1F\ ||\ 0 \times 00...\ ||\ 0 \times 80$	70
SHAKE-256	KDF(z||H(c))	$M\ ||\ 0 \times 1F\ ||\ 0 \times 00...\ ||\ 0 \times 80$	70

Table 5.12: Padding expressions

Chapter 6

Results and analysis

In this chapter, we provide implementation results and compare them to the counterparts. In the last decade, several research studies about hardware implementation of SHA3 hash functions have been presented in literature.
Nevertheless, we will focus on results reported in papers dealing with hardware implementations of the SHA3 core for PQC applications. The purpose of this architecture would in fact be to be integrated into a larger unit for the implementation of the PQC CRYSTALS-Kyber algorithm.
In most of the existing implementations of CRYSTALS-Kyber article, results focus on the entire hardware architecture design. In the event that the results were for the entire CRYSTALS-Kyber hardware unit, no speculation was made about area occupancy percentage or latency of the Keccak core, and only the results in terms of frequency were reported. Moreover, the frequency results with which we compare the performance of our component are actually results related to CRYSTALS-Kyber architectures. This does not imply that the frequency limit of those architectures is relative to cryptographic primitives' components. Nonetheless, as there are no references dealing with this specific component in the PQC domain, comparisons are made considering the broader view for which our component would then be integrated into a complete architecture. Therefore, comparisons provided in this chapter are still approximations. Moreover, the units that are compared do not necessarily include the same components, and in most cases, the documents provided for comparisons exclude the control unit explained in section 5.2 from their results. Furthermore, in the cryptographic community, there are discussion about the inclusion or not of the padding module in the hardware design of Keccak core, or if it must be implemented externally and not considered when analysing SHA-3 performance. In our design, the padding module is not included in the SHA3 core, but in the external part of control. Most of the documents do not specify it. Therefore, our results are here reported and compared with the others only to provide a general overview.

6.1 Simulation Results

Our proposed architecture has been simulated and verified for accuracy and functionality with valid input samples provided in [27] using the **ModelSim**.
Testing procedure relies both on VHDL and Python code.
Synopsys has been exploited to perform a first approximate synthesis on **65-nm technology**.
Then, it is synthesized with **Xilinx Vivado 2022.1** and implemented on **Xilinx Artix XC7A75-3**.
Results are investigated in terms of area, achievable frequency, throughput and efficiency. The throughput is defined as:

$$throughput = \frac{\#block\ size \times\ frequency}{\#clock\ cycles}$$

6.1.1 Modelsim Simulation

The number of clock cycles needed to complete the different CRYSTALS-Kyber-768 cryptographic functions is obtained performing Modelsim simulations.
Results are shown in Table 6.1.

Primitive	Function	Input Parallelism [bits]	**Clock Cycles**
SHA3-256	H(pk)	9472	268
SHA3-256	H(c)	8704	268
SHA3-256	H(m)	256	39
SHA3-512	G(d)	256	39
SHA3-512	G(m||H(pk))	512	43
SHA3-512	G(m'||h)	9216	476
SHAKE-128	XOF	272	40
SHAKE-256	PRF	264	40
SHAKE-256	KDF	512	43

Table 6.1: Simulation results

From the total amount of clock cycles needed, the extra clock cycles required in order to properly buffering the hash result over a 64-bit bus have not been considered (results reported Table 5.6).

The functions which requires less clock cycles is the one whose input and output are the shortest. This for surely predictable.
On the contrary, the function which requires the higher number of clock cycles is not the one whose input is the longest. As it can be seen from Table 6.1, the highest latency is achieved by SHA3-512 G(m'||h) function, and not by SHA3-256 H(pk) (whose input is 256-bit longer).
This is related to the rate of absorption of the two primitives. In the case of SHA3-256 function, by definition, we can acquire and process 1088 bit at time. This leads to 9 Keccak permutation cycles. Instead, SHA3-512 has a rate of 576 bit. This implies an higher amount of permutation cycles (17) and consequently a significant number of clock cycles more.

6.1.2 Synopsys 65nm Synthesis

The synthesis is carried out creating a clock signal with period greater than the longest critical path of the entire structure. With the proposed architecture, imposing to the clock signal a period of 3.2 ns, we obtain the following the result:

- frequency: 312.5 MHz.
- power: 29 mW, which accounts for dynamic power only.
- area: 340048 μm^2.

A first area analysis is reported in Table 6.2.

Area Group	**Area**(μm^2)
Combinational	146466
Buf/Inv	17287
Non-combinational	193582

Table 6.2: Synopsys area analysis

A more detailed area analysis for the ASIC synthesis is reported in Table 6.3. A fair comparison between resources utilization of ASIC and FPGA (reported in Table 6.6) is not possible.
In fact, the two analysis are performed in a different enviroment and considering two different measures. Just to obtain some similarity between one synthesis and the other, the percentage of Keccak core occupation has been evaluated and reported inside the parenthesis.

Name	Global cell area [µm^2]	Local cell area Combinational [µm^2]	Local cell area Non-combinational [µm^2]
SHA3 CORE	110608 (32.5%)	100.08	83.16
◇ `regB`	17904.96	624.96	17280
◇ Keccak core	41036.04	3.6	0
◇ mux02	7911.4	7911.35	0
◇ `regC`	11511.36	452.16	11059.2
◇ mux01	5568.5	5568.48	0
◇ output divider	3134.5	4.3	0
◇ counter	75.96	21.96	54
ACQUISITION BLOCK	24904	72.72	64.08
PADDING	45353.88	5.76	0
STREAM CONTROL	132229	65.16	55.44
SHA3	**340048**	**146466**	**193582**

Table 6.3: Resources utilization after synthesis

A more detailed power analysis is reported in Table 6.4.

Power Group	**Switch power** (mW)	**Int power** (mW)	**Leak power** (pW)	**Total power** (mW)
register	26.7341	0.0855	283	26.8481 (92.35%)
combinational	0.8566	1.3357	330	2.2254 (7.65%)

Table 6.4: Synopsys power analysis

The results shown in tables Table 6.2 and Table 6.4 demonstrate that the most expensive hardware component of the architecture is the sequential part.
Registers occupy a good percentage of the total area (more than half), while in terms of power they make up more than 90% of the total consumption.
This was quite deducible when observing the architecture.
In fact, the parallelisms involved within it are quite high: there are the 21 64-bit registers of the acquisition block, the 3 1088-bit registers and the 10 256-bit registers of the stream control block and all registers containing the Keccak state in the core.
This synthesis and analysis results do not take into account the interconnections and the routing of the system.

6.1.3 Vivado Synthesis and Implementation

The architecture is synthesized with **Xilinx Vivado 2022.1** and implemented on **Xilinx Artix XC7A75-TCS-G324-3**.
Artix-7 family has been chosen following NIST's recommendations.
Table 6.5 reports obtained frequency results. The result obtained post-synthesis but pre-implementation is in line with the one found with Synopsys in subsection 6.1.2. In fact, both do not take into account complexity and connections involved in the actual implementation of the architecture. When the design is actually implemented, frequency diminishes of the 30%.

	Frequency (MHz)
Post-synthesis & Pre-Implementation	357
Post-synthesis & Post-implementation	250

Table 6.5: Vivado frequency analysis

Table 6.6 reports the detailed area utilization of the main component of our architecture. Most of the resources, in particular about the 55%, is occupied by the core, and more than half of it is occupied by sequential components, as already proved and discussed subsection 6.1.2.

Name	Slice LUTs	Slice Registers	Slice	LUT as Logic
SHA3 CORE	6841	4279	1873(54.8%)	6841
◇ `regB`	1090	1600	901	1090
◇ Keccak core	5451	1600	1596	5451
◇ mux02	449	0	361	449
◇ `regC`	0	1024	254	0
◇ mux01	251	0	226	251
◇ output divider	256	0	64	256
◇ counter	21	9	13	21
ACQUISITION BLOCK	280	2198	826	280
PADDING	6	1089	654	6
STREAM CONTROL	39	24	21	39
SHA3	**9651**	**8697**	**3413**	**9651**

Table 6.6: Resources utilization after implementation

For Vivado implementation, the four memory blocks involved (described in section 5.2.1 and in section 5.2.2) have been implemented exploiting four distributed

memory generators from the IP catalog. In this way, we are sure memories design will be compatible with the FPGA chosen.

In Figure 6.1 is reported a summary of the post-implementation utilization percentage. As it can be observed, about the 70% of I/O pin is exploited. In the other cases, the resources used are only a small percentage of the total ones.

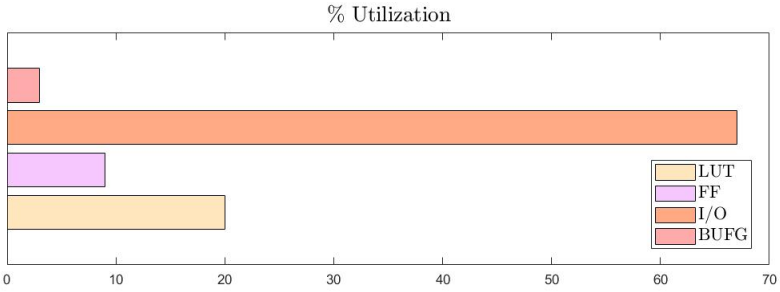

Figure 6.1: Post-implementation utilization percentage

The power analysis reported in Table 6.7 has been performed from the implemented netlist. The total on-chip power is:

$$P \ = \ 1.467 \text{ W}$$

Power	**Value [W]**	**Percentage [%]**
Dynamic	1.371	93
◇ Clocks	0.124	9
◇ Signals	0.813	59
◇ Logic	0.349	25
◇ I/O	0.085	7
Device State	0.096	7

Table 6.7: Power distribution

Now that the all the interconnections and the I/O have been finally put, as expected, the actual power of the architecture has increased with respect to the values found in 6.4.

6.2 Comparisons

In the following, we highlight the research that best connect with our.

Bisheh-Niasar *et al.* [28] propose an instruction set of components to perform polynomial sampling, NTT and point-wise multiplication to speed up lattice-based PQC. Keccak unit is configured to perform four functions: SHA3-256, SHA3-512, SHAKE-128, and SHAKE-256. Bisheh-Niasar and his team develop a dedicated buffer for interfacing with the Keccak core. As in our implementation, this has been implemented with 64-bit width and its length has been adjusted to the most extended required data, i.e. 1344-bit.

In [29] a lightweight Keccak core is presented, using 359 LUTs to perform a round of Keccak-f [1600] in 1665 cycles. This core is not used in CRYSTALS-Kyber applications, and in fact achieves one of the highest frequency. However, it has still been reported for the comparison.

Utsav Banerjee *et al.* [30] presents Sapphire, a lattice cryptography processor with configurable parameter. It is coupled with a low-power RISC-V micro-processor to demonstrate NIST Round 2 lattice-based CCA-secure key encapsulation, including CRYSTALS-Kyber, achieving up to an order of magnitude improvement in performance and energy-efficiency compared to state-of-the-art hardware implementations. Here, a low-power SHA-3 core has been implemented. A Keccak-f[1600] core inside Sapphire can be accessed standalone through RISC-V software, and is used to accelerate SHA-3 hashing and extendable output functions according to the requirements of the protocol.

In [31] a compact hardware implementation of CCA-Secure KEM CRYSTALS-Kyber is presented. This architecture offers a reasonable area-time efficiency on low hardware resources. Their design computes key-generation, encapsulation and decapsulation. However, we are interested in SHA3 core results. Their Keccak core completes one round of Keccak-f function per cycle, and one full 24-round Keccak-f function consumes 79 cycles, of which 25 are consumed in calculating hash value, the others mainly correspond to data input/output.
In our implementation, one full round Keccak-f consumes 27 cycles, 24 for the algorithm and 3 for the obtain the proper synchronization.

Guo and Li *et al.* [32] report an efficient hardware implementation of KYBER. The proposed design consumes 2253 slices in its integrity, but it is said that HASH module consumes 62% of the total resources. Therefore, approximately 1400 slices are used for the core. Our design is worst in terms of area occupation, but more efficient in terms of frequency achieved. There are no information about the amount of clock cycles required by HASH operation, therefore A×T factor cannot be compared.

In [33] PQC primitives using High-Level synthesis have been implemented and optimized. High-level synthesis produces area-optimized and speed-optimized solutions for the primitives. This is not the same procedure applied as in our design, but resource utilization for Keccak-f core are well reported and commented.

In Table 6.8 the different results related to the article just discussed are reported.

Parameter	[28]	[29]	[30]	[31]
Device	Artix-7	Virtex-6	Artix-7	Artix-7
Method	HW	HW	HW/SW	HW
LUTs	4405	359	5784	4014
FFs	1629	107	1605	1980
Slices	1825	91	1716	-
Frequency (MHz)	115	311	25	161
Total time (µs)	0.208	5.35	0.96	-
Area × Time(LUTs×µs)	916	1920	5552	-
Parameter	[32]	[33]	**Our Work**	
Device	Artix-7	Artix-7	**Artix-7**	
Method	HW	SW/HW	**HW**	
LUTs	4026	6322	**9651** (**6841**)	
FFs	1625	6993	**8697** (**4279**)	
Slices	1056	-	**3413** (**1873**)	
Frequency (MHz)	159	229	**250**	
Total time (µs)	-	0.48	**0.156/1.904**	
Area × Time(LUTs×µs)	-	3034	**1094/13271**	

Table 6.8: Comparison of Keccak-core hardware architecture in CRYSTALS-Kyber implementation

In Table 6.8, in the column related to our implementation results, the values between parenthesis represent the LUTs, FFs and slices of SHA3-core only (section 5.1), while the others are related to the complete architecture. Instead, the results on the left and right of the slash are respectively the best and worst results obtained in terms of latency (depending on the primitives which is required).

In Table 6.9 are reported the total times required by each primitive. The cases in bold represent the primitives which need the minimum and the maximum amount of time in order to properly compute the output, without considering the clock cycles needed for the output buffering (as reported in Table 6.8).

Primitive	**Function**	**Clock Cycles**	**Total Time [µs]**
SHA3-256	H(pk)	268	1.072
SHA3-256	H(c)	268	1.072
SHA3-256	H(m)	39	**0.156**
SHA3-512	G(d)	39	**0.156**
SHA3-512	G(m||H(pk))	43	0.172
SHA3-512	G(m'||h)	476	**1.904**
SHAKE-128	XOF	40	0.160
SHAKE-256	PRF	40	0.160
SHAKE-256	KDF	43	0.172

Table 6.9: Total times per primitive

Our frequency achieves a 48.9% improvement compared to other average values. It is lower with respect to [29], but a lower amount of clock cycles are needed in the execution. Thus, when single-block messages are required, there is a latency improvement (almost 19% less compared to the best time get in [28]).
On the contrary, when multi-block message are required, depending on the number of streams to be managed, latency results are more similar to the ones of the other designs. The highest delay achieved by our architecture is get in one of the case in which SHA3-512 G-function is required (1.904 µs).
However, in the counterpart implementations it not specified the maximum input length to be absorbed, and therefore the **comparison it is not truly fair**. The same holds true for resource utilization results. Apart from [28], which has the lowest area occupation factor but the worst performance, our best result is in line with the trend. Considering a single-block message, A×T is even better with respect to [29], [30] and [33]. Being all the other results mainly related to the Keccak core only, with the exclusion of the control part, results inside of round brackets have been considered (related to SHA3-core only, not to the whole SHA3 architecture).
In general, our implementation is characterized by an area×time average worsened with respect to other, but the overhead is due the amount of clock cycles required in the worst case condition mentioned above. Again, a fair comparison with the other area×time factors is not completely possible.

6.3 Potential Improvements

The architecture can be optimized in order to reduce the number of clock cycles required to compute the different hash functions and to reduce resources occupation. As noted in the previous sections, most of the area is occupied by sequential components. The most straightforward way to decrease area occupation would therefore be to eliminate, where possible, some of the inserted registers. However, this would lead to an increase in the critical paths and a consequent decrease in the maximum frequency.
Thus, it is necessary to find the right compromise to improve the architecture from the area point of view, without, however, putting too much strain on its performance.

Chapter 7

Conclusion

This thesis work presents an efficient hardware architecture for CRYSTALS-Kyber cryptographic primitives. Designing a dedicated architecture that can realize all the SHA-3 primitives used in the algorithm can significantly reduce hardware resource occupation, obtaining a component with higher performances.
Our architecture has been implemented for **Kyber-768 cryptographic primitives**. Future work can extend it to the other levels of security, modifying properly memories discussed in section 5.2 and changing consequently some components of the control unit.

These are exciting years for PQC.
There will be for sure an increase attention in post-quantum cryptography researches, motivated by NIST's standardization process.
In the future we would like to implement the entire CRYSTALS-Kyber architecture, integrating the SHA3 component discussed in this study. A lot of work is needed in order to achieve secure and invulnerable protocols and to be part of the evolution, integration and migration of cryptography systems towards quantum-safe security protocols.

Appendix A

Kyber C-code analysis

The different elements represented in the following tables are:

- **% time**: the percentage of the total running time of the program used by this function;
- **cumulative seconds**: a running sum of the number of seconds accounted for by this function and those listed above it;
- **self seconds**: the number of seconds accounted for by this function alone. This is the major sort for this listing;
- **calls**: the number of times this function was invoked, if this function is profiled, else blank;
- **self ms/call**: the average number of milliseconds spent in this function per call. If this function is profiled, else blank;
- **total ms/call**: the average number of milliseconds spent in this function and its descendents per call. If this function is profiled, else blank;
- **name**: the name of the function. This is the minor sort for this listing. The index shows the location of the function in the `gprof` listing. If the index is in parenthesis it shows where it would appear in the `gprof` listing if it were to be printed.

In Figure A.1, Figure A.2 and Figure A.3 are reported three pie graph representing the most consuming functions of CRYSTALS-Kyber, one for each of the different security level.

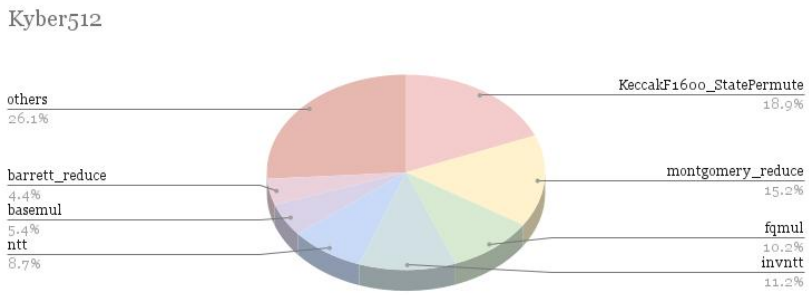

Figure A.1: Kyber-512 Analysis

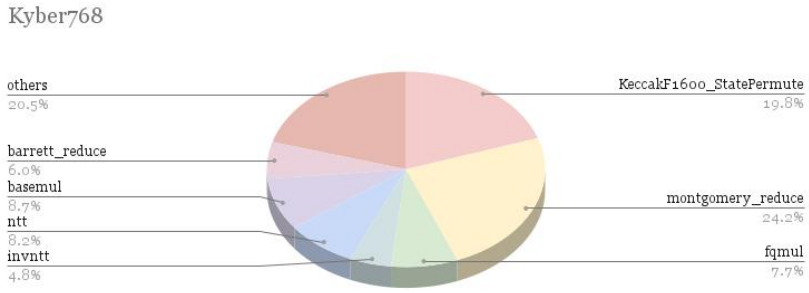

Figure A.2: Kyber-768 Analysis

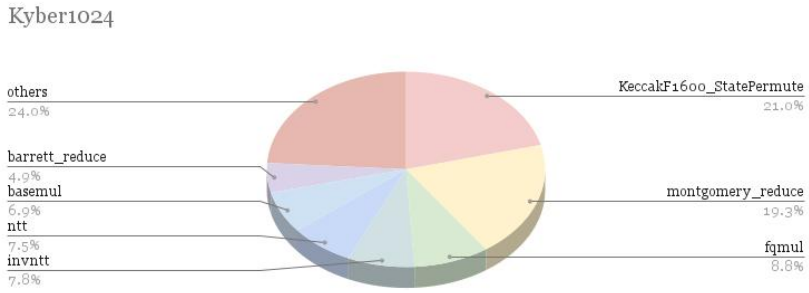

Figure A.3: Kyber-1024 Analysis

Table A.1, Table A.2 and Table A.3 present the same results in a more detailed way.

	Kyber 512					
Function	%time	cum. sec	self sec	calls	self call	tot. call
KeccakF1600 _StatePermute	18.85	0.26	0.26	264350	0.98	0.98
montgomery _reduce	15.22	0.47	0.21	87168000	0.00	0.00
invntt	11.24	0.63	0.16	21000	7.38	13.38
fqmul	10.15	0.77	0.14	85632000	0.00	0.00
ntt	8.70	0.89	0.12	30000	4.00	7.63
basemul	5.44	0.96	0.08	6912000	0.01	0.03
barrett_reduce	4.35	1.02	0.06	40320000	0.00	0.00
poly_add	4.35	1.08	0.06	57000	1.05	1.05
cbd3	3.62	1.13	0.05	24000	2.08	2.08
rej_uniform	2.90	1.17	0.04	36350	1.10	1.10
cbd2	2.17	1.20	0.03	18000	1.67	2.22
pref_poly_sub	2.17	1.23	0.03	3000	10.00	10.00
keccak_absorb_once	1.45	1.25	0.02	108000	0.19	0.82
poly_basemul _montgomery	1.45	1.27	0.02	54000	0.37	4.35
poly _cbd_eta2	1.45	1.29	0.02	18000	1.11	3.33
load64	0.72	1.30	0.01	1020000	0.01	0.01
load32 _littleendian	0.72	1.31	0.01	576000	0.02	0.02
keccak _squeezeblocks	0.72	1.32	0.01	84350	0.12	1.66
poly_reduce	0.72	1.33	0.01	84000	0.12	0.50
poly_compress	0.72	1.34	0.01	6000	1.67	1.67
poly_tomont	0.72	1.35	0.01	6000	1.67	2.28
polyvec_compress	0.72	1.36	0.01	6000	1.67	1.67
keypair	0.72	1.37	0.01	3000	3.33	104.32
poly_tomsg	0.72	1.38	0.01	3000	3.33	3.33

Table A.1: Profiling Kyber512

	Kyber 768					
Function	%time	cum. sec	self sec	calls	self call	tot. call
montgomery _reduce	24.17	0.50	0.50	142848000	0.00	0.00
KeccakF1600 _StatePermute	19.82	0.91	0.41	429727	0.95	0.95
basemul	8.70	1.09	0.18	13824000	0.01	0.04
ntt	8.22	1.26	0.17	45000	3.78	7.94
fqmul	7.73	1.42	0.16	140544000	0.00	0.00
barrett _reduce	6.04	1.55	0.13	54144000	0.00	0.00
invntt	4.83	1.65	0.10	27000	3.71	11.12
rej_uniform	4.35	1.74	0.09	81727	1.10	1.10
keccak _absorb_once	1.93	1.78	0.04	171000	0.23	0.83
polyvec _compress	1.93	1.82	0.04	6000	6.67	6.67
cbd_eta2	1.69	1.85	0.04	24000	1.46	2.29
load32 _littleendian	1.45	1.88	0.03	1920000	0.02	0.02
poly_add	1.45	1.91	0.03	111000	0.27	0.27
poly_basemul _montgomery	1.45	1.94	0.03	108000	0.28	4.92
poly_sub	1.45	1.97	0.03	3000	10.00	10.00
store64	0.97	1.99	0.02	5250267	0.00	0.00
cbd2	0.97	2.01	0.02	60000	0.33	0.83
load64	0.48	2.02	0.01	1632000	0.01	0.01
keccak _squeezeblocks	0.48	2.03	0.01	147727	0.07	1.77
keccak_squeeze	0.48	2.04	0.01	66000	0.15	1.11
poly_frombytes	0.48	2.05	0.01	27000	0.37	0.37
poly_compress	0.48	2.06	0.01	6000	1.67	1.67
verify	0.48	2.07	0.01	3000	3.33	3.33

Table A.2: Profiling Kyber768

	Kyber 1024					
Function	%time	cum. sec	self sec	calls	self call	tot. call
KeccakF1600 _StatePermute	20.99	0.73	0.73	673175	0.00	0.00
montgomery _reduce	19.26	1.40	0.67	210048000	0.00	0.00
fqmul	8.77	1.71	0.31	206976000	0.00	0.00
invntt	7.76	1.98	0.27	33000	0.01	0.02
ntt	7.47	2.24	0.26	60000	0.00	0.01
basemul	6.90	2.48	0.24	23040000	0.00	0.00
barrett _reduce	4.89	2.65	0.17	67968000	0.00	0.00
rej_uniform	4.31	2.80	0.15	145175	0.00	0.00
poly_basemul _montgomery	3.45	2.92	0.12	180000	0.00	0.00
poly_add	2.87	3.02	0.10	183000	0.00	0.00
cbd2	2.59	3.11	0.09	78000	0.00	0.00
store64	2.01	3.18	0.07	9228675	0.00	0.00
load64	1.72	3.24	0.06	2244000	0.00	0.00
keccak _absorb_once	1.72	3.30	0.06	252000	0.00	0.00
keccak _squeezeblocks	1.15	3.34	0.04	229175	0.00	0.00
load32 _littleendian	0.57	3.36	0.02	2496000	0.00	0.00
keccak_squeeze	0.57	3.38	0.02	84000	0.00	0.00
poly_sub	0.43	3.39	0.02	3000	0.01	0.01
kyber _shake128_absorb	0.29	3.40	0.01	144000	0.00	0.00
poly_frombytes	0.29	3.41	0.01	36000	0.00	0.00
poly_cbd_eta2	0.29	3.42	0.01	30000	0.00	0.00
poly_tobytes	0.29	3.43	0.01	24000	0.00	0.00
randombytes	0.29	3.44	0.01	12006	0.00	0.00
poly_tomont	0.29	3.45	0.01	12000	0.00	0.00
gen_matrix	0.29	3.46	0.01	9000	0.00	0.08
poly_compress	0.29	3.47	0.01	6000	0.00	0.00
polyvec_compress	0.29	3.48	0.01	6000	0.00	0.00

Table A.3: Profiling Kyber1024

Bibliography

[1] Kanad Basu, Deepraj Soni, Mohammed Nabeel, and Ramesh Karri. *NIST Post-Quantum Cryptography- A Hardware Evaluation Study*. Cryptology ePrint Archive, Report 2019/047. https://ia.cr/2019/047. 2019 (cit. on p. 3).

[2] asecuritysite. *Hash*. URL: https://asecuritysite.com/hash (cit. on p. 8).

[3] National Institute of Standards and Technology. *Data Encryption Standard (DES)*. FIPS Publication 46-3. Oct. 1999 (cit. on p. 9).

[4] National Institute of Standards and Technology. «Advanced Encryption Standard». In: *NIST FIPS PUB 197* (2001) (cit. on p. 9).

[5] David Ott, Christopher Peikert, and other workshop participants other workshop. *Identifying Research Challenges in Post Quantum Cryptography Migration and Cryptographic Agility*. 2019. DOI: 10.48550/ARXIV.1909.07353. URL: https://arxiv.org/abs/1909.07353 (cit. on p. 12).

[6] Summer School on real-world crypto and privacy. *Introduction to post-quantum cryptography and learning with errors*. June 11, 2018. URL: https://summerschool-croatia.cs.ru.nl/2018/slides/Introduction%20to%20post-quantum%20cryptography%20and%20learning%20with%20errors.pdf (cit. on p. 12).

[7] Phong Q. Nguyen and Jacques Stern. «The Two Faces of Lattices in Cryptology». In: *Cryptography and Lattices*. Ed. by Joseph H. Silverman. Berlin, Heidelberg: Springer Berlin Heidelberg, 2001, pp. 146–180. ISBN: 978-3-540-44670-5 (cit. on p. 13).

[8] Yu-Lun Chuang, Chun-I Fan, and Yi-Fan Tseng. «An Efficient Algorithm for the Shortest Vector Problem». In: *IEEE Access* 6 (2018), pp. 61478–61487. DOI: 10.1109/ACCESS.2018.2876401 (cit. on p. 14).

[9] Oded Regev. «The Learning with Errors Problem (Invited Survey)». In: *2010 IEEE 25th Annual Conference on Computational Complexity*. 2010, pp. 191–204. DOI: 10.1109/CCC.2010.26 (cit. on pp. 15, 16).

[10] William J Buchanan. *Learning With Errors (LWE) and Ring LWE*. https://asecuritysite.com/lattice/lwe_output. Accessed: September 24, 2022. Asecuritysite.com, 2022. URL: https://asecuritysite.com/lattice/lwe_output (cit. on p. 16).

[11] Yang Wang and Mingqiang Wang. *Module-LWE versus Ring-LWE, Revisited*. Cryptology ePrint Archive, Paper 2019/930. https://eprint.iacr.org/2019/930. 2019. URL: https://eprint.iacr.org/2019/930 (cit. on p. 16).

[12] Ward Beullens, Jan-Pieter D'Anvers, Andreas T. Hülsing, Tanja Lange, Lorenz Panny, Cyprien de Saint Guilhem, and Nigel P. Smart. *Post-Quantum Cryptography: Current state and quantum mitigation*. English. ENISA, Feb. 2021. DOI: 10.2824/92307 (cit. on p. 18).

[13] Roberto Maria Avanzi et al. «CRYSTALS-Kyber Algorithm Specifications And Supporting Documentation». In: 2017 (cit. on pp. 19, 22, 23, 35).

[14] Vadim Lyubashevsky, Adriana Palacio, and Gil Segev. *Public-Key Cryptographic Primitives Provably as Secure as Subset Sum*. Cryptology ePrint Archive, Paper 2009/576. https://eprint.iacr.org/2009/576. 2009. URL: https://eprint.iacr.org/2009/576 (cit. on p. 19).

[15] Oded Regev. «On Lattices, Learning with Errors, Random Linear Codes, and Cryptography». In: *J. ACM* 56.6 (Sept. 2009). ISSN: 0004-5411. DOI: 10.1145/1568318.1568324. URL: https://doi.org/10.1145/1568318.1568324 (cit. on p. 19).

[16] Jeffrey Hoffstein, Jill Pipher, and Joseph H. Silverman. «NTRU: A Ring-Based Public Key Cryptosystem». In: *ANTS*. Ed. by Joe Buhler. Vol. 1423. Lecture Notes in Computer Science. Springer, 1998, pp. 267–288. ISBN: 3-540-64657-4 (cit. on p. 19).

[17] Guido Bertoni, Joan Daemen, Seth Hoffert, Michaël Peeters, Gilles Van Assche, and Ronny Van Keer. *The Keccak reference*. https://keccak.noekeon.org/. 2011 (cit. on pp. 20, 27, 30).

[18] Pakize Sanal, Emrah Karagoz, Hwajeong Seo, Reza Azarderakhsh, and Mehran Mozaffari Kermani. «Kyber on ARM64: Compact Implementations of Kyber on 64-bit ARM Cortex-A Processors». In: *IACR Cryptol. ePrint Arch.* 2021 (cit. on p. 23).

[19] Kashif Latif, M. Muzaffar Rao, Athar Mahboob, and Arshad Aziz. «Novel Arithmetic Architecture for High Performance Implementation of SHA-3 Finalist Keccak on FPGA Platforms». In: *Reconfigurable Computing: Architectures, Tools and Applications*. Ed. by Oliver C. S. Choy, Ray C. C. Cheung, Peter Athanas, and Kentaro Sano. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, pp. 372–378. ISBN: 978-3-642-28365-9 (cit. on p. 31).

[20] G. Bertoni, J. Daemen, M. Peeters, and G.V Assche. *The Keccak SHA-3 Submission version 3*. URL: http://keccak.noekeon.org/Keccak-submission-3.pdf (cit. on p. 32).

[21] Peter Mell and Timothy Grance. *The NIST Definition of Cloud Computing*. en. 2011-09-28 2011. DOI: https://doi.org/10.6028/NIST.SP.800-145 (cit. on p. 35).

[22] George S. Athanasiou, George-Paris Makkas, and Georgios Theodoridis. «High throughput pipelined FPGA implementation of the new SHA-3 cryptographic hash algorithm». In: *2014 6th International Symposium on Communications, Control and Signal Processing (ISCCSP)*. 2014, pp. 538–541. DOI: 10.1109/ISCCSP.2014.6877931 (cit. on pp. 37, 39).

[23] G. Bertoni, J. Daemen, M. Peeters, G.V Assche, and R.Van Keer. *Keccak in VHDL*. URL: https://keccak.team/files/Keccak-implementation-3.2.pdf (cit. on p. 39).

[24] Hassen Mestiri, Fatma Kahri, Mouna Bedoui, Belgacem Bouallegue, and Mohsen Machhout. «High throughput pipelined hardware implementation of the KECCAK hash function». In: *2016 International Symposium on Signal, Image, Video and Communications (ISIVC)*. 2016, pp. 282–286. DOI: 10.1109/ISIVC.2016.7894001 (cit. on p. 39).

[25] Brian Baldwin, Andrew Byrne, Liang Lu, Mark Hamilton, Neil Hanley, Maire O'Neill, and William P. Marnane. «FPGA Implementations of the Round Two SHA-3 Candidates». In: *2010 International Conference on Field Programmable Logic and Applications*. 2010, pp. 400–407. DOI: 10.1109/FPL.2010.84 (cit. on p. 41).

[26] Ming Ming Wong, Jawad Haj-Yahya, Suman Sau, and Anupam Chattopadhyay. «A New High Throughput and Area Efficient SHA-3 Implementation». In: *2018 IEEE International Symposium on Circuits and Systems (ISCAS)*. 2018, pp. 1–5. DOI: 10.1109/ISCAS.2018.8351649 (cit. on p. 42).

[27] I. T. L. Computer Security Division. *Example values - cryptographic standards and guidelines: Csrc*. URL: https://csrc.nist.gov/projects/cryptographic-standards-and-guidelines/example-values (cit. on p. 66).

[28] Mojtaba Bisheh-Niasar, Reza Azarderakhsh, and Mehran Mozaffari-Kermani. «Instruction-Set Accelerated Implementation of CRYSTALS-Kyber». In: *IEEE Transactions on Circuits and Systems I: Regular Papers* 68.11 (2021), pp. 4648–4659. DOI: 10.1109/TCSI.2021.3106639 (cit. on pp. 71–73).

[29] Bernhard Jungk and Marc Stöttinger. «Hobbit — Smaller but faster than a dwarf: Revisiting lightweight SHA-3 FPGA implementations». In: *2016 International Conference on ReConFigurable Computing and FPGAs (ReConFig)*. 2016, pp. 1–7. DOI: 10.1109/ReConFig.2016.7857176 (cit. on pp. 71–73).

[30] Utsav Banerjee, Tenzin S. Ukyab, and Anantha P. Chandrakasan. *Sapphire: A Configurable Crypto-Processor for Post-Quantum Lattice-based Protocols (Extended Version)*. Cryptology ePrint Archive, Paper 2019/1140. https://eprint.iacr.org/2019/1140. 2019. DOI: 10.13154/tches.v2019.i4.17-61. URL: https://eprint.iacr.org/2019/1140 (cit. on pp. 71–73).

[31] Yufei Xing and Shuguo Li. «A Compact Hardware Implementation of CCA-Secure Key Exchange Mechanism CRYSTALS-KYBER on FPGA». In: *IACR Transactions on Cryptographic Hardware and Embedded Systems* 2021.2 (Feb. 2021), pp. 328–356. DOI: 10.46586/tches.v2021.i2.328-356. URL: https://tches.iacr.org/index.php/TCHES/article/view/8797 (cit. on pp. 71, 72).

[32] Wenbo Guo, Shuguo Li, and Liang Kong. «An Efficient Implementation of KYBER». In: *IEEE Transactions on Circuits and Systems II: Express Briefs* 69.3 (2022), pp. 1562–1566. DOI: 10.1109/TCSII.2021.3103184 (cit. on p. 72).

[33] Deepraj Soni and Ramesh Karri. «Efficient Hardware Implementation of PQC Primitives and PQC algorithms Using High-Level Synthesis». In: *2021 IEEE Computer Society Annual Symposium on VLSI (ISVLSI)*. 2021, pp. 296–301. DOI: 10.1109/ISVLSI51109.2021.00061 (cit. on pp. 72, 73).