Politecnico di Torino

Tesi di Laurea Magistrale

Integration and optimization of a RISC-V based Keccak accelerator

Relatori: Prof. Guido Masera, Prof. Maurizio Martina, Ing. Alessandra Dolmeta

Candidato:

Mattia Mirigaldi

April 6, 2023

ABSTRACT

Nowadays the use of cryptography is pervasive. It is an essential component of modern digital infrastructure and has various applications that go from financial transactions to medical devices.
Cryptographic systems are built on hard mathematical problems, hard since it is assumed they require an exponential time to be solved. For example, the security of online communication relies on the hardness of the RSA, whose hardness is based on factorizing the product of two large prime numbers. The previous statement has been challenged by Peter Shor in 1994. He provided an algorithm that runs on a quantum computer that could solve the underlying RSA problem in polynomial time. This discovery implies that many commonly used cryptosystems would be completely broken if large quantum computers would exist. Considering the progress at which quantum computers are developing, this poses a big threat to current security protocols.
Since 2015 the National Institute of Standards and Technology (NIST) is running a selection process to define one or more quantum-resistant algorithms. These algorithms are NP-hard to solve by quantum computers and are defined as post-quantum cryptography (PQC) algorithms. Among the finalists, lattice-based cryptographic systems are the most promising ones. The computational complexity of these algorithms though requires non-negligible running time in deployments scenarios.
The state-of-the-art solutions consist of reducing processing unit workload by accelerating them totally or partially in hardware.
In this thesis the acceleration of the lattice-based algorithm CRYSTALS-Kyber has been investigated: a hardware accelerator has been developed and tested with the RISCV-based advanced microcontroller PULPissimo. In a preliminary step, by profiling the Kyber algorithm, it has been found that the Keccak sub-function is the most expensive in terms of running time and therefore the most promising to accelerate.
The keccak accelerator is built upon Bertoni's team implementation and has been tailored to achieve the best performance with PULPissimo SoC. The accelerator is connected to SoC peripherals and communicates with PULPissimo through an AXI interface. The accelerator shows outstanding results in its preliminary form, then there is ample room for improvement.
The results achieved by the proposed architecture highlight the importance of an accelerator with the CRYSTALS-Kyber algorithm and is a starting point to explore the acceleration of others PQC algorithms and cryptographic primitives.

Contents

List of Tables

List of Figures

List of code snippets

Glossary

confidentiality It prevents sensitive information from unauthorized access attempts. It is common for data to be categorized according to the amount and type of damage that could be done if it fell into the wrong hands. 7

cryptanalytic attack Is an attack to a cryptographic system, the attack is dependent on the algorithm's nature as well as knowledge of the general qualities of the plaintext, which can be a traditional English document or Java code. 1

cryptography Is a method of protecting information and communications through the use of codes, so that only those for whom the information is intended can read and process it. 1

cryptosystem Is shorthand for "cryptographic system", a cryptosystem is a suite of cryptographic algorithms needed to implement a particular security service, such as confidentiality (encryption). Typically, a cryptosystem consists of three algorithms: one for key generation, one for encryption, and one for decryption. 1

decryption oracle Is a function that is available to the adversary and that provides the decryption of any ciphertext of her choice, except for the challenge ciphertext. 9

eXtendable-Output Function A function on bit strings in which the output can be extended to any desired length.. 15

lattice-based problem In computer science, lattice problems are a class of optimization problems related to mathematical objects called lattices. Lattice problems are an example of NP-hard problems which have been shown to be average-case hard, providing a test case for the security of cryptographic algorithms. 9

polylogarithmic time An algorithm is said to run in polylogarithmic time if $T(n) = O(\log(n)^k)$ (also written as $T(n) = O(\log^k(n))$. 1

quantum computer Are machines that use the properties of quantum physics to store data and perform computations. 1

runtime the code executed before the main function is called on the core. 28

Acronyms

FLL Frequency-locked loop. 28, 46

IoT Internet of Things. 24

IPC Instruction Per Cycle. 26

KEM Key Encapsulation Mechanisms. 71

LWE Learning With Errors. 9

NIST National Institute of Standards and Technology. iv, 2, 3, 71

PQC Post Quantum Cryptography. v, 2

SCA Side channel attacks. 70

SDK Software Development Kit. 28

SoA State of the Art. 25, 68

SoC System on a Chip. 3, 25

CHAPTER 1

Introduction

1.1 Motivation

In today's interconnected world, where digital communication is widespread, cryptography has become a cornerstone of modern communication infrastructure. Without it, sensitive information would be at risk of being intercepted and compromised, and messages could not be trusted to be authentic.
In State of the Art (SoA) systems, the security of the connections relies on Public Key Cryptography (PKC) which employs a pair of keys denoted as public and private. Hence, data transmission is protected by PKC algorithms which are based on hard mathematical problems such as the *Rivest–Shamir–Adleman (RSA)*[1] and *Elliptic-Curve Cryptography (ECC)*[2].
The hardness of these problems depends on the difficulty of factoring the product of two large prime numbers. There are no published methods to defeat the system if a large enough key is used.
However, in 1994, Shor introduced a fast quantum algorithm[3] which can find the prime factorization of any positive integer in polylogarithmic time.
In recent years there has been a substantial amount of research on quantum computer and considering the rate at which quantum computers are developing, the building of a large quantum computer is only a matter of time. If large-scale quantum computers are ever built, they will be able to break many of the public-key cryptosystems currently in use, like the one previously mentioned.[4].
This would seriously compromise the confidentiality and integrity of digital communications on the Internet and elsewhere.
To withstand this threat, designing quantum-safe cryptosystems is crucial. *Post-quantum cryptography (PQC)* refers to cryptographic algorithms that are thought to be secure against a cryptanalytic attack by a quantum computer.
The goal of post-quantum cryptography (also called quantum-resistant cryptography) is to develop cryptographic systems that are secure against both quantum and classical computers and can interoperate with existing communications protocols and networks.
Post-quantum cryptography algorithms are based on math problems too but are more robust to quantum computer attacks since are much more complex.
The added complexity makes these algorithms highly inefficient on a general-purpose processor. State of the Art solutions consist of relieving CPU workload by accelerating these algorithms totally or partially in hardware.

1.2 NIST Post-Quantum Cryptography Standardization

In 2016 the National Institute of Standard and Technology (NIST) started a process[5] to solicit, evaluate, and standardize one or more quantum-resistant public-key cryptographic algorithms.
Since most symmetric primitives are relatively easy to modify in a way that makes them quantum-resistant, efforts have focused on public-key cryptography, namely digital signatures and key encapsulation mechanisms. The algorithms are designed for two main tasks for which encryption is typically used: general encryption, used to protect information exchanged across a public network; and digital signatures, used for identity authentication. In particular:

- **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.
- **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.

Initially, 82 submissions were received. Between them, only 69 candidates qualified for NIST competition's round 1. In 2018, there was the first NIST PQC standardization conference. Out of the initial candidates, in January 2019, 26 candidates were chosen to compete in round 2. In 2020, for the third round, 7 candidates were chosen as finalists and 8 candidates were considered as alternates.
On July 5, 2022, NIST announced the first group of winners from its six-year competition: the four selected encryption algorithms will become part of NIST's post-quantum cryptographic standard, expected to be finalized in about two years.

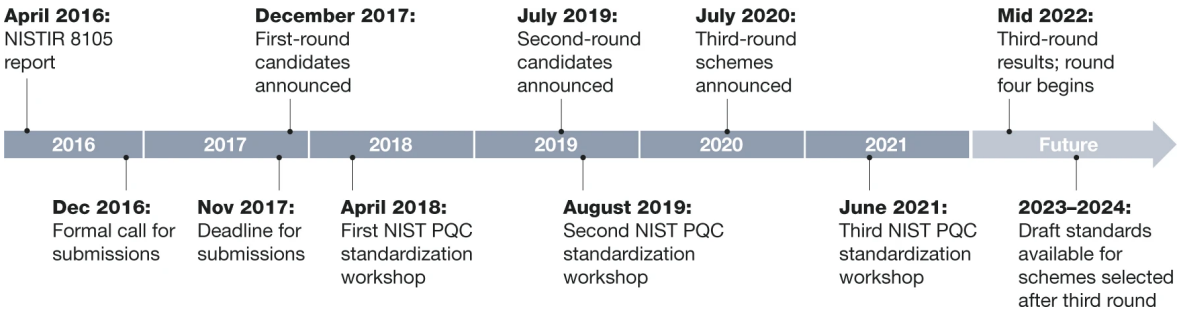


Figure 1.1: PQC standardization history [6]

The notable events during the course of the NIST PQC standardization process are shown, from its inception in 2016 to the present day. This process is the longest and most comprehensive study into PQC conducted thus far.

Four additional algorithms are under consideration for inclusion in the standard, and NIST plans to announce the finalists from that round at a future date.
NIST is announcing its choices in two stages because of the need for a robust variety of defense tools. As cryptographers have recognized from the beginning of NIST's effort, there are different systems and tasks that use encryption, and a useful standard would offer solutions designed for different situations, use varied approaches for encryption, and offer more than one algorithm for each use case in the event one proves vulnerable. In Table 1.1 are reported the winner encryption algorithms of the 4th round.
For general encryption, used when we access secure websites, NIST has selected the ***CRYSTALS-Kyber algorithm***. Among its advantages are comparatively small encryption keys that two parties

can exchange easily, as well as its speed of operation.

	Standardized	**4th round candidates**
PKE/KEM	CRYSTALS-Kyber	Bike
		SIKE
		HQC
		Classic McEliece
Digital Signature	CRYSTALS-Dilithium	
	FALCON	
	SPHINICS+	

Table 1.1: NIST First Four Quantum-Resistant Cryptographic Algorithms and last round candidates

The *CRYSTALS-Kyber* algorithm is the algorithm investigated in this thesis work. In a preliminary step, by profiling the Kyber algorithm, it has been found that the *Keccak* sub-function is the most expensive in terms of running time and therefore the most promising to accelerate.
Then the Keccak accelerator has been developed. It is built upon Bertoni's team implementation[7] and has been integrated within the *PULPissimo* SoC, a *RISC-V*-based microcontroller.

1.3 Tristan project

The *TRISTAN* (Together for RISc-V Technology and ApplicatioNs) project is a European Union-funded initiative that aims to mature and expand the European RISC-V ecosystem for the next generation of industrial hardware to compete with existing commercial alternatives.
One of the project's key focuses is to promote European digital sovereignty and democratic access to a majority of the RISC-V IPs, which is achieved by leveraging the Open-Source community to gain in productivity and quality.
This thesis work contributes to the TRISTAN project by developing a dedicated accelerator for CRYSTAL-Kyber, a NIST quantum-resistant cryptographic primitive, in a RISC-V based system. All the IPs blocks and scripts developed during this project are openly available on GitHub [1].

The accelerator developed in this thesis has been accepted at the workshop "**OSHW 23**".
In this workshop researchers and practitioners in computer science discuss and share their latest research and developments in open hardware and open-source hardware.

1.4 State of the Art and thesis contribution

The quantum-resistant public-key algorithms selected by NIST require non-negligible computational resources to be executed. This is quite critical in low-power embedded devices that have a limited amount of resources and computational power.
One viable solution is to accelerate them totally or partially in hardware, to alleviate the workload of the main processing unit.
There exist various HW design solutions that address this problem.
In [8] is designed and built an Application Specific Integrated Circuit (ASIC) accelerator for the Post-Quantum Digital Signature Scheme *XMSS*. This solution leads to the best performance results but requires considerable design effort and cost, which is undesirable when dealing with algorithms that

[1] https://github.com/aledolme/pqc_riscv

are still under evaluation.
Another solution may be to bring a small hardware accelerator directly into the processor pipeline, like done in works [9] and [10], or to use a co-processor like in work [11]. These solutions though cause a large overhead in area and energy. Also, the normal flow of operations in the processing unit is modified and as a consequence, these hardware implementations make the system vulnerable to side-channel attacks[12].
In this thesis work, the focus has been developing an external HW accelerator for the CRYSTALS-Kyber algorithm. The choice of an external accelerator has been preferred since it is a versatile solution that can be used with different computer architectures without the need for any major modifications. It can provide a significant speedup, without paying a large overhead of area, and also is a safe solution against side-channel attacks. However, it may not always provide the best performance when compared to a co-processor or in-pipeline accelerator.
In the initial phase, the algorithm has been analyzed and the most worthy parts to be accelerated have been identified. The accelerator has been designed to deliver high performance being careful not to add too much overhead area. To make it very versatile and easy to attach it has an *AXI* interface, indeed the AXI protocol is commonly used in System on Chip (SoC).
The accelerator has been tested with PULPissimo [13], an open-source RISC-V based microcontroller. The results obtained (Table 6.1) show that the accelerator has a large impact on the algorithm speed-up and demonstrate how effective a hardware accelerator can be when dealing with post-quantum cryptography algorithms.

1.5 Thesis structure

After this first introductory chapter 1, the thesis is structured as follows:

- **chapter 2 :** in section 2.1 a concise overview of the cryptography primitives is presented, followed by a discussion on the *CRYSTALS-Kyber* algorithm and its main module in section 2.2. Finally, the computational profile of the algorithm is presented and the reasons why accelerating the Keccak sub-function is the optimal choice are discussed.
- **chapter 3:** the Keccak algorithm is presented in detail in section 3.1, while the focus shifts to the design of the Keccak in section 3.2. Lastly, in section 3.3 is presented the hardware implementation of the Keccak accelerator.
- **chapter 4 :** in section 4.2 the PULPissimo microcontroller, the cross-compiler, and the runtime utilized in the project are introduced. Furthermore, is discussed which of the RISC-V architecture supported by the PULP group is the best one for this project. Additionally, in section 4.3, a detailed description is given of how to build the simulation platform and run a basic test application. Then in section 4.4 is evaluated the benchmark of the CRYSTALS-Kyber algorithm on PULPissimo.
- **chapter 5 :** show all the steps to integrate the Keccak IP as an external accelerator in PULPissimo. In addition, the Keccak accelerator is optimized to achieve the best performances (in section 5.8), and the speed-up contribution is tested running the *CRYSTALS-Kyber* algorithm (in section 4.4). After that, in section 5.9 the PULPissimo system with the added accelerator is synthesized with *Xilinx Vivado 2022.1* and implemented on "Xilinx Artix XC7A75-3 ".
- **chapter 6 :** in section 6.1 the results of the accelerator are reported and compared against the software implementation, while in section 6.2 the accelerator is compared with state of art counterparts. Additionally, a discussion on future research activities is presented in section 6.3.
- **chapter 7 :** is the conclusion of this thesis, the results of the accelerator are summarized, and the effectiveness of hardware acceleration is discussed.

CHAPTER 2

CRYSTALS-Kyber

The *CRYSTALS-Kyber* is a post-quantum secure key exchange protocol (KEM). It is a cryptographic hash function winner of the NIST hash function competition in 2022.
This chapter begins with a brief background on modern cryptographic techniques, followed by an overview of the CRYSTALS-Kyber algorithm and its main components. The profile of the algorithm from a computational point of view is then presented, and it is discussed why the Keccak sub-function is an ideal candidate to be accelerated.

2.1 Cryptography overview

Cryptography is the practice and study of secure communications techniques that allow only the sender and intended receiver of a message to view its contents[14]. The message wanted to transmit is encrypted, meaning that a sequence of operations is applied to make it unintelligible ("*cipher text*"). Only the receiver must be able to decrypt the message, meaning to convert the cipher text back into the original message.
Encryption in most cases involves an algorithm and a key.
The sequence of steps, or series of mathematical equations, used to describe a cryptographic process (i.e. encryption, decryption,...) defines a cryptographic algorithm.
There are three general classes of NIST-approved cryptographic algorithms, which are defined by the number or types of cryptographic keys that are used with each.

Hash functions

A cryptographic hash function takes in input data of an arbitrary length and through a one-way process transforms it into another compressed value (Figure 2.1).
The returned value has a fixed length and is called "*message digest*"or "*hash value*".
The hash functions allow uniquely identifying any content in cryptography. The obtained hash value is unique to the input (*deterministic*) and cannot be reversed back into the original input (*irreversible*). For these properties, hashing is mainly used for comparison reasons like verifying that the message received came from the right person and has not been tampered with.

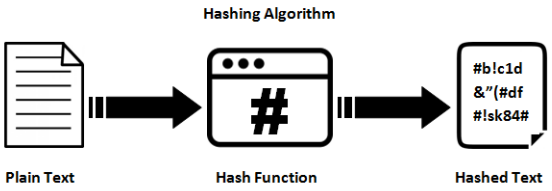


Figure 2.1: Cryptographic hash function [15]

Symmetric-key algorithms

In the symmetric-key algorithm, also referred to as "*secret-key algorithm* ", a single key is responsible for encrypting and decrypting data. The involved parties share that key, hence the key is considered symmetric, and they can use it to decrypt or encrypt any messages they want (Figure 2.2).
Symmetric encryption is the faster and more efficient method of encryption.
Because of this, symmetric encryption is generally used for encrypting databases where secret key sharing is not a problem.

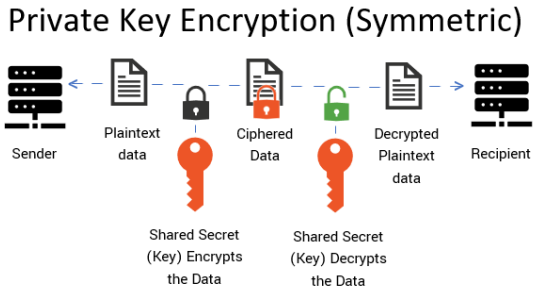


Figure 2.2: Symmetric key encryption [16]

Asymmetric-key algorithms

The asymmetric cryptography algorithms also referred to as "*public-key algorithms* ", enable parties to establish a secure communications channel across an inherently insecure network.
In asymmetric cryptography, the key used for encryption/decryption is divided into two parts: a public key and a private key. The public key can be given to anyone, trusted or not, while the private key must be kept solely by the owner of that key pair just like the key in symmetric cryptography.
The private key cannot be mathematically calculated through the use of the public key even though they are cryptographically related.
Asymmetric cryptography has two primary use cases:

- **Authentication**: messages can be signed with a private key, and then anyone with the public key is able to verify that the message was created by someone possessing the corresponding private key.
- **Encryption**: someone with the public key is able to encrypt a message, providing confidentiality, and then only the person in possession of the private key is able to decrypt it.

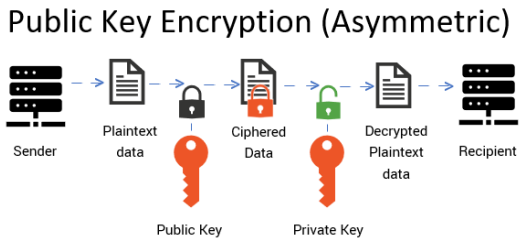


Figure 2.3: Asymmetric key encryption [16]

2.1.1 Key Encapsulation Mechanism (KEM)

The public, or asymmetric, key encryption is usually less efficient than symmetric encryption schemes. The length of asymmetric keys needs to be much longer to offer the same level of security, due to the mathematical link between public and private keys that may be exploited to crack the encryption. Also, since symmetric encryption uses a single key for both encryption and decryption, it requires less computational overhead compared to asymmetric encryption, which requires more complex mathematical operations.
The asymmetric cryptosystems result slower compared to symmetric ones and are suitable only for transmitting small amounts of data.
On the internet, two parties cannot meet in advance to agree on a shared secret key. It follows that is needed a way to exchange securely a key so that the two can start a symmetric-key communication.
The **Key Encapsulation Mechanism (KEM)** is a cryptographic technique that provides a way to securely exchange the key between two parties in a public key cryptography system (hence insecure channel). After the symmetric key is shared, the two parties can use a symmetric algorithm to efficiently encrypt data.
KEM makes it possible through a collection of three algorithms:

- A key generation algorithm,"*Generate*", which generates a public key and a private key (a keypair).
- An encapsulation algorithm, "*Encapsulate*", which takes as input a public key, and outputs a shared secret value and an "encapsulation" (a ciphertext) of this secret value.
- A decapsulation algorithm, "*Decapsulate*", which takes as input the encapsulation and the private key, and outputs the shared secret value.

The KEM procedure is the following:

1. Key Generation: The sender generates a random symmetric key that will be used to encrypt the data to be transmitted.
2. Encryption: The sender then generates a random value, called the "ephemeral key," and encrypts the symmetric key using the recipient's public key and the ephemeral key. This produces a ciphertext that is sent to the recipient along with the ephemeral key.
3. Decryption: The recipient uses their private key to decrypt the ephemeral key from the ciphertext sent by the sender.
4. Key Derivation: The recipient then uses the decrypted ephemeral key and a key derivation function to generate the same symmetric key that the sender used in step 1.

5. Data Encryption: The sender and recipient can now use the shared symmetric key to encrypt and decrypt the data transmitted over the insecure channel.

The advantage of using a KEM is that it enables two parties to establish a shared secret key without having to directly exchange the key itself. This is particularly useful in scenarios where the key exchange must take place over an insecure channel.

2.2 CRYSTALS-Kyber

The CRYSTALS-Kyber algorithm is a key encapsulation method (KEM) designed to be resistant to cryptanalytic attacks by future powerful quantum computers. The CRYSTALS-Kyber algorithm is the winner of NIST "PQC standardization contest"[5] and is the one that will be standardized for general encryption.
According to the algorithm's creators, the CRYSTALS-Kyber is an *IND-CCA2-secure* key encapsulation mechanism (KEM), whose security is based on the hardness of solving the *learning-with-errors (LWE)* problem over module lattices.
The IND-CCA2 stands for "indistinguishability under chosen ciphertext attack" and is a security notion for public-key encryption schemes. An encryption scheme is IND-CCA2-secure if it is secure against chosen ciphertext attacks (CCA). In other words, an adversary who has access to the decryption oracle cannot distinguish between the decryption of two ciphertexts, one of which was generated by the encryption of a random message and the other one was generated by the encryption of a message chosen by the adversary. This is an important security property for public-key encryption schemes.

2.2.1 Learning with Errors (LWE) problem

The LWE problem is a lattice-based problem in the field of cryptography that forms the basis for several cryptographic schemes. The LWE problem is a hard computational problem[17], meaning there is no known efficient algorithm to solve it.
The LWE is a method defined by Oded Regev in 2005, it involves finding a solution to a system of linear equations in the form of:

$$B = A * s + e \tag{2.1}$$

Where

- **A** is a random matrix with integer coefficients
- **e** is a random vector of small integers (*error*)
- **s** is the vector of secret integers
- **B** is the result vector of integers

The values of **A** and **B** become the public key, and the LWE involves the difficulty to recover the vector **s** from the given values.
The best-known algorithms for solving the LWE problem have a running time that grows exponentially with the size of the parameters, making it computationally infeasible to solve the problem for large values of q and n (modulo and dimension of the vectors).

2.2.2 Kyber algorithm

Kyber's construction involves a two-stage process[18].
Firstly, ***Kyber.CPAPKE*** (Chosen-Cyphertext Attack Public Key Encryption), an INDCPA-secure public-key encryption scheme, is presented for encrypting messages with a fixed length of 32 bytes.
Secondly, the construction of an IND-CCA2-secure KEM (Chosen-Cyphertext Attack Key Encapsulation Mechanism) is achieved by utilizing a slightly modified ***Fujisaki-Okamoto (FO)*** transform.
The Kyber submission to the NIST competition lists three different parameter sets (Table 2.1) aimed at different security levels.

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

Table 2.1: Kyber Crypto Length

Specifically, Kyber-512 aims at security roughly equivalent to AES-128, Kyber-768 aims at security roughly equivalent to AES-192, and Kyber-1024 aims at security roughly equivalent to AES-256.

2.3 Kyber-PKE

Kyber-PKE is a public-key encryption scheme that ensures IND-CPA security for messages with a fixed length of 32 bytes. The scheme comprises three algorithms: Key Generation, Encryption, and Decryption.

Kyber-PKE Key generation

During *Key Generation*, the polynomial matrix $\mathbf{A}$ is randomly generated, while the polynomial vectors $\mathbf{s}$ and $\mathbf{e}$ are sampled according to B_{η_1} (where η_1 defines the noise level). The secret key is $\mathbf{s}$ and follows that the public key is **A*s+e**.
However, for efficient implementation, the multiplication **A*s** is performed in the NTT domain by generating $\mathbf{A}$ in the NTT domain (i.e., $\hat{\mathbf{A}}$) and transforming s to $\hat{\mathbf{s}} == \text{NTT}(\mathbf{s})$.
To avoid the NTT^{-1} operation, $\mathbf{e}$ is also transformed to $\hat{\mathbf{e}}$ and added to $\hat{\mathbf{A}}$ o $\hat{\mathbf{s}}$.
As a result, the secret and public keys are in the NTT domain and are encoded to *sk* and *pk*, respectively. Additionally, the seed for randomness is added to the public key, allowing the recipient to generate matrix $\mathbf{A}$.
The pseudo-code of "`KYBER.CPAPKE: KeyGen` "is found in Algorithm 1.
In summary, the Kyber.CPAPKE.KeyGen() algorithm generates in NTT domain a random polynomial $\mathbf{s}$, a random polynomial $\mathbf{e}$. Then computes the polynomial "a" as the product of a fixed polynomial "A" and "s". Then it generates a public key polynomial "b" and a matrix "t" using a pseudorandom function and a random binary string "seed". The public key is the tuple (b, p) where "p" is the sum of "a" and "e". The secret key is the tuple (s, t).

The algorithm starts by randomly generating the polynomial 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 $\leftarrow \mathcal{B}^{32}$
(ρ, σ) := 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 $\mathcal{B}_{\eta_1}$
  $\mathbf{s}[i]$:= CBD_{η_1}(PRF(σ,N))
  N:= N+1
end for
for i from 0 to k-1 **do** ▷ Samples $\mathbf{e} \in \mathcal{R}_q^k$ from $\mathcal{B}_{\eta_1}$
  $\mathbf{e}[i]$:= CBD_{η_1}(PRF(σ,N))
  N:= N+1
end for
$\hat{\mathbf{s}}$:= NTT($\mathbf{s}$)
$\hat{\mathbf{e}}$:= 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)

Kyber-PKE Encryption

The process of encrypting message $\mathbf{m}$ to ciphertext $c = (c_1, c_2)$ in Kyber-PKE Encryption involves using the public key pk and random coins $\mathbf{r}$. The polynomial vector t and matrix A are obtained from the public key. The polynomial vector $\mathbf{r}$ is sampled according to $\mathbf{B}_{\eta_1}$ using $\mathbf{r}$. The polynomial vectors $\mathbf{e_1}$ and the e_2 are sampled according to B_{η_2} using $\mathbf{r}$. Then, normally, the ciphertext $c = (c_1, c_2)$ is $(\mathbf{A^T r} + \mathbf{e_1}, \mathbf{t^T}r + e2 + m)$. The ciphertext c is obtained by performing multiplications in the NTT domain and transforming them to the normal domain using NTT^{-1}.
The pseudo-code of "`KYBER.CPAPKE: Encryption` "is found in Algorithm 2.

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$_{\eta_1}$(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$_{\eta_2}$(PRF(r,N))
  N:= N+1
end for
e_2 := CBD$_{\eta_2}$(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$_q$(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

Kyber-PKE Decryption

To perform Kyber-PKE Decryption, the polynomial vector $\mathbf{u}$ and $\boldsymbol{v}$ are first obtained from the cipher-text by decoding and decompressing it. The vector $\mathbf{s}$ is then obtained from the secret key. The message $\boldsymbol{m}$ is calculated by subtracting the dot product of $\mathbf{s}$ and $\mathbf{u}$ from $\boldsymbol{v}$ ($m = v - \mathbf{s^T u}$.). Multiplications are performed in the NTT domain and then transformed to the normal domain 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 $m \in \mathcal{B}^{32}$

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

Kyber KEM

Kyber.CCAKEM is an IND-CCA2-secure KEM that utilizes the Fujisaki-Okamoto transform to create a three-step process: Key Generation, Encapsulation, and Decapsulation.The procedure is illustrated in Figure 2.4.
During the first step, the *sender* generates public and secret keys using the Kyber-PKE Key Generation algorithm and shares her public key with the *receiver*. In the second step, the *receiver* encrypts the message using the Kyber-PKE Encryption algorithm, computes the shared secret, and sends the ciphertext to the *sender*. In the last step, the *sender* decrypts the ciphertext and verifies if it can be encrypted to the same ciphertext as the *receiver*. If so, she computes the shared secret using the message, her public key, and the ciphertext. Otherwise, she uses a random value and the ciphertext to compute the shared secret.

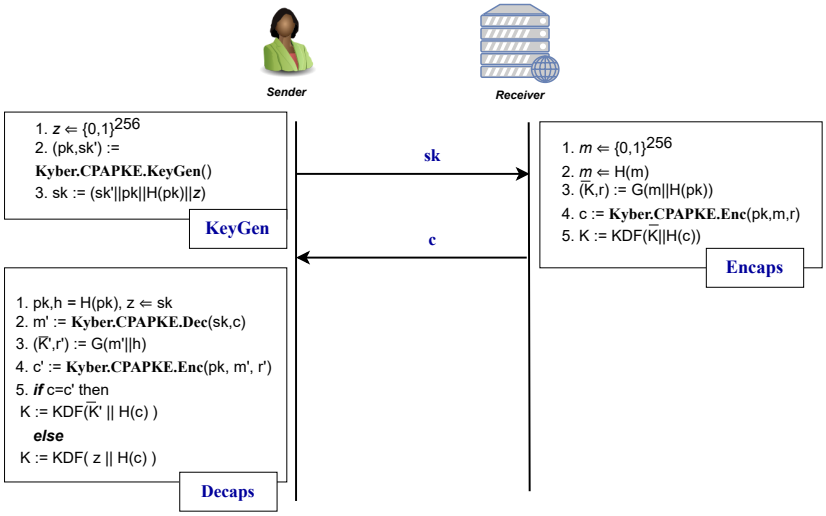


Figure 2.4: Kyber-CCA KEM

Kyber C-code analysis

The computational cost of the CRYSTALS-Kyber algorithm is analyzed by compiling the C-code provided by the CRYSTALS-Team.[1]
The focus is on the *test_kyber*$*ALG* program, which tests key generation, encapsulation, and decapsulation 1000 times. Where *ALG* value ranges over the parameter set 512, 768, and 1024. The implementation used is the one in **ref/**, which prioritizes clean code over-optimization and is, therefore, slower than an optimized implementation.
Regardless of the level of security analyzed, the code is the same. The only difference is in the definition of the parameter involved, with changes only made to parameter definitions in the param.h file based on the level of security analyzed.
As security levels increase, the size of keys and ciphertexts also increase, resulting in more function calls.Figure 2.5 shows the percentage of program running time used by different functions, with the computation of the Keccak permutation being particularly time-consuming despite not being the most frequently called.

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

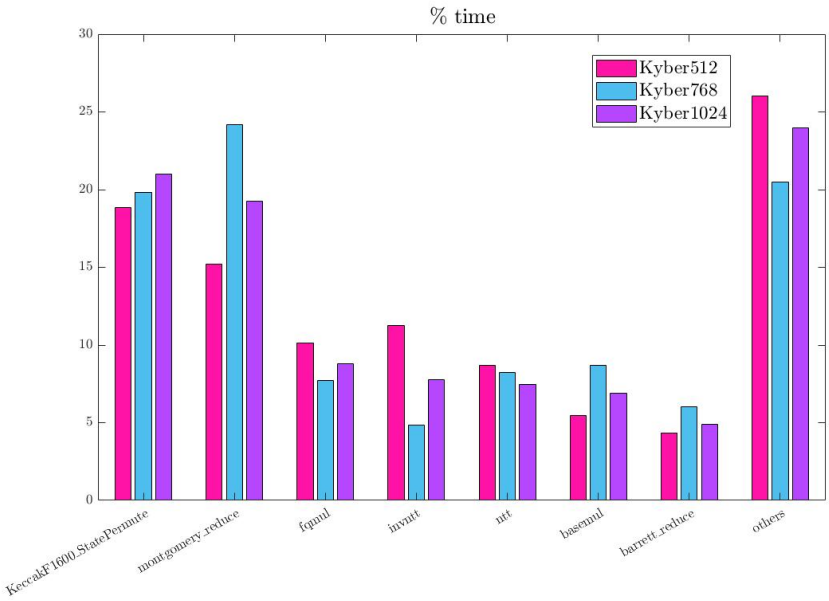


Figure 2.5: Percentage of the total running time of the program used [19]

- In Kyber512 and Kyber1024, `KeccakF1600_StatePermute` is the function that occupies most of the running time. This is the core of the Keccak algorithm, which is a type of cryptographic sponge function.

- In Kyber768, the `Montgomery_reduction` function occupies most of the time running time. However, as for Kyber512 and Kyber1024,`KeccakF1600_StatePermute` is called a number of times which is two orders of magnitude less with respect to `Montgomery_reduction`.

The Keccak function is the one that in almost all three levels of security occupies most of the running time, even if it is called a low amount of time. In this thesis work the Keccak function has been the one selected to be accelerated.

CHAPTER 3

Keccak

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).
On August 5, 2015, the U.S. government announced the SHA-3 as the new standard for cryptographic hash algorithms[20]. The SHA-3 family employs the permutation that in this standard is called *Keccak-p*.
Keccak is a versatile cryptographic function best known as a hash function, it nevertheless has been standardized by NIST in various applications, including digital signatures, key derivation, and message authentication codes. The Keccak permutation is designed to be highly resistant to various attacks, including differential and linear cryptanalysis, and is considered to be one of the most secure cryptographic primitives available today.
Keccak is used in different forms such as SHAKE128 and SHAKE256, which are eXtendable-Output Functions, and SHA3-256 to SHA3-512, which are hash functions. These standardized forms of the Keccak algorithms are specified in the FIPS-202 standard, these are reported in Table 3.3.
The Keccak function is the one that occupies most of the CRYSTALS-Kyber running time and for this reason, is the one being accelerated. This chapter describes the Keccak algorithm in detail and lastly, presents the developed Keccak HW implementation.

3.1 Keccak permutation

The Keccak function is a cryptographic hash function that operates on messages of arbitrary length and produces a fixed-length output (*digest*).
The basis of this algorithm is the sponge construction[21]. Inside the sponge, the input data are divided into blocks of the same length and the Keccak-f permutation is applied to them.
There are 7 permutations in total denoted as Keccak-f[b], where b = 25×2^l and l ranges from 0 to 6. The four SHA-3 hash functions (*SHA3-224, SHA3-256, SHA3-384,* and *SHA3-512*) use the Keccak-f[1600] permutation, which means that the length of the state is 1600 bits. The 1600-bit state of Keccak [1600] consists of **5x5 matrix of 64-bit words**[7], as shown in Figure 3.1. This is the width of the permutation.
The number of round n_r depends on the permutation width and is given by $n_r = 12 + 2 * l$, where $2^l = \frac{b}{25}$. This gives ***24 rounds*** for each compression step of the `Keccak-f[1600]`.
The algorithm pseudo-code is shown in Algorithm 4.

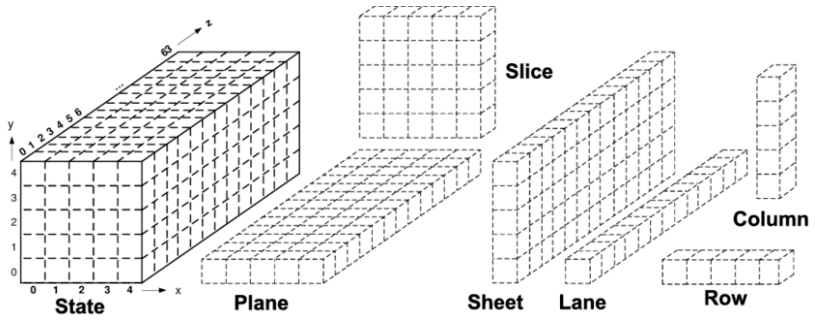


Figure 3.1: Keccak state [7]

Algorithm 4 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 which are θ, ρ, π, χ and ι. [22].
Each round applies a set of operations to the input state, denoted as **A**, to produce an output state, denoted by **A***.
Before each round, a round key is added to the state. The round key is derived from the input data and the current round number and is added to the state using XOR operations
The different five steps, depicted in Figure 3.2 and whose pseudo-code implementation is reported in Algorithm 5, can be briefly described as follow:

- θ is responsible for adding diffusion to the state. The θ step uses a linear transformation to mix the bits of the state, it computes the parity of each of the 5×64=320 columns, and XORs the result with two neighboring columns chosen in a regular pattern;
- ρ and π states are responsible for adding confusion to the state.
  The ρ step performs bit rotations on the state by rotating 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 3.1). The π step reorders the bits within each lane of the state, it permutes

	X=3	X=4	X=0	X=1	X=2
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=5	21	8	41	45	15

Table 3.1: Values r[i] constants

the 5×5=25 64-bit words using a fixed pattern and rearranges the positions of the lanes;

- χ step is responsible for adding non-linearity to the state by applying a non-linear function to each lane of the state. The non linearity function is a combination of bitwise XOR, NOT and AND operations;
- ι is responsible for adding round constants to the state by XORing the round constant (from Table 3.2) 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 3.2: Values RC[i] constants

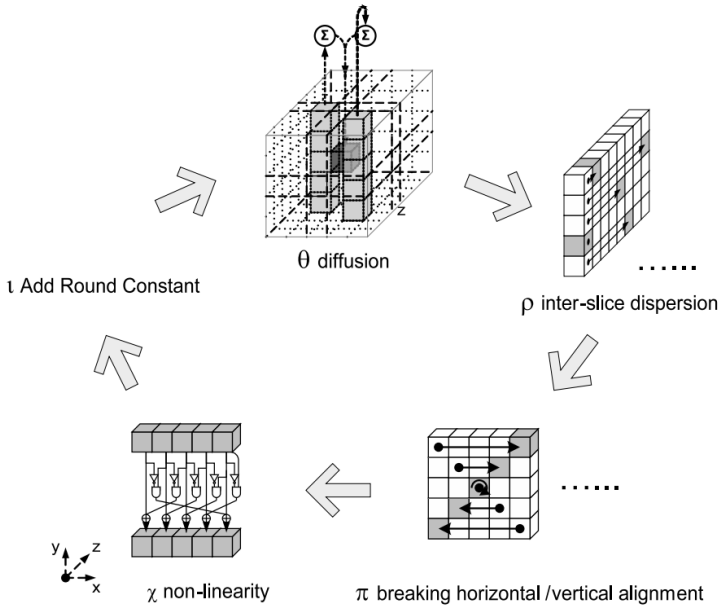


Figure 3.2: Keccak-f steps [21]

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) is the bitwise cyclic shift operation, moving the bit at position i into position $i + r$ (modulo 64).

Algorithm 5 Round[b]

procedure ROUND[B]((A,RC))
▷ θ **step**
C[x]= A[x,0] $\oplus$ A[x,1] $\oplus$ A[x,2] $\oplus$ A[x,3] $\oplus$ A[x,4] $\oplus$, ▷ $\forall$x in 0...4
D[x]= C[x-1] $\oplus$ ROT(C[X+1],1), ▷ $\forall$x in 0...4
A[x,y]=A[x,y]$\oplus$ D[x] ▷ $\forall$ (x,y) in (0...4, 0...4)
▷ ρ and π **steps**
B[y,2x+3y]=ROT(A[x,y], r[x,y]) ▷ $\forall$ (x,y) in (0...4, 0...4)
▷ χ **step**
A[x,y]=B[x,y] $\oplus$ ((NOT B[x+1,y]) AND B[x+2,y]) ▷ $\forall$ (x,y) in (0...4, 0...4)
▷ ι **step**
A[0,0]=A[0,0] $\oplus$ RC
return A
end procedure

The constants `RC[i]` and `r[x,y]` are cyclic shift offset and round constant respectively (Table 3.2 and Table 3.1)

3.2 Keccak design

The construction of the Keccak function is based on the "sponge construction"[21], a design paradigm for building hash functions and other cryptographic primitives.
The sponge function has the structure illustrated in Figure 3.3.
The sponge function is constructed from two main components: a fixed-size permutation function, which operates on a state consisting of a fixed number of bits, and a padding rule, which transforms the input message into a sequence of blocks that can be processed by the permutation function. The sponge function is initialized with an initial state that consists of all zeros. The input message M is divided into blocks of r bits (if necessary they are completed with 0s if the number of bits in the last block is less than r). Then the sponge operation involves two phases: an absorbing phase and a squeezing phase.

- In the **absorbing phase**, the r-bit input message blocks are XOR-ed with the first r bits of the state, interleaved with applications of the permutation function f. Once all the blocks of the message are processed, the sponge function moves on to the "release"phase.
  If the last block of the input message is not a full block, it is padded with a bit that signals the end of the message, followed by enough zeros to fill the block. This ensures that the input message has a fixed length, which is required for the sponge construction.
- In the **squeezing phase** the final hash value is obtained. Specifically, the output blocks are obtained by taking the first r bits of the state, and then applying the permutation function to the entire state to update it. This process is repeated until the desired number of output blocks has been generated.
  The last c bits of the state never depend directly on the input blocks and are never released as output in the squeezing phase, but are only modified by the permutation f.

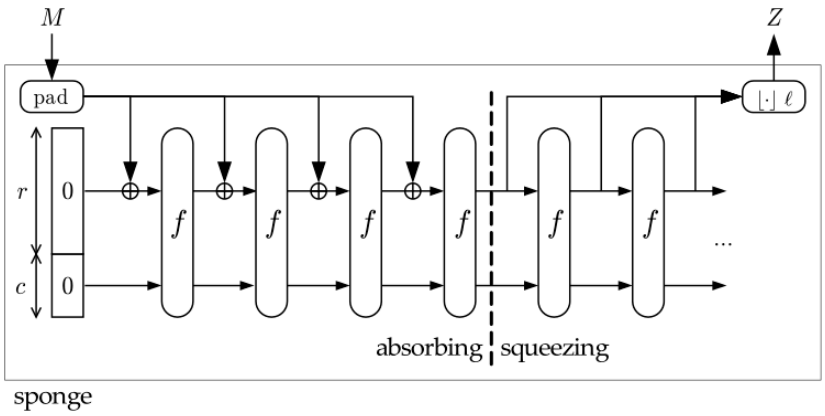


Figure 3.3: Sponge construction [21]

In Algorithm 6 is presented the pseudo-code for the `Keccak[r,c]` sponge function, with parameters capacity c and bit-rate r. It's assumed for simplicity that r is a multiple of the lane size.
In the pseudo-code, the input message `M` is represented as a string of bytes $Mbytes$ followed by a number (possibly zero, at most 7) of trailing bits $Mbits$. In the standard instances typically are added a few trailing bits for domain separation. [1].

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

The delimited suffix is d, which encodes the trailing bits *Mbits* and their length. The padded message P is organized 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.

Algorithm 6 Pseudo-code description of the sponge functions

procedure KECCAK[R,C](M)
  ▷ Padding
  d = $2^{(|Mbits|)}$ + sum for i=0 ... |Mbits|-1 of $2^{(i*Mbits[i])}$
  P = Mbytes || d || 0x00 || ... || 0x00
  P = P ⊕ (0x00 || ... || 0x00 || 0x80)
  ▷ Absorbing phase
  S[x,y] = 0 ▷ ∀ (x,y) in (0...4, 0...4)
  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)
  end for
  ▷ Squeezing phase
  Z = empty string
  while output is requested
  Z = Z || S[x,y] ▷ ∀ (x,y) such that x+5*y ¡ r/w
  S = Keccak-f[r+c](S)
  return Z
end procedure

3.3 Keccak HW implementation

The designed HW accelerator is specifically dedicated to the CRYSTALS-Kyber application, the SHA-3 primitives required by the algorithm with their parameters are listed in table Table 3.3.

	r	c	**Output length (bits)**	**Security level (bits)**	Mbits
SHA3-256	1088	512	256	128	01
SHA3-512	576	1024	512	256	01
SHAKE128	1344	256	unlimited	128	1111
SHAKE256	1088	512	unlimited	256	1111

Table 3.3: Standard Keccak primitive used in CRYSTALS-Kyber

All of these SHA-3 primitives can be obtained by calling the same Keccak accelerator, which implements the Keccak-f[1600] permutation.
The hardware architecture provided by the Keccak team has been used[23], which is a straightforward implementation of the Keccak function. The architecture is composed of the logic for one round and the register for storing the state, as can be appreciated in the block diagram in Figure 3.4.

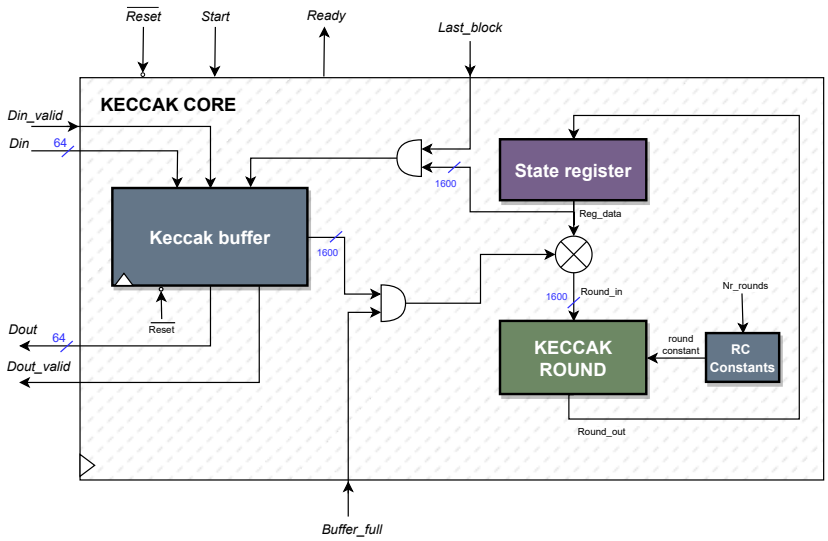


Figure 3.4: Block diagram of the Keccak accelerator

The accelerator performs the Keccak permutation and therefore the whole 1600 bits are processed and saved, without distinguishing between rate-capacity bit.
The XOR is needed to put as input din_buffer_reg at the first iteration, and the output of the state register for the following ones. Due to the short critical path, this design ensures very high throughput. The throughput can be further increased by computing more rounds in a single clock cycle.

To get an idea of the performance results found with different round instantiations, check the Table 3.4, taken from [7]. Note that the technology is different from the one used in this thesis. Indeed, this thesis uses a 45 nm Nangate technology while the one in the table has been done with 130 nm technology.

Number of round instances	Size	Critical path	Frequency	Throughput
$n = 1$	48 kgates	1.9 ns	526 MHz	22.44 Gbit/s
$n = 2$	67 kgates	3.0 ns	333 MHz	28.44 Gbit/s
$n = 3$	86 kgates	4.1 ns	244 MHz	31.22 Gbit/s
$n = 4$	105 kgates	5.2 ns	192 MHz	32.82 Gbit/s
$n = 6$	143 kgates	6.3 ns	135 MHz	34.59 Gbit/s

Table 3.4: Performance estimation of variants of the high speed code Keccak-f[1600], with r=1024 and c=576 [7]

The round block implements the five steps of the Keccak algorithm, 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 section 3.1. The architecture of the Keccak round is depicted in Figure 3.5.

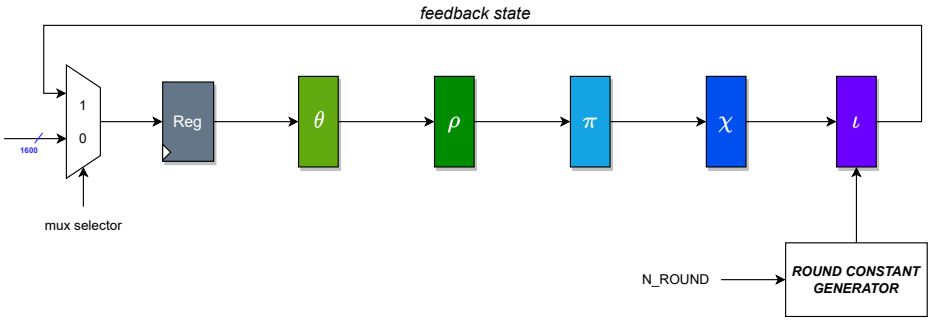


Figure 3.5: Keccak round architecture

The accelerator is designed in *vhdl* and has been tested with *ModelSim*, the top entity of the architecture can be appreciated in listing 1.

```
entity keccak is
    port (
    clk         : in  std_logic;
    rst_n       : in  std_logic;
    start       : in std_logic;
    din         : in  std_logic_vector(63 downto 0);
    din_valid   : in std_logic;
    buffer_full : out std_logic;
    last_block  : in std_logic;
    ready       : out std_logic;
    dout        : out std_logic_vector(63 downto 0);
    dout_valid  : out std_logic);
end keccak;
```

Listing 1: Top entity of the Keccak accelerator

The start signal is a synchronous reset which resets the internal buffer to state zero.
When received the start signal then the Keccak operation may start, its workflow can be divided into three sub-operations.

- Load input value: if the ready signal is pulled down then the architecture is ready to start a new permutation. If so, then the 1600 bits value on which is performed the permutation are stored 64-bit at the time. This phase last 25 cycles and when concludes the Buffer_full signal is pulled up.
- Permutation: when the input value has been completely stored then the Keccak can proceed to perform the permutation. As explained in section 3.1 24 rounds are required, therefore since this architecture computes one round each cycle then the permutation lasts 24 clock cycles.
- Output result: lastly the 1600-bit results of the permutation are stored in the internal buffer and are unpacked on D_out result 64-bit at the time. To unpack completely the output 25 cycles are required.

The timing simulation of the Keccak accelerator is shown in Figure 3.6, Figure 3.7 and Figure 3.8.

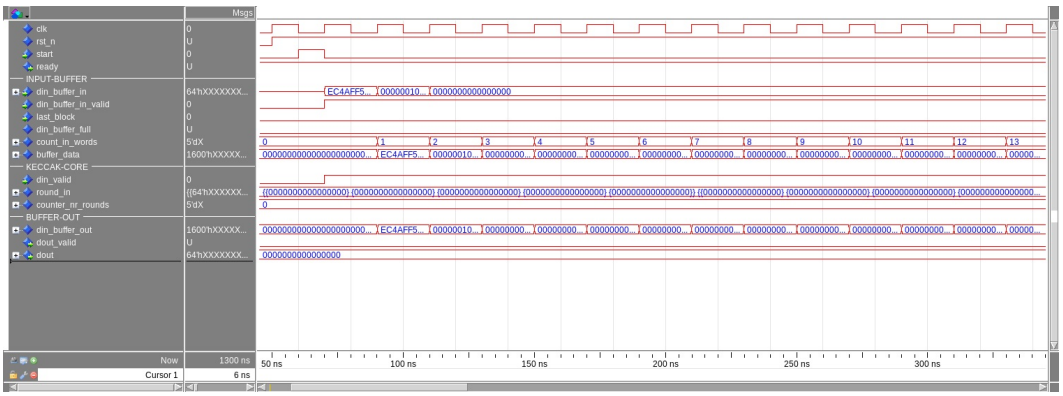


Figure 3.6: Load input value in Keccak internal buffer

The input value is 1600-bit wide, it is loaded into Keccak's internal buffer at chunks of 64-bit at each cycle. This process lasts 25 cycles.

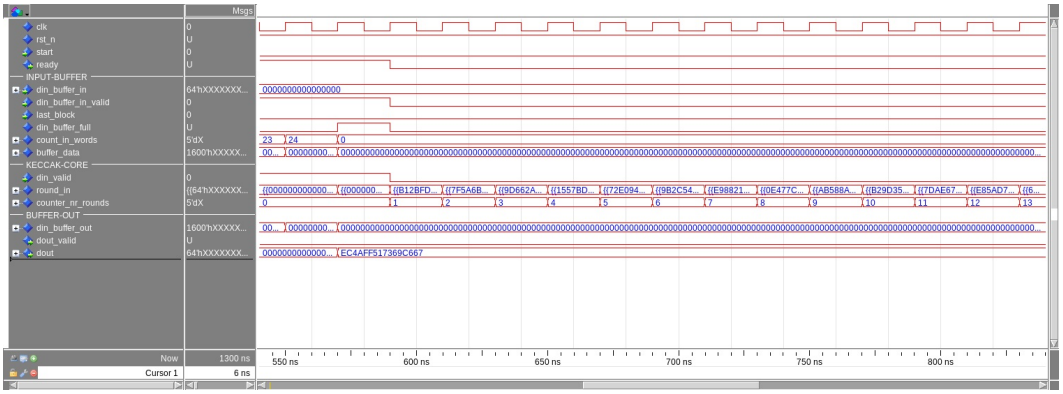


Figure 3.7: Keccak start of permutation

Once the input value has been loaded, the permutation can start and this phase lasts for 26 cycles. Due to synchronization requirements, it takes 2 cycles more than the one required to compute the permutation result. During the permutation, the ready signal is pulled down.

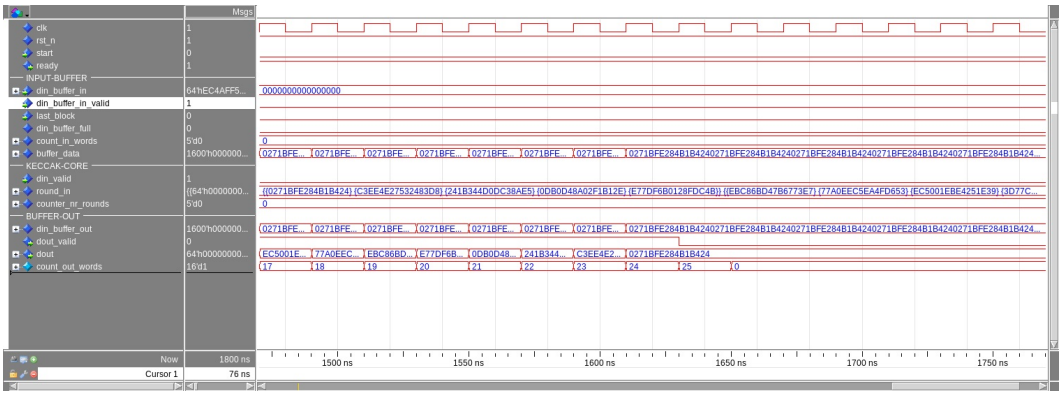


Figure 3.8: Output of Keccak permutation

The permutation produces a 1600-bit result, which is output in 64-bit chunks.

CHAPTER 4

PULPissimo

PULPissimo is an open-source, 32-bit microcontroller system-on-chip (SoC) designed for ultra-low power consumption and high energy efficiency. It is based on the PULP platform, which stands for Parallel Ultra Low Power, and is developed by the Integrated Systems Laboratory at the Swiss Federal Institute of Technology in Zurich (ETH Zurich)[13].

4.1 PULP platform

PULP is a silicon-proven Parallel Ultra Low Power platform targeting high energy efficiencies.
The primary objective of PULP is to create an advanced, open-source, and scalable hardware and software research and development platform. This platform aims to overcome the challenge of energy efficiency within a power envelope of a few milliWatts (mW), while also meeting the computational needs of IoT applications.
The open-source strategy has multiple benefits. To name a few :

- It promotes collaboration and knowledge sharing within the technology community.
  By making the hardware and software designs of the PULP platform openly available, it enables developers and researchers worldwide to access and contribute to the development of the technology, ultimately leading to a higher quality, more reliable, and more feature-rich solution.
- It fosters innovation by enabling developers to build upon existing technology and create new applications and use cases.
  By providing an open-source platform that is scalable and adaptable, PULP encourages innovation and empowers developers to create new solutions that may not have been possible with proprietary technology.

The hardware and software designs of the PULP platform are openly available and are based on the open-source RISC-V instruction set architecture[24].
RISC-V builds and improves upon the original Reduced Instruction Set Computer (RISC) architectures, it was developed at the University of California, Berkeley in 2010 and is now maintained by the RISC-V Foundation.
RISC-V is a load/store architecture, all operations are on internal registers. There are 32 registers, named x0 to x31, each 32 or 64, or 128 bits long.
The RISC-V ISA is unique because it is modular, which means it can be customized for specific applications, allowing developers to choose the features they need and leave out the ones they don't.
It is structured as a small base ISA with a variety of optional extensions.

I	Integer instructions
E	Reduced number of registers, used only 16 registers
M	Multiplication and Division
A	Atomic instructions
F	Single precision Floatingpoint
D	Double-Precision FloatingPoint
C	Compressed Instructions
X	Non Standard Extensions

Table 4.1: Standardized RISC-V extension

In table Table 4.1 the supported ones are reported. Only the **I** extension is mandatory. The type of architecture is specified as "*RV + word-width + extensions* ", like for example *RV32IMC* is a 32bit RISC-V processor with integer, multiplication, and compressed extensions.
The PULP-platform developed and maintains efficient implementations of RISC-V cores.
These include:

- 32 bit 1-stage Snitch[25]
- 32 bit 4-stage core CV32E40P (formerly RI5CY) [26]
- 64 bit 6-stage CVA6 (formerly Ariane)[27]
- 32-bit 2-stage Ibex (formerly Zero-riscy) [28]

PULP's platform is organized in clusters of RISC-V cores that share a tightly-coupled data memory. The platform encompasses a collection of Intellectual Property (IP) components that are described using the SystemVerilog hardware description language (HDL), accompanied by their corresponding simulation and synthesis scripts. The platform has released efficient 32 and 64 bits implementations, starting from simple micro-controllers, to the SoA OPENPULP release which sets a new bar for low-power multicore IoT processors.
To facilitate the use of the platform the PULP's platform includes runtime software that is written in C and RISC-V assembly, more about run-time is explained in the section 4.3.

4.2 PULPissimo architecture

PULPissimo is the second version of the PULPino system [29] and is the main microcontroller architecture of the more recent PULP chips.
PULPino and PULPissimo are microprocessor systems intended for direct use and contain most of the interfaces developed throughout the PULP project.
Both PULPino and PULPissimo contain only a single RISC-V core, there is no multi-core support. This means that most of the memory and cache-sharing infrastructure in multi-core PULP systems isn't needed for these smaller systems.
The PULPissimo block diagram is illustrated in Figure 4.1.
The peripherals are connected to the $uDMA$ which transfers the data to the memory subsystem efficiently. The $JTAG$ and the AXI plug have also access to the SoC. The AXI plug can be used to extend the microcontroller with a multi-core cluster or an accelerator.
As for PULPino, also with PULPissimo the processor core can be configured at design time by selecting either the "CV32E40P"or "Ibex".

The PULPissimo architecture includes:

- Either the CV32E40P core or the Ibex as main core;
- Autonomous Input/Output subsystem (μDMA) that can efficiently transfers data with the peripherals;
- Support for Hardware Processing Engines (HWPEs);
- Simple interrupt controller;
- Supports I/O on interfaces such as SPI (as master), I2S, Camera Interface (CPI), I2C, UART, JTAG.

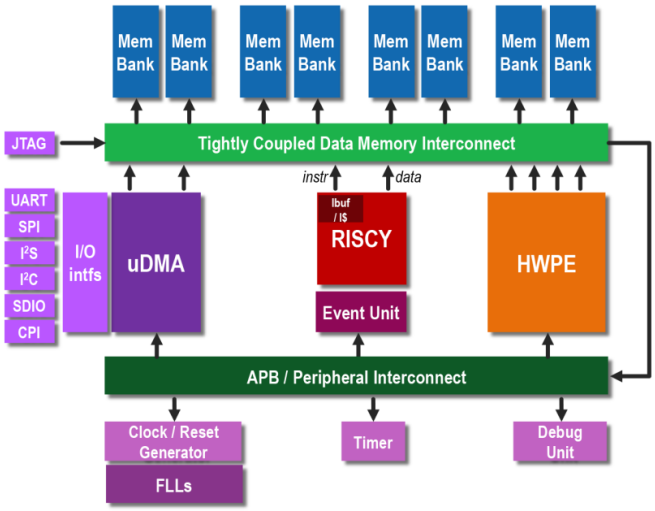


Figure 4.1: PULPissimo block diagram [13]

The "CV32E40P "[26], formerly RI5CY, is an in-order, single-issue core with 4 pipeline stages and it has an Instruction Per Cycle (IPC) close to 1. The block diagram is in Figure 4.2.
The processor has full support for the base integer instruction set (RV32I), compressed instructions (RV32C), and multiplication instruction set extension (RV32M). Also, it can be configured to have a single-precision floating-point instruction set extension (RV32F).
It implements several ISA extensions such as hardware loops, post-incrementing load, and store instructions, bit-manipulation instructions, MAC operations.
It supports fixed-point operations, packed-SIMD instructions, and the dot product.
It has been designed to increase energy efficiency in ultra-low-power signal processing applications.

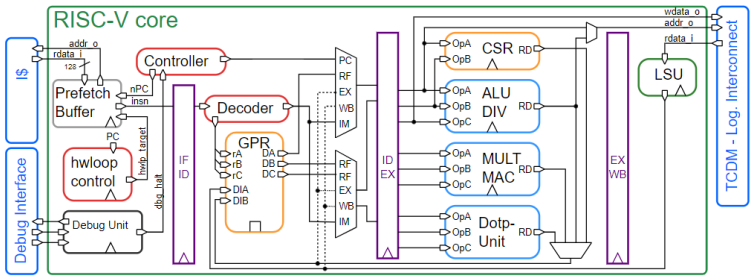


Figure 4.2: CV32E40P core [26]

Ibex[28] , formerly Zero-riscy, is an in-order, single-issue core with 2 pipeline stages. The ibex block diagram is in Figure 4.3.
The processor has full support for the base integer instruction set (RV32I version 2.1) and compressed instructions (RV32C version 2.0). It can be configured to support the multiplication instruction set extension (RV32M version 2.0) and the reduced number of registers extensions (RV32E version 1.9). Ibex implements the Machine ISA version 1.11 and has RISC-V External Debug Support version 0.13.2. Ibex has been designed to met ultra-low-power and ultra-low-area constraints.

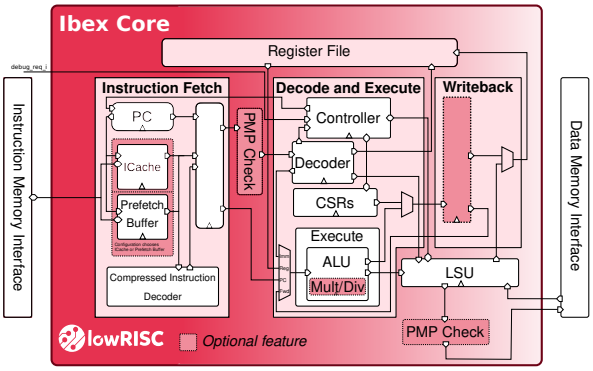


Figure 4.3: Ibex core [28]

The processor selected in this thesis work is the CV32E40P, it is the main 32-bit RISC-V core that finds application in the more general scenario such as signal processing and machine learning. Conversely, the Ibex core is better suited for simpler applications such as control applications.
Nevertheless, the HW accelerator developed in chapter 5 is feasible with Ibex core too since it has an AXI interface.
The system is split into three power domains to optimize power consumption:

- The **always-on domain**, that remains active even when the main processing subsystem is turned off. It includes dedicated hardware components such as the real-time clock and the wake-up logic that are specifically designed to operate in low-power mode and consume minimal energy.
- The **pulp_soc domain**, it is the main soc fabric of all the larger 32-bit PULP chips. It contains the CPU (referred to as the “fabric controller ”), the peripheral subsystem, clock generators, and the main memory.
- The **PULP cluster**, it is the processing subsystem and consists of a cluster of processing elements (PEs) that are tightly coupled together for high-performance parallel processing. The PEs are based on RISC-V architecture and can execute both integer and floating-point operations.

The latter two domains can be switched on and off as needed.

4.3 Trivial application run on PULPissimo

Pulpissimo has been set up following the readme provided on its GitHub page [1], the version used is the "*v7.0.0* ". This version currently only supports the simple runtime and does not support the Software Development Kit (SDK) which is a more complete runtime (drivers, tasks etc.).
The simple runtime is minimal bare-metal runtime, it does not provide any advanced features but all it does is initialize some hardware (FLL, event unit), sets up the stack, and call main.

Build the RTL simulation platform

The PULPissimo design is built upon a 32-bit RISC-V processor and consists of multiple sub-domains or components that work together to achieve the desired performance and energy efficiency.
After cloning the PULPissimo platform from its GitHub repository, the first step to build the RTL simulation platform is to get the latest version of the IPs composing the PULP system. The following command, run in the PULPissimo directory, downloads all the required IPs, solves dependencies, and generates the scripts.

```
$ make checkout
```

The simulation platform then can be built by executing the following commands :

```
$ source setup/vsim.sh
$ make clean build
```

These commands build a version of the simulation platform with no dependencies on external models for peripherals.

Simulation and debug flow

The application to run in PULPissimo is compiled with the riscv-gnu-toolchain [2]. It is the PULP RISC-V C and C++ cross-compiler.
After installing it, it is needed to add the pulp-toolchain binary file to the PATH environment variable.

```
$ export PULP_RISCV_GCC_TOOLCHAIN=<riscv-gnu-toolchain path>
$ export PATH=$PULP_RISCV_GCC_TOOLCHAIN/bin:$PATH
```

The simple runtime supports many different hardware configurations.
These can be found in the *configs* folder of pulp-runtime.
In this thesis has been used PULPissimo with the CV32E40P core, this configuration is selected by executing the command below.

```
$ source pulp-runtime/configs/pulpissimo.sh
```

The *Makefile* that compiles and run the application is "pulp_rt.mk". This makefile is built upon other makefiles that define all the needed targets and configurations, including the application Makefile.

[1] https://github.com/pulp-platform/pulpissimo
[2] https://github.com/pulp-platform/riscv-gnu-toolchain

The makefile hierarchy can be appreciated in figure Figure 4.4.

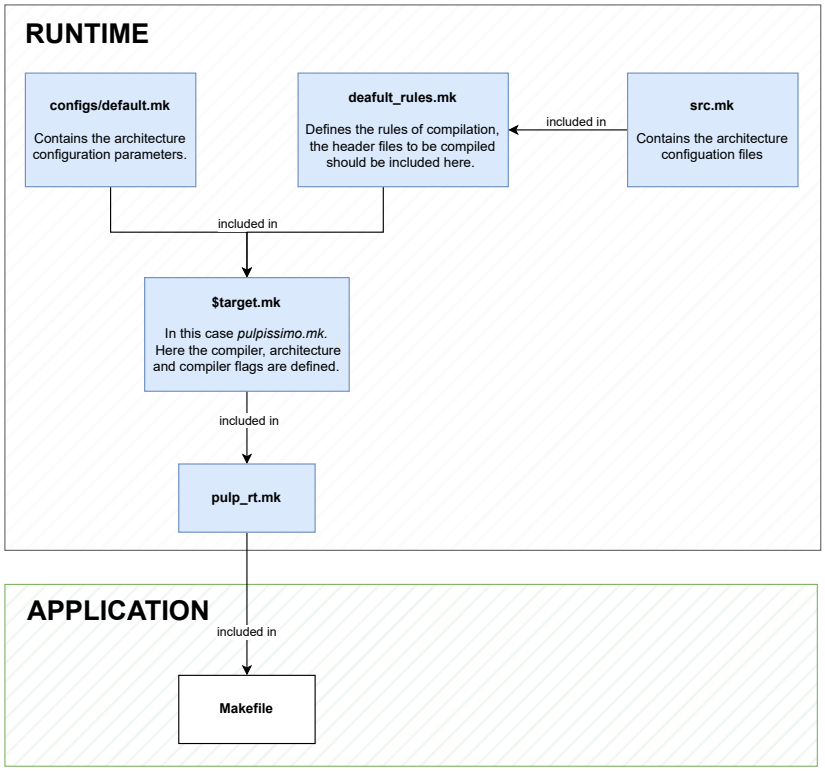


Figure 4.4: Makefile hierarchy

In the application Makefile, reported in listing 2, is included only pulp_rt, since this one contains all the architecture rules to compile correctly the application.

```
PULP_APP = hello
PULP_APP_FC_SRCS = hello.c
PULP_APP_HOST_SRCS = hello.c
PULP_CFLAGS = -O3 -g

include $(PULP_SDK_HOME)/install/rules/pulp_rt.mk
```

Listing 2: Makefile content of the “Hello ”application

In “*pulpissimo.mk* ”are define the systems flags. These have been defined so that are used only the extensions supported by PULPissimo. compile using only the RV32IMC instructions

```
    PULP_LDFLAGS +=
    PULP_CFLAGS += -D__cv32e40p__ -U__riscv__ -UARCHI_CORE_HAS_PULPV2
    PULP_ARCH_LDFLAGS ?= -march=rv32imc -mnohwloop
    PULP_ARCH_LDFLAGS ?= -march=rv32imc -mnohwloop
    PULP_ARCH_OBJDFLAGS ?= -Mmarch=rv32imc
```

Listing 3: System flags defined in *pulpissimo.mk*

The RTL simulation of PULPissimo can be run only on *Questasim* or *Xcelium*.
Lastly, to run the application in the application's directory is run the following command.

```
$ make clean all run
```

This command cleans the previous run, compiles the application, and starts the simulation. If it is all successful the application will prints in stdout "Hello world"and return to the core the value 0.
The simulation can be run in *Modelsim GUI*, which is a useful tool for debugging purposes.

```
$ make clean all run GUI=1
```

The program execution trace is also a useful debugging tool; it is saved in the application's build folder once the application has finished running (file: "trace_core_000003e0.log ").

4.4 Simulating *CRYSTALS-Kyber* algorithm on PULPissimo

This section provides a detailed description of how to run the CRYSTAL-Kyber algorithm on PULPissimo and gives a more complete view of the complexity and challenges found when dealing with complex applications on a limited SoC such as PULPissimo.
The C code of the algorithm is taken from the PQClean repository [3].
PQClean is an effort to collect clean implementations of the post-quantum schemes that are in the NIST post-quantum project. PQClean aims to provide standalone implementations that can easily be integrated into frameworks targeting embedded platforms.
The *pulp-runtime* is not compliant with the standard C library, and the C code of CRYSTALS-Kyber algorithms has to be modified to simulate it on PULPissimo.
The PULP team is working on a prototype that provides "minimal" but standard support, though it is not yet mainstream.
Also, in order to generate a pseudo-random sequence of bytes the "randombytes" file was modified.
To make the CRYSTALS-Kyber algorithm executable on PULPissimo, simply replacing the malloc and free functions with those provided by the PULP team is sufficient.

- void *malloc(size_t size) $\xrightarrow{Replaced \quad with}$ void *pi_l2_malloc(int size);
- void *free(void *ptr) $\xrightarrow{Replaced \quad with}$ void pi_l2_free(void *_chunk, int size);

The *pi_l2_malloc* prototype has the same semantics as the standard C library counterpart, while the *pi_l2_free* differs since it takes also the size of the previously allocated space as an additional parameter.
Once the C code is compliant with the pulp-runtime standard, it is ready to be compiled and run.
In the directory "/pulp-runtime/kernel" has been created a new folder named "kyber "in which all the C-files (test file excluded) that must be compiled to execute the CRYSTALS-Kyber algorithm are moved in.
In "pulp-runtime/include" a new folder called "kyber" has been created. Inside it, all the header files of CRISTALS-Kyber have been moved in.
The only file included in the applications is *pulp.h*, so it must be modified to take into account the added CRYSTALS-Kyber headers. The added files are the ones reported in listing 4.

[3] https://github.com/PQClean/PQClean

```
#include "kyber/api.h"
#include "kyber/cbd.h"
#include "kyber/fips202.h"
#include "kyber/indcpa.h"
#include "kyber/kem.h"
#include "kyber/ntt.h"
#include "kyber/params.h"
#include "kyber/poly.h"
#include "kyber/polyvec.h"
#include "kyber/randombytes.h"
#include "kyber/reduce.h"
#include "kyber/symmetric.h"
#include "kyber/verify.h"
```

Listing 4: Headers files included in pulp.h

All the C file dependencies that need to be compiled to run properly CRYSTALS-Kyber algorithm are added to *pulpissimo.mk*.
In listing 5 are shown the additions done to *pulpissimo.mk* file.

```
#SHA3
#PULP_SRCS     += kernel/SHA3/fips_202.c
#kyber_ip
PULP_SRCS     += kernel/kyber_ip/cbd.c
PULP_SRCS     += kernel/kyber_ip/fips202.c
PULP_SRCS     += kernel/kyber_ip/indcpa.c
PULP_SRCS     += kernel/kyber_ip/kem.c
PULP_SRCS     += kernel/kyber_ip/ntt.c
PULP_SRCS     += kernel/kyber_ip/poly.c
PULP_SRCS     += kernel/kyber_ip/polyvec.c
PULP_SRCS     += kernel/kyber_ip/randombytes.c
PULP_SRCS     += kernel/kyber_ip/reduce.c
PULP_SRCS     += kernel/kyber_ip/symmetric-shake.c
PULP_SRCS     += kernel/kyber_ip/verify.c
```

Listing 5: Source files added to pulpissimo.mk

In *default_rules.mk* is added the following line of code. This line compiles the header file present in the "*Kyber*"directory.

```
    PULP_APP_CFLAGS += -I$(PULPRT_HOME)/sw/kyber
```

These are the changes needed to run the CRYSTALS-Kyber algorithm. Though, if tried to simulate it the simulation will fail. This is due to a stack overflow problem.
The CRYSTALS-Kyber algorithm is made up of multiple functions, each of which performs a specific task. Each time a function is called its local variables are added to the stack, and the stack becomes too full and this causes a problem called a "stack overflow".
In the trace of execution, this error can be noticed at some point during execution.

As shown in the Figure 4.5, is read from the stack a value labeled as “badcab1e ”. This invalid value then disrupts the program execution and leads to the simulation failure.

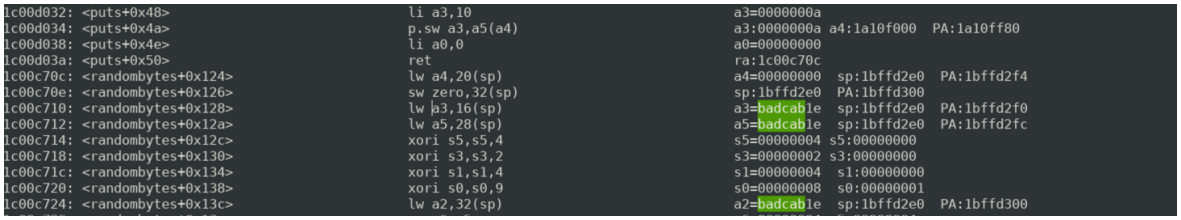


Figure 4.5: Stack overflow error

This problem can be solved in different ways :

- Increasing the stack pointer size: the stack pointer size can be set in file “link.ld ”(file is in directory *pulp-runtime/kernel/chips/PULPissimo/*).
  Increasing the stack size from 0x800 to 0x4000 the algorithm may be run correctly.

```
1     .stack : {
2         . = ALIGN(4);
3         . = ALIGN(16);
4         stack_start = .;
5         . = . + 0x4000;
6         stack = .;
7     } > L2
```

  This solution though consumes more memory resources, which could potentially lead to other memory-related issues if the system runs out of memory.

- Using alternative data structures that do not rely on the stack: the larger arrays can be taken off the stack, either by moving the definitions globally or by adding the keyword ’static’ to the definition.
  For example by adding static to the arrays defined in the main is it possible to reduce the stack to 12KB (i.e., 0x3000 in *link.ld*)
  This second solution may cause problems in the case of reentrant code.

By increasing the stack size the algorithm can then successfully be tested.
The Table 4.2 reports the cycle counts for the key generation, encryption, and decryption algorithms. These algorithms are analyzed for all three levels of security: I (size 512), III (size 768), V (size 1024).

		Cycle counts
Kyber512	KeyGen	1,101,598
	Encaps	1,435,915
	Decaps	1,432,310
Kyber768	KeyGen	1,772,967
	Encaps	2,280,600
	Decaps	2,258,939
Kyber1024	KeyGen	2,767,127
	Encaps	3,383,780
	Decaps	3,352,080

Table 4.2: Cycle counts for RISC-V PULPissimo platform

CHAPTER 5

Integrating Keccak HW accelerator in PULPissimo

This chapter presents a detailed description of how the Keccak core is connected as an external accelerator to PULPissimo [13] and how it is tested.
This is a synthesis of the information obtained following PULP tutorials and documentation and aims to clarify how PULPissimo manages an external accelerator.
The developed architecture is publicly available on GitHub.

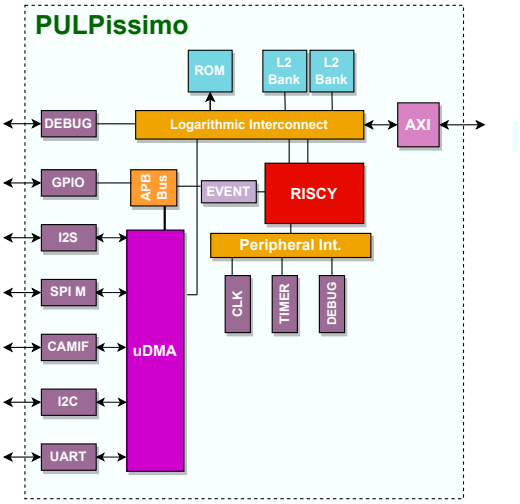


Figure 5.1: PULPissimo block diagram

5.1 General overview

Figure 5.1 shows a simplified block diagram of the PULPissimo SoC.
The AXI plug of the microcontroller has access to the processor bus and can be used to extend the SoC with an accelerator. The accelerator, through the SoC peripheral bus, is attached to the AXI plug and interfaces with SoC using the AXI protocol interface.
Moreover, the accelerator has been modified to be driven in a memory-map fashion style, hence its registers are mapped within the address space of the CPU. The CPU instructions can then access the device by addressing the memory region associated with the accelerator's registers.

The device monitors the CPU's address bus and responds to any CPU access of an address assigned to the device, connecting the data bus to the desired device's hardware register.
Starting from version 2.0.0 of the *pulp_soc* sub-IP of PULPissimo, the interconnect fabric contains a fully parametrizable AXI crossbar switch (called *AXI XBAR*).
The workflow followed to integrate the accelerator is to :

1. Add a new slave port to *AXI XBAR*
2. Map the added port with a memory region and consequently, to implement the modification, a new address rule must be added to SoC interconnect
3. Attach the accelerator to this new dedicate port

When CPU instructions refer to an address within the address range associated with the port, they will be directed through the AXI XBAR to the accelerator.
Figure 5.2 shows the modified SoC interconnect with the port added to *AXI XBAR*. The address rules of the *TCDM* demultiplexer and of the contiguous crossbar are modified such that when the CPU wants to access the accelerator it is directed to the corresponding AXI XBAR port.

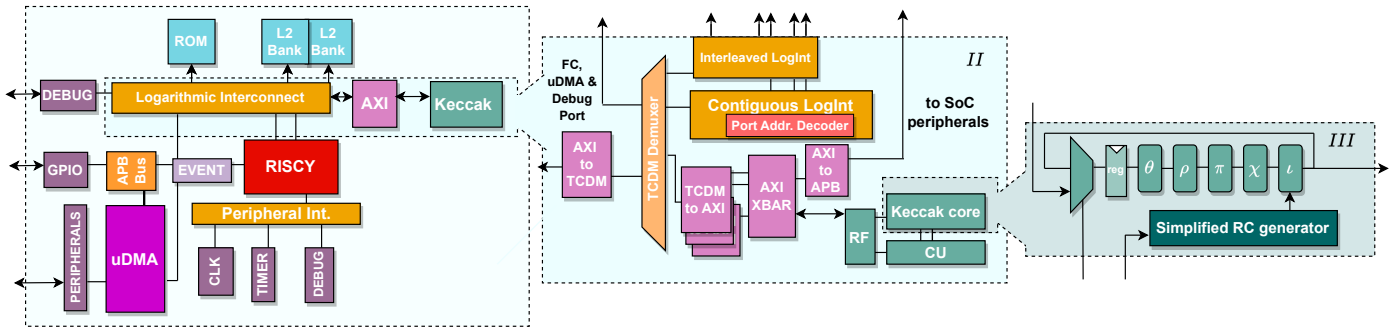


Figure 5.2: Simplified architecture overview [30]

On the left side of the figure, there is a scheme of PULPissimo architecture. Then, a zoomed overview of the logarithmic and AXI interconnections, to better understand where Keccak is connected. Finally, a zoomed of the inner datapath of Keccak core is reported.

5.2 IP register file

5.2.1 register tool

The HW accelerator has a non-standard interface. Therefore, in order to control it through normal load and store operations from the core, it must be first wrapped and connected to a register file.
The configuration and I/O registers of the device wrapper has been constructed using the lowrisc's register file generator "Register tool".
It is run stand-alone using the utility script *regtool.py*, which is a Python 3 tool that takes in input the register file description in *hjson* format and generates the register file in system Verilog.
The register tool can be also used to generate other output formats, such as HTML documentation, standard JSON, compact standard JSON (whitespace removed), and various forms of C header files.
The hjson description of a register file is composed of a first part, where common configurations and signals to all registers are specified. After this common part, then each register is listed with its specific configurations.
Therefore, common signals like the clock and reset are defined at the top level, together with common parameters, like the register width and the bus interface. Then, there's the register field, where the list of all registers is grouped. There, each register is required to be defined with a name, a brief description, and its bit fields.
Optionally, the register can be further described by defining its access permission, reset value, etc ...
The basic description of a register file in hjson looks like the code in listing 6.

```
{
    name: "Keccak_RF",
    clock_primary: "clk_i",
    reset_primary: "rst_ni",
    bus_device: "reg",
    bus_host: "",
    regwidth: "32",
    registers: [
        {   name: "STATUS"
            desc: " Contains status information about the Keccak"
            swaccess: "ro",
            hwaccess: "who",
            hwext: "true",
            fields: [
            { bits: "0", name: "STATUS",
              desc: "If set output of keccack is valid"
            }
            ]
        }

        // ... Here are defined the other registers
```

Listing 6: Hjson description of the Keccak core

The *regtool.py* tool used in this thesis work is a patched version by *ETH* of the lowRISC one. This version uses the "*Generic Register Interface Protocol*"[31] that is simpler than the alternative TileLink Protocol used by the lowRISC version.

The register tool generates the RTL by invoking the regtool.py script with -r flag, as shown by the below command.

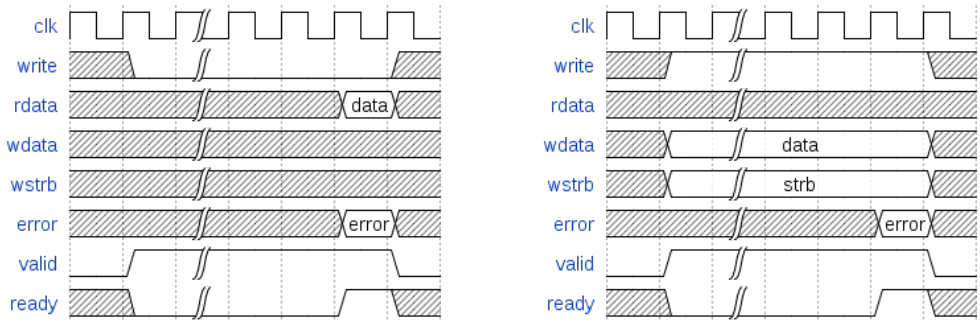
```
$ python3.6 ./regtool.py -r Keccak.hjson --outdir gen_sv/
```

The tool takes in input the file "*Keccak.hjson*", which contains the register description and generates as output two Verilog files in the specified directory "*gen_sv*".
The first created file "*keccak_reg_pkg.sv*" contains auto-generated data structures, which are defined with *SystemVerilog package* that include type definitions for two packed structures that have details of the registers and fields. In particular, all names are converted to lowercase.
These structures are :

- *keccak_reg2hw_t*: contains the signals that are driven from the register module to the rest of the hardware.
- *keccak_hw2reg_t*: contains the signals that are driven from the rest of the hardware to the register module.

The file also contains parameters giving the byte address offsets of the registers. These are prefixed with the peripheral name and converted to uppercase.
The second file "*keccak_reg_top.sv*" contains the module "*keccak_reg_top*" that instantiates the registers. This module provides the register connections to the rest of the hardware.
The default bus fabric control used is the Tielink variant "Uncached Lightweight (TL-UL)" but the register tool has been patched to expose the "generic register interface" protocol.
Transactions of the "*generic register interface*" protocol, as shown in Figure 5.3, consist of only one phase :

- Master sets the address, write, write data and write strobe signals
- Master pulls valid high and waits ready signal, while it waits none of the signals may change
- Transaction completes when both valid and ready are high



(a) Timing waveform of read operation (b) Timing waveform of write operation

Figure 5.3: Read and write with "generic register" protocol

The slave presents the read data and error signals. Moreover, valid must not depend on the read, and slave signals must be constant while valid and ready are both high.
Compared to the Tielink protocol this is much simpler and in addition, the PULP group provides adapters from APB, AXI-Lite, and AXI to the said interface.

5.2.2 Configuration and I/O registers

There are different ways to implement the memory mapping of Keccak. In this work, two of them have been studied in more detail. The first solution is the direct mapping of the Keccak interface, while the second solution is based on a behavior-oriented approach.
The former requires fewer registers to memory map the accelerator: it is not straightforward to implement and due to the added complexity of managing, it makes it not worth it. On the other hand, the latter solution is easier to be implemented and offers a wider scope for optimization.

- **Solution 1**

The PULPissimo data bus width is 32 bits, while the Keccak state works on a data block of 64 bits (*D_in* and *D_out* are 64 bits width). Therefore the straightforward mapping of the Keccak interface is the follow.

- Two 32-bit "I/O registers" of type read-write where CPU writes *D_in* and reads *D_out*.
- Single 32-bit "control register" of type write-only where CPU writes *Din_valid*, *last_block* signals.
- Single 32-bit "status register" of type read-only where the CPU reads Keccak status (*ready* signal value) and *D_out_valid* signal.

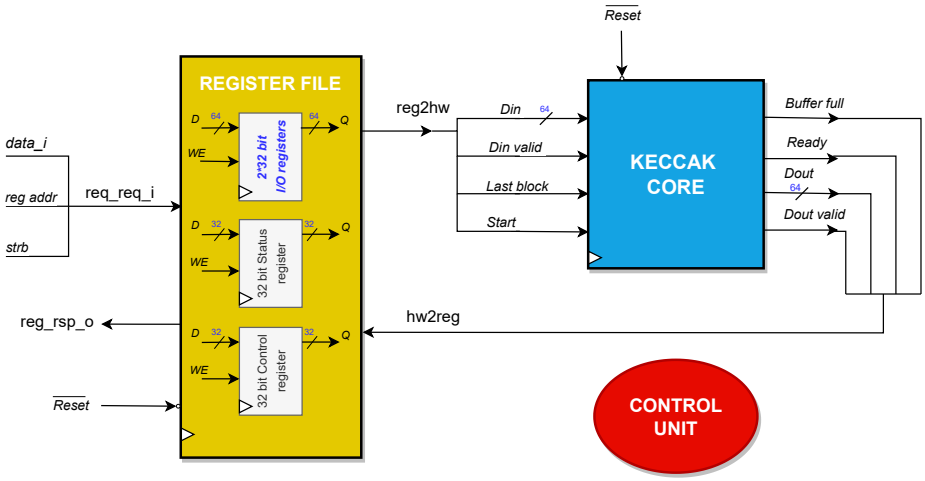


Figure 5.4: Register mapping solution 1

Using the mapping shown in Figure 5.4, Keccak waits for the start signal and then expects to read a new 64-bit input on each new cycle for 25 cycles. When the permutation is complete, it unpacks the 1600 bits of the result into 64-bit chunks. Consequently, the Keccak driver must be managed in a way that meet these synchronization requirements.

- **Solution 2**

Keccak needs at most a 1600-bit input to perform the permutation.
The accelerator can work stand-alone if it has a 1600-bit width buffer. The CPU just loads all input value in the register file and then let the accelerator perform the computation without taking any further action.
The connections with this solution are illustrated in Figure 5.5, therefore the the register file is composed of :

- Fifty registers 32-bits width for I/O (for a total of 1600 bits) of type read-write, the CPU loads the input vector and reads the output result from these.
- Single 32-bit "control register" of type write-only where CPU, after loading the input value, writes to start the permutation.
- Single 32-bit "status register" of type read-only where CPU reads when the IP has finished the permutation.

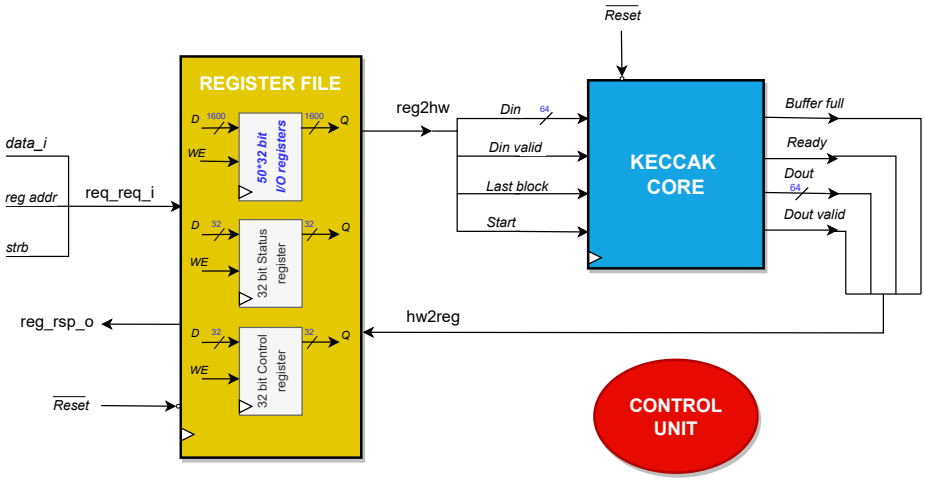


Figure 5.5: Register mapping solution 2

At first glance, the first solution seems the better one, since is the one requiring fewer registers and so contributes to less area overhead. However, to make it work, the IP must be re-designed. Therefore, a more complex driver is needed.
There are a few problems to take into account when dealing with this solution. In particular:

1. When *Din_valid* is set to 1, Keccak expects to read a 64 bits input, but the data bus is on 32-bit.
   With each new input, the driver should first stall the Keccak, then loads the 64-bit in the I/O registers, and finally write *Din_valid* to 1. In the subsequent cycle, the driver must stall again the Keccak and set *Din_valid* to 0.
2. When permutation is finished Keccak pulls up *Dout_valid* and keeps it high for 25 cycles, at each new cycle a 64-bit chunk of the output result is given on *D_out*.
   As said before, the data bus is 32 bits so the CPU cannot read a new output value at each cycle.

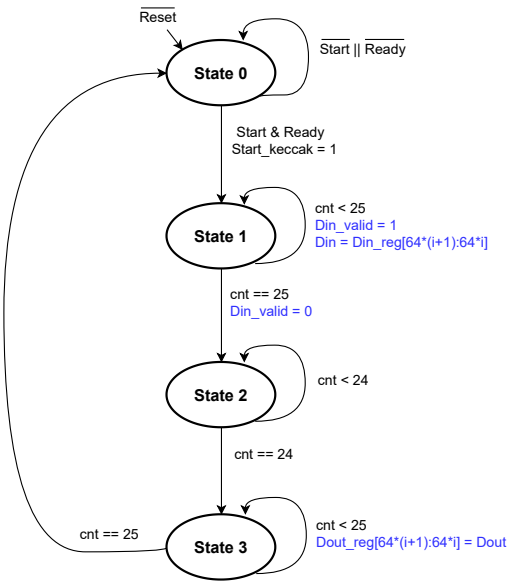
The first problem and its strict synchronization requirements cannot be achieved by writing the driver in C. Indeed, these time-sensitive operations can be guaranteed only in assembly and by setting the compiler options to not apply optimizations to the Keccak driver, like instruction reordering, which would otherwise cause an error.
The second problem can be managed by modifying the Keccak architecture so that between each new *D_out* value it stalls and waits that the CPU correctly reads the result value.
This solution, due to its high complexity, requires a fine-tuning of the driver and the Keccak architecture, and even if all these problems were managed, it still determines a heavy performance degradation. Also, if the CPU is stopped by an interrupt while in the process of driving the accelerator, then the result of the permutation would be wrong. Since this solution overall needs too much care, from the driver to the *interrupt service routine (ISR)*, it has been discarded.

The second solution instead simplifies the CPU work and gives better performance. The higher complexity added to the Keccak core simplifies the driver needed to interact with the IP.
The small developed driver for the second solution is written in C and it simply loads the 1600-bit input, starts the accelerator, and then waits for the end of the permutation (see section 5.6).
This solution also is preferred since there's ample room for optimization with the added register file to the Keccak architecture.
Exploiting the added register file, as will be seen in section 5.8, all of the Keccak internal registers can be eliminated and the resultant architecture is stripped down, **achieving an almost null area overhead**.

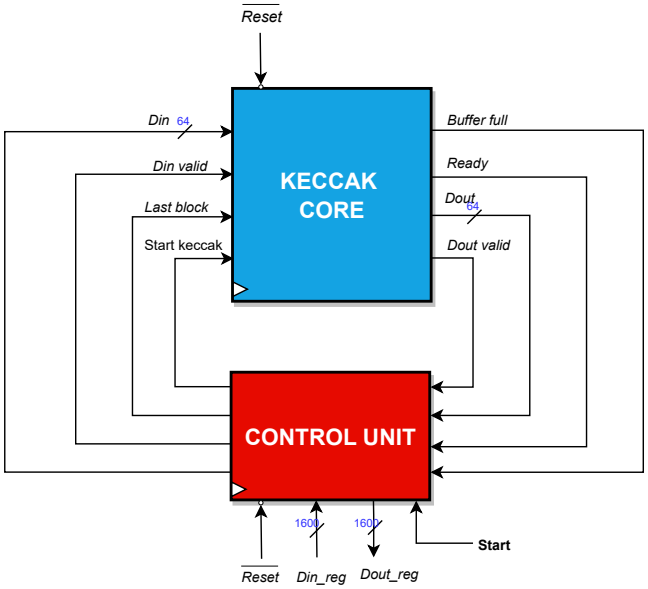
5.3 Control unit

To further reduce the CPU workload and to make the Keccak accelerator more autonomous a control unit has been added to the IP.
The control unit's job is to fetch input from the register file, wait till Keccak finishes the permutation, and lastly collect the 1600-bit output result.
The control unit is designed as a Finite State Machine (FSM) and is composed of four states :

- **State 0**: the CU waits for the start signal from the CPU. When it is received, then the CU checks if Keccak is ready to perform a new permutation. If Keccak is ready, then will set the *start_keccak* signal high and will go to state 1; otherwise, it remains in this state.
- **State 1**: it lasts 25 cycles. The CU set *Din_valid* high and at each cycle fetches a new input from the register file and feeds it to Keccak.
- **State 2**: here the CU waits for the Keccak to finish the permutation. The permutation requires 24 cycles and Keccak signals its end by setting *Dout_valid* high.
- State 3: in this state the CU collects the 1600-bit output vector. Since the Keccak gives a 64-bit chunk of it at each new cycle on *D_out* this state lasts for 25 cycles.

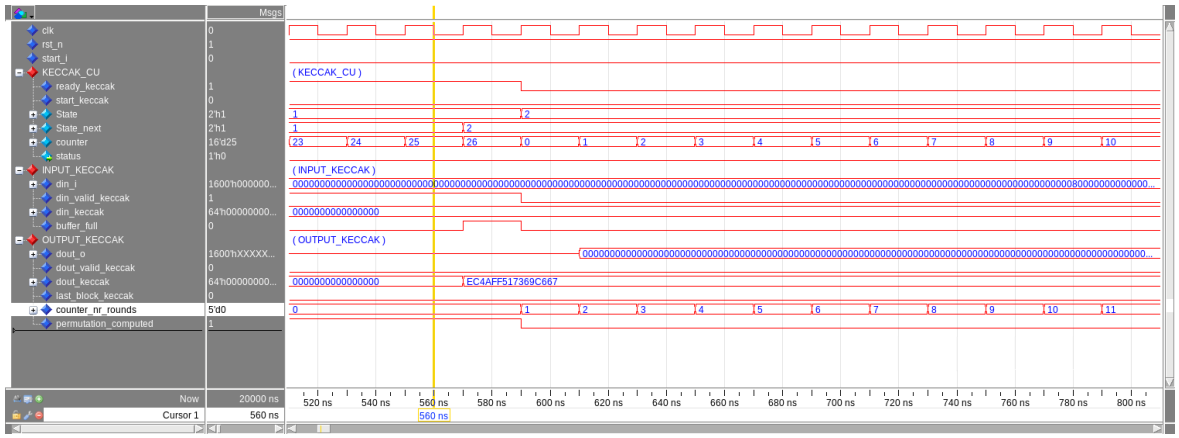


(a) State diagram of the control unit

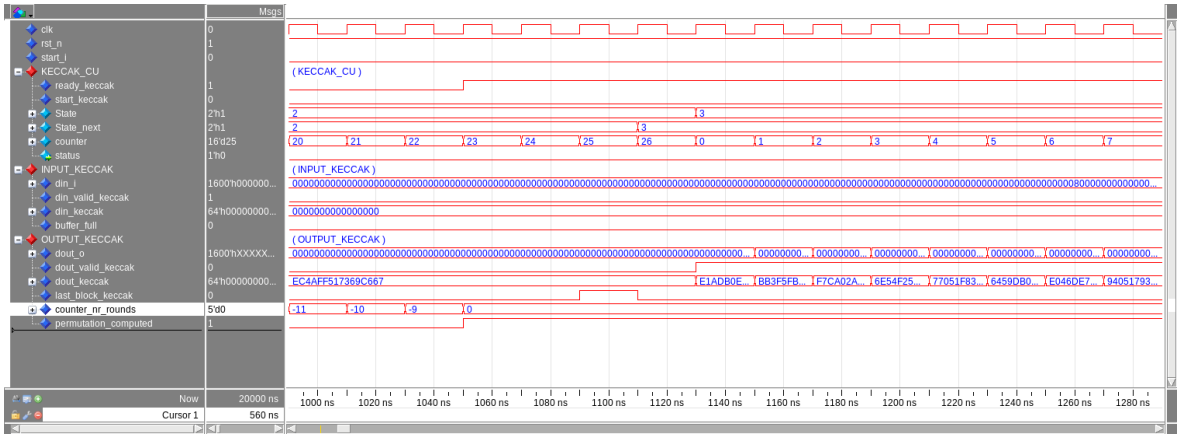


(b) Connection between the CU and the accelerator

The new architecture called "*Keccak_core*" has been tested with Questasim. The test can be reproduced by running the shell script called "*init_sim.sh*", which can be found in the appendix Listing A.1.
This shell file can be run with an optional boolean parameter that selects whether to run or not the simulation in GUI mode. It starts Questasim and executes the TCL script "*Keccak.tcl*", this TCL file is the one that compiles all the Keccak components, sets up configurations and wave window, and then runs the simulation.



(a) Start of the Keccak accelerator



(b) Store of output value in registers

5.4 Keccak wrapper

As mentioned in section 5.2, the IP cannot be controlled as it is with normal loads and stores from the CPU and has to be first wrapped by an AXI interface.
Indeed, the generated register file has a "generic register interface". Therefore, a protocol converter from AXI to the just-mentioned interface is needed.
The PULP group provides this protocol converter, as well as others, in the **register_interface** repository on GitHub [1].
The provided protocol converter called "*axi_to_reg_intf*" uses structs instead of SystemVerilog Interface. The use of structs is preferred in favor of interfaces since they are more suitable with EDA tools. To ease the handling of these structs the PULP group provides pre-defined macros, which are still in the *register_interface* repository. In the file "*typedef.svh*" macros to create the register bus struct

[1] https://github.com/pulp-platform/register_interface

(*REG_BUS*) are defined, while macros to assign the register bus request/response to the *REG_BUS* struct are defined in the file "*assign.svh*".
The designed Keccak wrapper is defined in the file "*Keccak_top.sv*"and its top-entity is in listing 7, as can be observed in figure Figure 5.9 the top entity of this module has the following portlist.

⇒ clk_i, rst_ni, test_mode_i

⇒ A single AXI slave port

The AXI port is an *AXI_BUS.Slave* SystemVerilog Interface (part of the AXI IP). The AXI bus is a high-performance scalable bus system that makes it easy to connect and communicate between different components in an SoC.
The AXI bus specification defines a standard interface for communication between components, allowing components from different vendors to be easily combined in a single SoC.
The AXI bus specification defines several channels or signals to communicate between the bus master and bus slave components in an SoC.
The main channels are the following.

- AW (Address Write) Channel: This channel carries the write address and write control signals from the bus master to the bus slave. It is used to initiate a write transaction.
- W (Write Data) Channel: This channel carries the write data from the bus master to the bus slave. It is used to transfer the data to be written to the bus slave's memory.
- B (Write Response) Channel: This channel carries the response from the bus slave to the bus master for a write transaction. It confirms the completion of the write transaction and provides status information.
- AR (Address Read) Channel: This channel carries the read address and read control signals from the bus master to the bus slave. It is used to initiate a read transaction.
- R (Read Data) Channel: This channel carries the read data from the bus slave to the bus master. It is used to transfer the data read from the bus slave's memory.

```
1  module keccak_top
2  #(
3      parameter int unsigned
4      AXI_ADDR_WIDTH = 32,
5      localparam int unsigned
6      AXI_DATA_WIDTH = 32,
7      parameter int unsigned
8      AXI_ID_WIDTH,
9      parameter int unsigned
10     AXI_USER_WIDTH
11 )
12 (
13     input logic clk_i,
14     input logic rst_ni,
15     input logic test_mode_i,
16     AXI_BUS.Slave axi_slave
17 );
```

Listing 7: Top entity of the Keccak wrapper

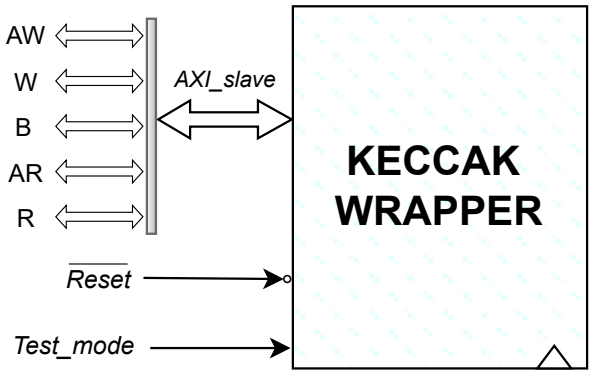


Figure 5.8: Keccak wrapper interface

The AXI port goes in input to the "*axi_to_reg_intf*"converter, the output of this converter is a signal of type *REG_BUS* called "*reg_o*". The converter instantiation is shown in listing 8.

```
    axi_to_reg_intf #(
        .ADDR_WIDTH(AXI_ADDR_WIDTH),
        .DATA_WIDTH(AXI_DATA_WIDTH),
        .ID_WIDTH(AXI_ID_WIDTH),
        .USER_WIDTH(AXI_USER_WIDTH),
        .DECOUPLE_W(0)
    ) i_axi2reg (
        .clk_i,
        .rst_ni,
        .testmode_i(test_mode_i),
        .in(axi_slave),
        .reg_o(axi_to_regfile)
    );
```

Listing 8: Converter instantiation

As mentioned before, the auto-generated register file uses structs signals at the interface.
The output of the converter "*reg_o*"is a SystemVerilog Interface, and in order to be connected to registers it must be first converted to a struct signal.
The signal "*reg_o*"is first connected to a new wiring signal called "*axi_to_reg_file*"of type SystemVerilog interface. The snippet of code in listing 9 does the conversion from the *REG_BUS* interface to the struct signal.

```
    typedef logic [AXI_DATA_WIDTH-1:0] data_t;
    typedef logic [AXI_ADDR_WIDTH-1:0] addr_t;
    typedef logic [AXI_DATA_WIDTH/8-1:0] strb_t
    `REG_BUS_TYPEDEF_REQ(reg_req_t, addr_t, data_t, strb_t)
    `REG_BUS_TYPEDEF_RSP(reg_rsp_t, data_t);
    reg_req_t to_reg_file_req;
    reg_rsp_t from_reg_file_rsp;
    `REG_BUS_ASSIGN_TO_REQ(to_reg_file_req, axi_to_reg_file)
    `REG_BUS_ASSIGN_FROM_RSP(axi_to_reg_file, from_reg_file_rsp)
    Keccak_reg2hw_t reg_file_to_ip;
    Keccak_hw2reg_t ip_to_reg_file;
```

Listing 9: Conversion from REG_BUS to $axi\_to\_reg\_file$

Macros "*REG_BUS_TYPEDEF*"are used to declare the structs data type "*reg_req_t* "and "*reg_rsp_t*". These structs are compatible with the one used at the interface by the register file.
The conversion of a struct signal to the SystemVerilog interface *REG_BUS* and vice-versa is done with "*REG_BUS_ASSIGN*"macros.
The structs data type "*Keccak_reg2hw_t*"and "*Keccak_hw2reg_t*"are imported from file "keccak_reg_pkg.sv ". This file has been generated by the register tool in section 5.2, and is used to declare the wiring signals that connect the register file and the Keccak IP.
The register file can then be instantiated as shown in listing 10.

```
keccak_reg_top #(
    .reg_req_t(reg_req_t),
    .reg_rsp_t(reg_rsp_t)
) i_regfile (
    .rst_ni,
    .devmode_i(1'b1),
    // From the protocol converters to regfile
    .reg_req_i(to_reg_file_req),
    .reg_rsp_o(from_reg_file_rsp),
    // Signals to keccak IP
    .reg2hw(reg_file_to_ip),
    .hw2reg(ip_to_reg_file)
);
```

Listing 10: Instantiation of Keccak's register file

The *devmode_i* signal of the *register_file* is set to 1'b1.
Then the Keccak IP and the control unit can be instantiated by adding the two structs data type auto-generated by the register which contains sub-fields for each register in the hjson description.
Those register fields contain sub-fields for the individual bit-fields of the registers.
Those bitfields of the register file are the ones that connect to the corresponding port of the Keccak IP, see listing 11.

```
    wire logic [63:0] din_keccak, dout_keccak;
    wire logic ready_keccak, dout_valid_keccak, start_keccak,
               last_block_keccak, din_valid_keccak;

     keccak_cu i_keccak_cu (
        .clk_i(clk_i),
        .rst_ni(rst_ni),
        .start_i(reg_file_to_ip.ctrl.q & reg_file_to_ip.ctrl.qe),
        .ready_keccak_i(ready_keccak),
        .din_i(reg_file_to_ip.din),
        .dout_keccak_i(dout_keccak),
        .dout_valid_keccak_i(dout_valid_keccak),
        .start_keccak_o(start_keccak),
        .last_block_keccak_o(last_block_keccak),
        .dout_o(ip_to_reg_file.dout),
        .din_keccak_o(din_keccak),
        .din_valid_keccak_o(din_valid_keccak),
        .status(ip_to_reg_file.status)
    );

    keccak i_keccak (
        .clk(clk_i),
        .rst_n(rst_ni),
        .start(start_keccak),
        .din(din_keccak),
```

```
26          .din_valid(din_valid_keccak),
27          .buffer_full(),
28          .last_block(last_block_keccak),
29          .ready(ready_keccak),
30          .dout(dout_keccak),
31          .dout_valid(dout_valid_keccak)
32      );
```

Listing 11: Keccak and control unit instantiation

The resultant Keccak IP block diagram is shown in figure Figure 5.9

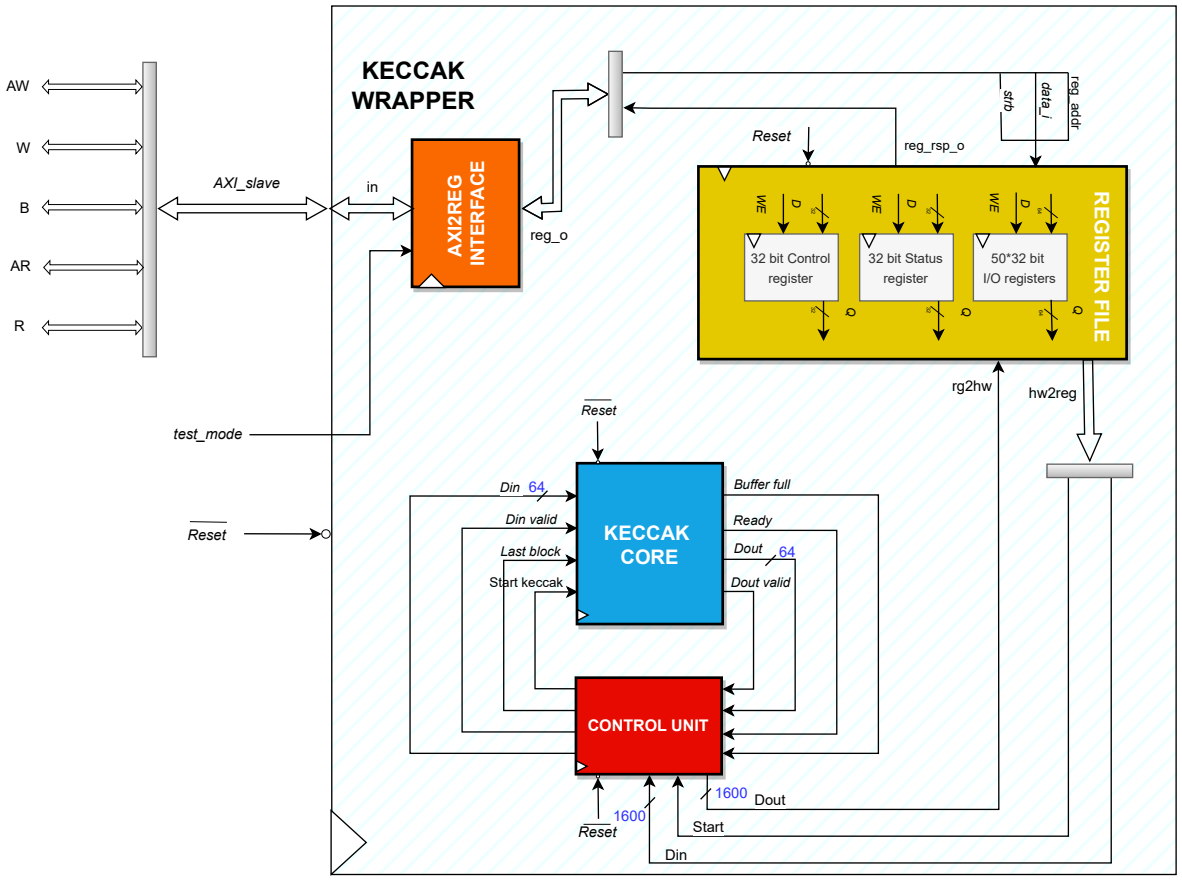


Figure 5.9: Keccak IP block diagram

5.5 Integrating Keccak IP in PULPissimo

Once the IP is wrapped with an AXI interface it is ready to be connected to *pulp_soc*.
Dependencies between IPs in PULPissimo are managed with Bender.[32] How it is done will be better clarified in subsection subsection 5.5.2.
Bender checkouts the sub-ips of PULPissimo in a hidden directory called "*.bender/git/checkouts*", when wanting to modify an IP it should not be done inside this directory.

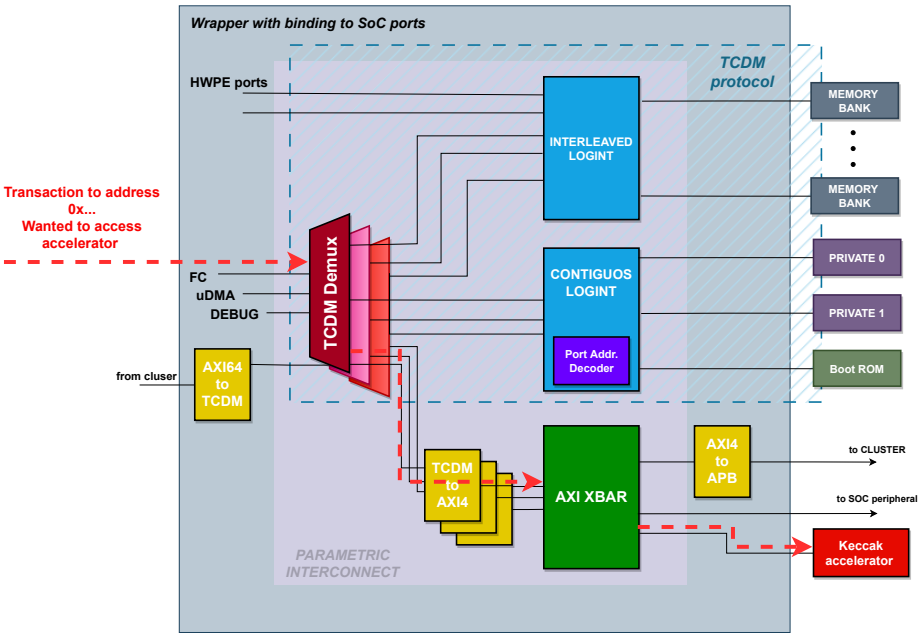


Figure 5.10: Modified SoC interconnect [33]

5.5.1 Adding Keccak to PULPissimo RTL

The workflow followed in order to modify the *pulp_soc* is to first create a working copy of this IP, this can be done by invoking, in the top-level directory of PULPissimo, the following command.

```
$ ./bender clone pulp_soc
```

This checks out the working copy of the IP into a directory, whose default one is "*working_dir*"but can be overridden with the flag " $-path < DIR >$".
If instead wanted to use your customized version of the pulp_soc, is needed to modify the PULPissimo manifest file (Bender.yml) so that it points to the custom version instead of the default one. This was the approach taken during the thesis work, indeed the pulp_soc has been customized to integrate the accelerator and PULPissimo refers to this patched pulp_soc. The customized pulp_soc is available as an open-source project on GitHub [2].
The integration of the Keccak IP in the RTL of PULPissimo is done by incrementing the number of slave ports of the parametric AXI crossbar, connecting Keccak to the added port, and then memory mapping the added port by associating it with an address range.
The SoC interconnect module is modified accordingly and a new port dedicated to Keccak IP is added

[2] https://github.com/aledolme/pulpissimo

to the SoC portlist.
The change to the address space is integrated by reserving a memory region for the peripheral and adding a new address rule to SoC interconnect.

In the fabric interconnect(file soc_interconnect_wrap.sv**)**

The implementation steps followed are reported below.

1) Added a new port to SoC interconnect that connects the AXI plug with Keccak IP (see listing 12).

```
module soc_interconnect_wrap
import pkg_soc_interconnect::addr_map_rule_t;
(
   input logic clk_i,
   input logic rst_ni,
   ...
   AXI_BUS.Master         Keccak_slave  // Used to communicate with Keccak
);
```

Listing 12: AXI port added to SoC interconnect

2) The AXI port created is assigned to the axi_slaves interfaces array (check listing 13 for reference).

By default, there are 2 AXI slaves, one for the *s_data_out_bus* that's used to attach a cluster (not in use for PULPissimo) and one that is converted to APB to communicate with the peripherals within the uDMA domain and the FLL.
The number of slave ports is incremented from 2 to 3 by modifying the *axi_slaves* array, the new entry is then assigned to the added port using the *AXI_ASSIGN* macro.

```
AXI_BUS #(.AXI_ADDR_WIDTH(32),
          .AXI_DATA_WIDTH(32),
          .AXI_ID_WIDTH(pkg_soc_interconnect::AXI_ID_OUT_WIDTH),
          .AXI_USER_WIDTH(AXI_USER_WIDTH)
          ) axi_slaves[3]();
`AXI_ASSIGN(axi_slave_plug, axi_slaves[0])
`AXI_ASSIGN(axi_to_axi_lite_bridge, axi_slaves[1])
// Assigned added AXI\_BUS to keccak slave port
`AXI_ASSIGN(Keccak_slave, axi_slaves[2])
```

Listing 13: Axi_slaves interfaces array

3) The parameters of "soc_interconnect"and the size of the corresponding Interface Array is modified to contain an additional AXI slave port, refer to listing 14.

```
soc_interconnect #(
    // SoC interconnect parameters
    .NR_MASTER_PORTS(pkg_soc_interconnect::NR_TCDM_MASTER_PORTS),
    ...
```

```
5
6           .NR_AXI_SLAVE_PORTS(3), // 1 for AXI to cluster,
7                                   // 1 for SoC peripherals (converted to APB),
8                                   // 1 for Keccak IP
9           ...
10      ) i_soc_interconnect (
11          .clk_i,
12          .rst_ni,
13          ...
14          .axi_slaves(axi_slaves)
```

Listing 14: Modification to Interface Array in $soc_interconnect$

4) The address rule of the added peripheral port is defined in the $AXI\_CROSS\_BAR\_RULES$

If looked to figure Figure 5.10 to access the device the CPU transaction should be routed by the DEMUX to port 0 (the TCDM) and then should be routed by AXI XBAR to the keccak slave port. The code modification is reported in listing 15.
The implementation should normally consist of adding a new address rule to L2_Demux and to AXI XBAR. Although when a transaction to DEMUX does not match any rule, it is by default routed to port 0 (the TCDM to AXI4, AXI XBAR) and so is not needed to add it.

```
1       localparam NR_RULES_AXI_CROSSBAR = 3;
2       localparam addr_map_rule_t [NR_RULES_AXI_CROSSBAR-1:0] AXI_CROSSBAR_RULES = '{
3           '{  idx: 0,
4               start_addr: `SOC_MEM_MAP_AXI_PLUG_START_ADDR,
5               end_addr: `SOC_MEM_MAP_AXI_PLUG_END_ADDR },
6           '{  idx: 1,
7               start_addr: `SOC_MEM_MAP_PERIPHERALS_START_ADDR,
8               end_addr: `SOC_MEM_MAP_PERIPHERALS_END_ADDR },
9           '{  idx: 2,
10              start_addr: `SOC_MEM_MAP_Keccak_START_ADDR,
11              end_addr: `SOC_MEM_MAP_Keccak_END_ADDR }
12      };
```

Listing 15: New address rule in AXI_XBAR

The memory region Macros used to delimit the added address rule are defined in the file *soc_mem_map.svh*, the memory region assigned to Keccak peripheral is reported in listing 16.
In this file are defined all the CPU address space regions, like for example the start/end addresses for the $L2\_MEMORY$ region.

```
1       -- Memory region Keccak
2       `define SOC_MEM_MAP_Keccak_START_ADDR        32'h1A40_0000
3       `define SOC_MEM_MAP_Keccak_END_ADDR          32'h1A40_1000
```

Listing 16: Memory region assigned to the Keccak peripheral

In the top level of pulp_soc (file *pulp_soc.sv*)

Lastly, the added port to AXI_XBAR is connected to Keccak IP

1) Created a new AXI_BUS SystemVerilog interface signal (listing 17).

```
// new axi bus interface
AXI_BUS #(
    .AXI_ADDR_WIDTH(AXI_ADDR_WIDTH),
    .AXI_DATA_WIDTH(AXI_DATA_OUT_WIDTH),
    .AXI_ID_WIDTH(AXI_ID_OUT_WIDTH),
    .AXI_USER_WIDTH(AXI_USER_WIDTH)
) s_Keccak_bus();
```

Listing 17: Added AXI slave interface

This wiring signal connects the port added to the *soc_interconnect_wrap* with the Keccak IP

2) The Keccak wrapper is instantiated and its AXI slave port is connected to *s_Keccak_bus* signal (listing 18).

```
// Instatited Keccak wrapper
Keccak_top #(
    .AXI_ADDR_WIDTH(AXI_ADDR_WIDTH
    .AXI_ID_WIDTH(AXI_ID_OUT_WIDTH),
    .AXI_USER_WIDTH(AXI_USER_WIDTH)
) i_Keccak_top (
    .clk_i(s_soc_clk),
    .rst_ni(s_soc_rstn),
    .test_mode_i(dft_test_mode_i),
    .axi_slave(s_Keccak_bus)
);
```

Listing 18: Keccak wrapper instantiation

3) The wiring bus signals lastly is connected to SoC interconnect (listing 19).

```
soc_interconnect_wrap #(
  .NR_HWPE_PORTS(NB_HWPE_PORTS),
  .NR_L2_PORTS(NB_L2_BANKS),
  .AXI_IN_ID_WIDTH(AXI_ID_IN_WIDTH),
  .AXI_USER_WIDTH(AXI_USER_WIDTH)
) i_soc_interconnect_wrap (
    .clk_i                ( s_soc_clk      ),
    .rst_ni               ( s_soc_rstn     ),
    ...
    .Keccak_slave         ( s_Keccak_bus   )
);
```

Listing 19: Keccak wrapper attach to SoC interconnect

5.5.2 Updating PULPissimo dependencies

Dependency management in PULPissimo is done with Bender [32]. It is a command line tool used in hardware design projects that provides a simple way to define dependencies among IPs, execute unit tests, and verify that the source files are valid input for various simulation and synthesis tools.
Dependency resolution is performed according to a manifest file called "*Bender.yml*". This file lists all source files of the RTL project as well as their direct dependencies.
Once all the dependencies have been successfully resolved, Bender generates the lock file "*Bender.lock* "which contains the exact revision of each dependency and may be put under version control to allow for reproducible builds.
The source files generated by Bender can be used in various tools for simulation, ASIC/FPGA synthesis, etc ...
An optional file that can be present is the "*Bender.local* "which contains local configuration overrides (ignored in version control, i.e. added to .gitignore).
When cloned an IP (command *./bender clone* < *ip_name* >) the linking to this local variant is specified in the Bender.local file.
When adding a new ip to PULPissimo, as in this thesis work, the *pulp_soc* sub-ip is forked since is the main repository that contains most of the SoC RTL logic.
The Keccak IP has been added to the source tree of *pulp_soc*, inside its "rtl "directory, and dependency to *pulp_soc* has been registered by adding it to the manifest Bender.yml of *pulp_soc*.
However, since the Keccak IP uses a patched version of the "*register file*" [31] then this IP has been copied inside the *pulp_soc* directory too and its dependency is registered in the dependencies section of *pulp_soc*'s Bender.yml

```
# Bender.yml of pulp_soc
dependencies :
  register_interface:     { path: "./register_interface"}
  # ... here listed other dependencies

sources:
  # keccak ip
  - files:
      - rtl/keccak_ip/gen_sv/keccak_reg_pkg.sv
      - rtl/keccak_ip/gen_sv/keccak_reg_top.sv
      - rtl/keccak_ip/keccak_globals.vhd
      - rtl/keccak_ip/Keccak_2to1mux.vhd
      - rtl/keccak_ip/Keccak_REG_rst_n.vhd
      - rtl/keccak_ip/reg_en_rst_n.vhd
      - rtl/keccak_ip/keccak_round_constants_gen.vhd
      - rtl/keccak_ip/keccak_round.vhd
      - rtl/keccak_ip/keccak_buffer.vhd
      - rtl/keccak_ip/keccak_cu.sv
      - rtl/keccak_ip/keccak_datapath.vhd
      - rtl/keccak_ip/keccak.vhd
      - rtl/keccak_ip/keccak_top.sv
    # ... here listed other source files
```

Listing 20: Updates to Bender manifest of pulp_soc

The source files of Keccak are listed in the sources section of the package manifest, as can be observed in listing 20. The order in which the source files are listed is hierarchical (similar to how to compile source files in Questasim).
When the list of dependencies is modified to make the update effective the following command is run.

```
$ .\bender update
```

This re-resolves the dependency versions and recreates the Bender.lock file.
If added flag *–fetch* then all git dependencies are re-fetched from their corresponding URLs.
To update the source files in the Makefiles, the script command is called.

```
$ make scripts
```

This will parse the Bender.yml to generate tcl scripts for all the IPs used in the PULPissimo project, incorporating the changes of this working copy into the RTL simulation model.
These changes though have an effect locally, indeed in version control the Bender.lock file is ignored.
The strategy suggested by the PULP group is to first develop and test the added modification locally within this working copy of *pulp_soc*, then once all is verified the patched *pulp_soc* directory is added to GitHub[3] and the manifest of PULPissimo is modified so that it refers to own IP.
As previously explained, to make the changes effective "*./bender update*"must be run in the top-level directory of PULPissimo.
The RTL simulation platform is rebuilt executing n the top-level of PULPissimo the following commands.

```
# The command chechout downloads all the required IPs, solves
# dependencies and generates the scripts.
$ make checkout
# The TCL scripts are regenerated due to the added IP.
$ source setup/vsim.sh
$ make clean build
```

5.6 Keccak drivers

Now that the IP is integrated into PULPissimo,a driver is needed to actually use it.
The accelerator is memory mapped. Therefore the CPU can access the device by exchanging data with the accelerator's register file.
The addresses of Keccak's registers can be auto-generated using the *register tool*.

```
$ Python3.9 ./regtool.py --cdefined Keccak.hjson > Keccak_auto.h
```

It takes in input the same register description in hjson used in subsection 5.2.1 and generates a C header file called "Keccak_auto.h ".
Inside the generated file register addresses are provided. Here single-bit fields have a define with their bit offset, while multi-bit fields have a define for the bit offset, a mask and may have defines giving the enumerated names and values.
The snippet of code in listing 21 shows the definitions of Din, Control, and Status registers.

[3] https://github.com/aledolme/pulpissimo

```
// starting address of keccak regs file
#define KECCAK0_BASE_ADDR 0x1a400000
// auto added parameter
#define KECCAK_PARAM_DIN 50

// auto added parameter
#define KECCAK_PARAM_DOUT 50

// Register width
#define KECCAK_PARAM_REG_WIDTH 32

// Common Interrupt Offsets

// Subword of input of Keccak module (common parameters)
#define KECCAK_DIN_DIN_FIELD_WIDTH 32
#define KECCAK_DIN_DIN_FIELDS_PER_REG 1
#define KECCAK_DIN_MULTIREG_COUNT 50

// Subword of input of Keccak module
#define KECCAK_DIN_0(id) (KECCAK##id##_BASE_ADDR + 0x0)
#define KECCAK_DIN_0_REG_OFFSET 0x0

// Subword of input of Keccak module
#define KECCAK_DIN_1(id) (KECCAK##id##_BASE_ADDR + 0x4)
#define KECCAK_DIN_1_REG_OFFSET 0x4

{ ... Here define all the 50 reg for Din }

//  Contains control signals to drive the Keccak
#define KECCAK_CTRL(id) (KECCAK##id##_BASE_ADDR + 0x190)
#define KECCAK_CTRL_REG_OFFSET 0x190
#define KECCAK_CTRL_START 0

//  Contains status information about the Keccak
#define KECCAK_STATUS(id) (KECCAK##id##_BASE_ADDR + 0x194)
#define KECCAK_STATUS_REG_OFFSET 0x194
#define KECCAK_STATUS_STATUS 0
```

Listing 21: Autogenerated Keccak's register file addresses

The format in which registers' addresses are defined assumes that there is a base address "$Keccak < id > \_BASE\_ADDR$", where id is an identifying number to allow for multiple instantiations of the peripheral.
The base address "$Keccak0\_BASE\_ADDR$" is the starting address of the memory region assigned to the Keccak peripheral.
The driver implementation is written in the file "*Keccak_driver.c*", the header file associated is called "*Keccak_driver.h*".
The function that performs the permutation using the Keccak accelerator is called "*KeccakF1600_StatePermute*" which takes in input two 1600-bit vectors (vector of 50 entries, each

entry is 32-bit). The former vector is the one containing the input value, while the latter is the one where the output result is stored. The code is in listing 22.

```
#include <stdint.h>
#include <stdio.h>
#include <keccak_auto.h>

void KeccakF1600_StatePermute(uint32_t Din[50], uint32_t Dout[50] )
{
    set_input(Din);
    trigger_keccak();
    poll_done();
    get_result(Dout);
}
```

Listing 22: Keccak driver

So the steps executed by the driver are :

1. set_input(Din): Writes input value in the Keccak's I/O registers (listing 23).

```
    void set_input(uint32_t* Din){
        uint32_t volatile *Din_reg_start = (uint32_t*)KECCAK_DIN_0(0);
        for (int i = 0; i<50; i++)
        {
            Din_reg_start[i] = Din[i];
        }
    }
```

Listing 23: Load input to register file

2. trigger_Keccak(): Starts the Keccak permutation by writing 1 in the start bit of Keccak's control register (listing 24).
   This operation is time-sensitive: memory barriers are used to constrain the compiler to execute instructions in the given order.

```
    void trigger_op(void)
    {
        uint32_t volatile * ctrl_reg = (uint32_t*)KECCAK_CTRL(0);
        asm volatile ("": : : "memory");
        *ctrl_reg = 1 << KECCAK_CTRL_START;
        asm volatile ("": : : "memory");
    }
```

Listing 24: Trigger of the Keccak permutation

3. pool_done(): The end of the permutation is checked via polling. The Keccak's status register is repeatedly sampled till the status bit is set to 1 which corresponds with the finish of the permutation (listing 25).

```
void poll_done(void){
    uint32_t volatile *status_reg = (uint32_t*)KECCAK_STATUS(0);
    uint32_t current_status;
    // Busy waiting till the Keccak is ready to start
    do {
       current_status = (*status_reg)&(1<<KECCAK_STATUS_STATUS);
    } while ( current_status == KECCAK_BUSY);
    printf("Keccak : ready\n");
}
```

Listing 25: Pooling the status bit

4. get_result(Dout): The result stored in Keccak's I/O registers is saved in the Dout vector (listing 26).

```
void get_result(uint32_t* Dout){
    uint32_t volatile *Dout_reg_start = (uint32_t*)KECCAK_DOUT_0(0);
    for (volatile int i = 0; i<50; i++){
        Dout[i] = Dout_reg_start[i];
    }
}
```

Listing 26: Store of Keccak result

The driver is then integrated into pulp-runtime by :

- Moving the header file (*Keccak_driver.h*) and the auto-generated header file (*Keccak_auto.h*) in "pulp-runtime/include "directory.
- Moving the driver implementation file (*Keccak_driver.c*) into "pulp-runtime/driver ".

The Makefile of PULPissimo "pulpissimo.mk "is modified too, by adding the following line.

```
PULP_SRCS += drivers/Keccak_driver.c
```

Listing 27: code: Keccak source files registered in pulpissimo.mk

Moreover, the header files of the driver are included in the file "pulp.h". The added lines of code are in listing 28. This is the main header file that includes all others and is the only one included in application programs.

```
#include <keccak_auto.h>
#include <keccak_driver.h>
```

Listing 28: code: added header file to pulp.h

The driver implementation is finished and the Keccak accelerator is ready to be tested.

5.7 Testing Keccak accelerator with PULPissimo

The Keccak accelerator can be tested in PULPissimo by running the developed application "*Keccak_test.c*".
This program loads in the input vector a known value and then proceeds to call the driver of the accelerator (*KeccakF1600_StatePermute* function). It busy waits and lastly prints the output vector in hexadecimal.
The input test value and the result expected from the permutation are reported in table 5.1. Since the Keccak core reads/writes data 64 bits at a time, in the table the values are presented as vectors of 25 elements where each element is 64 bits in width.

Index	**Input vector [HEX]**	**Expected result [HEX]**
0	EC4AFF517369C667	E1ADB0E2E7CB8356
1	00000010ABBACD29	BB3F5FB8573A5BD7
2	0000000000000000	F7CA02A1E9784CC5
3	0000000000000000	6E54F25660A4C685
4	0000000000000000	77051F83243FCBAA
5	0000000000000000	6459DB0B4C063DD5
6	0000000000000000	E046DE71CB4B81C6
7	0000000000000000	94051793DB31F24C
8	0000000000000000	A13FC86CF16E32DD
9	0000000000000000	B962FC91B7737708
10	0000000000000000	D3CA2E7AFA27C801
11	0000000000000000	53C85108F72A3CCA
12	0000000000000000	73E732CDADF0E783
13	0000000000000000	8470BD54C4BDD1BF
14	0000000000000000	D10B916F7C8C1F77
15	8000000000000000	51129474440A2670
16	0000000000000000	3D77CB49E9960C44
17	0000000000000000	EC5001EBE4251E39
18	0000000000000000	77A0EEC5EA4FD653
19	0000000000000000	EBC86BD47B6773E7
20	0000000000000000	E77DF6B0128FDC4B
21	0000000000000000	0DB0D48A02F1B12E
22	0000000000000000	241B344D0DC38AE5
22	0000000000000000	C3EE4E27532483D8
24	0000000000000000	0271BFE284B1B424

Table 5.1: Test application input and expected result

The C code of *Keccak_test* is reported in listing 29.

```
1  #include <stdio.h>
2  #include <inttypes.h>
3  #include "pulp.h"
4
5  int Keccak_test()
6  {
7      uint32_t Din[50];
```

```
8       uint32_t Dout[50];
9       // set vectors inital value to 0
10      memset(Din, 0, sizeof(Din));
11      memset(Dout, 0, sizeof(Dout));
12      // Store in Din the test value
13      Din[0] = 0x7369C667;
14      Din[1] = 0xEC4AFF51;
15      Din[2] = 0xABBACD29;
16      Din[3] = 0x00000010;
17      Din[31] = 0x80000000;
18      // Called Keccak function
19      KeccakF1600_StatePermute(Din, Dout);
20      printf("lenght of Dout %d\n", sizeof(Dout)/sizeof(Dout[0]));
21      // Busy waiting
22      for (int i=0; i<50; i++){}
23      for (int i = 0; i<50; i+=2){
24      printf("iteration %d\n", i/2);
25      printf("%" PRIx32 "-", Dout[i+1]);
26      printf("%" PRIx32 "\n", Dout[i]);
27     }
28     return 0;
29  }
```

Listing 29: Keccak test

The application starts filling the vectors with 0s. This turns out to be a simple but effective way to discriminate problems during debugging.
It then proceeds to load the input vector with the test value, then calls the device driver, and busy-waits for a very short time so that the hardware accelerator has time to complete the permutation and write the output result.
Generally, if hardware timers are present in the SoC is preferred to use these instead of busy-waiting for performance reasons. In this case, is not done since is wanted to just verify the accelerator behavior.
As explained in section section 5.6, the only file included in application programs is “pulp.h ”since this one has inside the headers of all the runtime functions.
The program is then compiled and run by invoking the following commands in the program directory.

```
# clean previous build data
$ make clean
# compile the program
$ make all
# run the rtl simulation without gui
$ make run
```

As can be observed in figure Figure 5.11 the output of the program checks the expected one.
To get a greater understanding of the driver behavior in the application, can be run the RTL simulation in GUI mode.

```
> make run GUI=1 # run the rtl simulation with gui
```

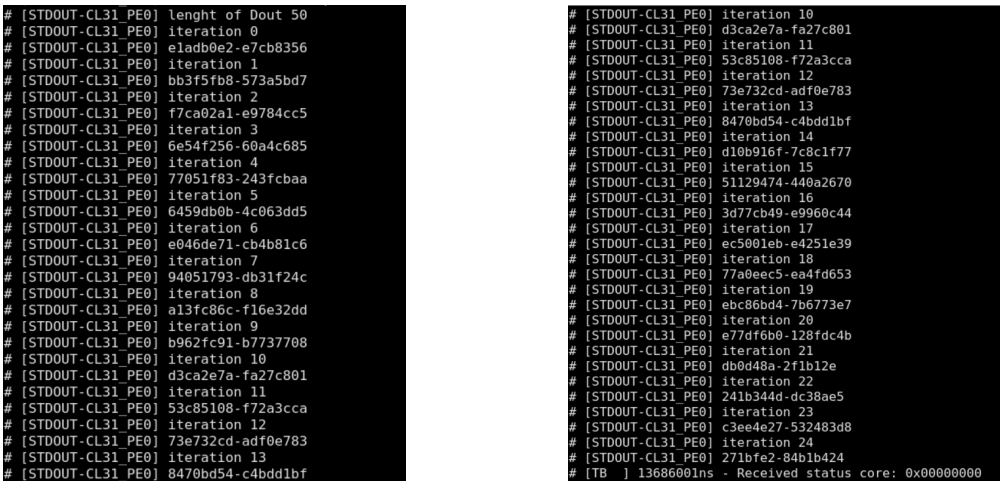


Figure 5.11: Output of Keccak test

The graphical simulation is set to group signals of interest, as can be noticed in Figure 5.12. In the *wave window* is shown the CPU workflow (instructions disassembly, PC) and the signals of the Keccak module.
Figure Figure 5.12a shows how the input data is loaded in the keccak register file while figure Figure 5.12b shows how the CPU checks the status bit via polling and how it starts to read the output result.

The PULP team provides useful wave files to test and debug an application, which are inside the *pulpissimo* directory in folder "*sim/waves* ". Among the ones provided is noteworthy the file "*software.tcl*"that groups the instruction disassemble, the program counter and the content of the processor register file.
To test the optimized keccak accelerator has been created a wave file named "*keccak_optimized.tcl*". The graphical simulation shown in Figure 5.12 can therefore be replicated by running in VSIM shell the following commands Listing A.2.

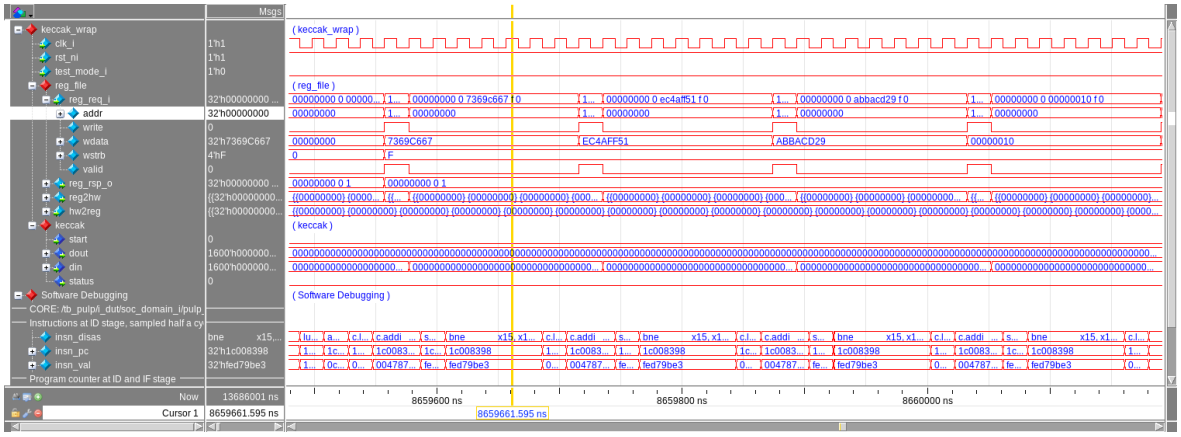
```
do waves/software.tcl
do waves/keccak_optimized.tcl
```

Once the application has been built and run, a new directory called *build* is created inside the application directory. The build directory contains all the application object files. Also, it contains the trace log of the application (named *trace_core_000003e0.log*). The trace file contains all the instructions executed by the application.
In Table 5.2 is an example of how an instruction executed is represented in the trace file.

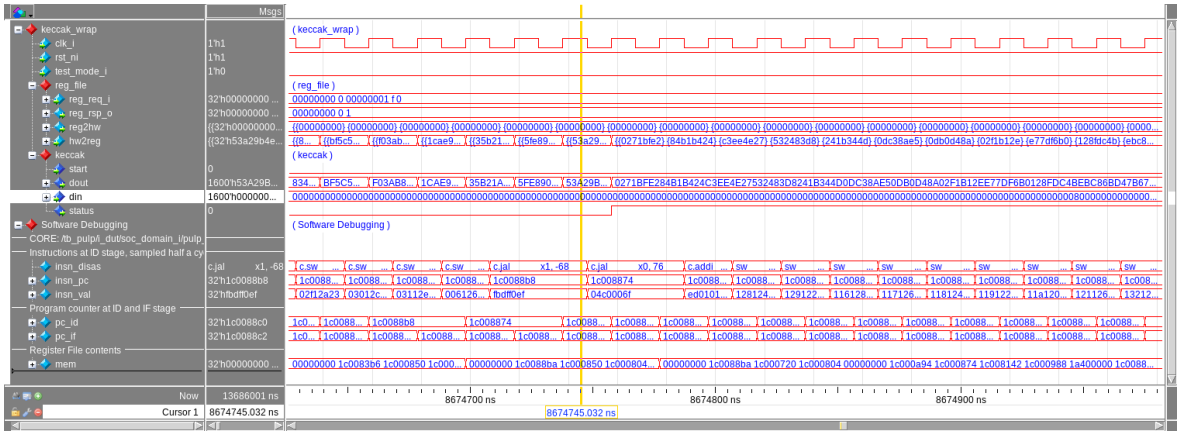
Time [ns]	**Cycle count**	**PC**	**Instruction code**	**Operands regs**	**Result**	**Operands Value**
19856274	340552	1c00835e	ff010113	c.addi x2, x2, -16	x2=1c003f90	x2:1c003fa0

Table 5.2: Format used to represent an instruction in the Trace File

The application benchmark can be derived by analyzing this file. However, this format represen-



(a) Load of input data in Keccak register file



(b) Write of status bit from Keccak

Figure 5.12: Waveforms of Keccak simulation

tation is not easily readable.
PULP group provides a script tool named "*pulptrace*"that generates a more human-readable format of the trace file. The script is in "pulp-runtime/scripts".
The processed trace file can be obtained by running in the build directory of the application the following command.

```
$ export SCRIPT_DIR=PULP_RUNTIME_DIR/scripts
$ ./$SCRIPT_DIR/pulptrace trace_core_000003e0.log keccak_test/keccak_test
  > processed_trace.log
```

This tool takes as inputs the trace file of the application and its binary file (in this case *keccak_test*). In the generated file the executed instructions are represented in the Table 5.3.

PC	**Function**	**Instruction mnemonic**	**Operands regs**	**Result**
1c00835e	$KeccakF1600\_StatePermute$ +0x4	addi sp,sp,-16	sp=1c003f90	sp:1c003fa0

Table 5.3: Processed trace file representation

The duration of Keccak's function can be retrieved as follow.

- The start and end points of the function and the relative program counter values are found in the processed trace file.
- The program counter values are used in the trace file to find the start and end cycle of the function. The difference is the execution time.

Another comparison between the two implementations is the number of machine instructions that make up the application. This can be calculated from the disassembly of the application.

```
$ make dis > keccak_dis.s
```

In the reference Table 5.4 is reported a summary of the performance improvements obtained with the accelerator.

	SW implementation	**HW accelerator**	Improve factor
Keccak duration [cycles]	26529	1348	19.68x
Machine instructions	1348	100	13.48x

Table 5.4: Keccak permutation benchmark

The results obtained show that the accelerator is almost **twenty times faster** than the software implementation. In addition, the number of machine instructions that make up the algorithm is far less, a great improvement that can be attributed to the solution chosen which requires a simpler and hence smaller driver.
This test can be reproduced by running the bash script "*run_keccak_test.sh*". This file is in the public Github directory [4].

5.8 Keccak optimization

A critical part of this thesis is the optimization of the Keccak core within PULPissimo.
To integrate the Keccak core as an external accelerator into PULPissimo, hardware additions to the accelerator were made.
As explained section 5.4, to make the accelerator suitable to be driven by the RISC-V core it needs to be wrapped by a register file. The choice adopted has been to use a complex register file to mitigate performance degradation and reduces the CPU workload. This added register file though adds a substantial area overhead.
This solution has been preferred not only for its performance benefits but also because with the added register file the architecture of the Keccak core can be greatly optimized.
In the end, redesigning the Keccak architecture the area overhead has been totally canceled and Keccak's performances have been increased too.
Overall, the decision to use a more complex register file in the memory mapping solution is a strategic choice that aimed to have the best performance and area overhead.
Analyzing the architecture of the Keccak accelerator, in Figure 5.13, can be noticed that it has an internal 1600-bit buffer width. This buffer is used to store I/O data and the intermediate results of the permutation.

[4] https://github.com/aledolme/pqc_riscv

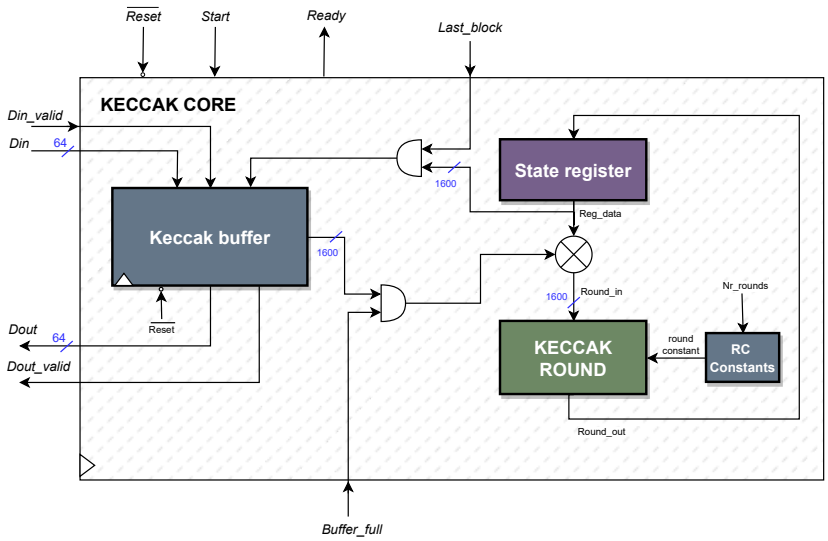


Figure 5.13: Keccak accelerator RTL

The register file added to the Keccak wrapper has 50 registers of 32-bit width (so a total of 1600-bit) that it uses for I/O data exchange.
Having two buffers of the same size that perform similar tasks is inefficient and needlessly increases the area of the architecture. Instead, we can optimize the architecture by eliminating the internal buffer of Keccak and utilizing the register file to replace its function.
This is feasible because the time in which the register file and Keccak's buffer operate do not overlap. When Keccak is computing the permutation, the register file is idle. It can be used to store the permutation intermediate results making the buffer unnecessary.
This solution has multiple benefits :

- Eliminates the main contribution to the area overhead added by the register file.
- Reduce Keccak's latency and improve power performance.
- Requires a simpler control unit to manage the accelerator.

The block diagram of the resultant architecture can be appreciated in the Figure 5.14.
As can be observed the optimized Keccak core is a combination of a controller, modeled as a Finite State Machine (FSM) and a Datapath.
This hardware design, known as Finite State Machine with Datapath (FSMD), is a common technique for efficient custom hardware design.
The datapath is the Keccak accelerator removed of its internal buffer and modified so that it computes at each clock cycle one stage of the permutation on 1600 bit.
The datapath takes in input the following signals.

- *D_in*: input data 1600-bit width, on which computes the permutation on.
- *start_i*: start signal, works like a synchronous reset and so when it is pulled up then the datapath reset and starts computing a new permutation.

while the datapath output signals are the following.

- *D_out*: output data 1600-bit width, the result of the permutation.
- *Ready_o*: ready signal, when is pulled up notifies that the keccak has finished computing the permutation and so is ready to start a new one.

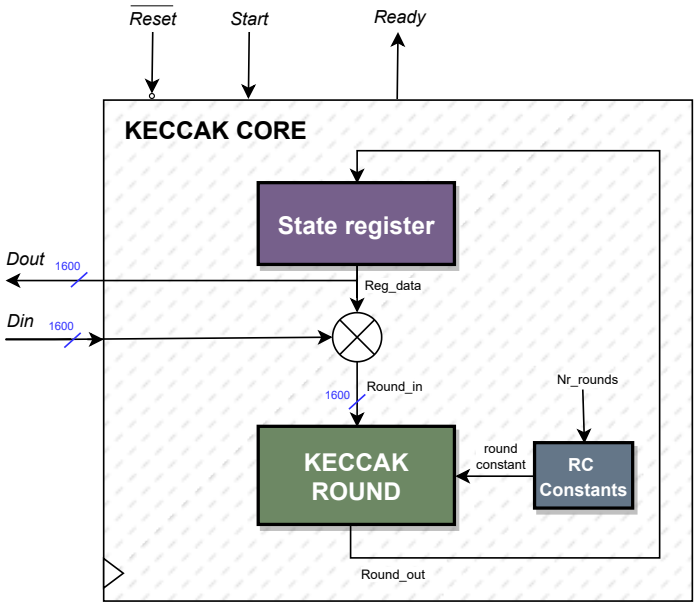


Figure 5.14: Optimized Keccak core architecture

The Keccak's datapath is defined in the file *keccak_datapath.sv* and in listing 30 is reported its top entity.

```
    entity keccak_dp is

    port (
        clk     : in  std_logic;
        rst_n   : in  std_logic;
        start_i : in std_logic;
        din     : in  std_logic_vector(1599 downto 0);
        ready_o : out std_logic;
        dout    : out std_logic_vector(1599 downto 0));

end keccak_dp;
```

Listing 30: Entity of the optimized Keccak accelerator

The control unit's job is to handle the datapath and ensure that the permutation is executed correctly. The control unit is designed as a Finite State Machine (FSM) and is composed of three states :

- **wait_start**: the CU waits for the start signal from the CPU, the start should be received after the input vector has been loaded in the register file.
  When the start is received, then the CU checks if the Keccak is ready to perform a new permutation (*ready_o* is equal to '0). If it is the case, then set the *start_keccak* signal high and go to the next state *do_permutation*. Otherwise, it stays in this state.
- **do_permutation**: it lasts 24 cycles, the time needed by Keccak to perform the permutation. In the end, the CU goes to the next state *permutation_finished*.

- **permutation_finished**: the CU signals the end of permutation by writing in the status bit '1 and then returns to *wait_start* state.

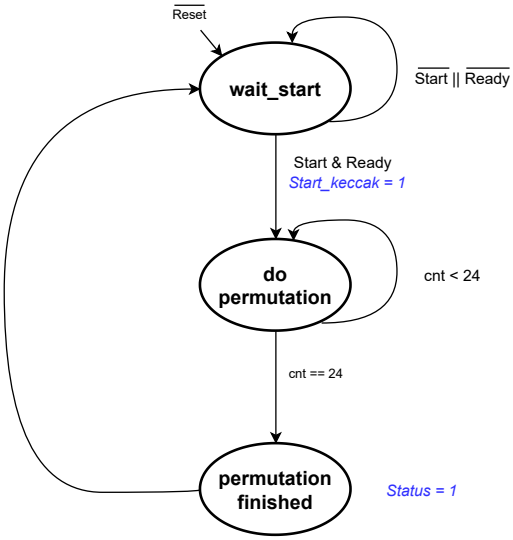


Figure 5.15: Control unit state diagram

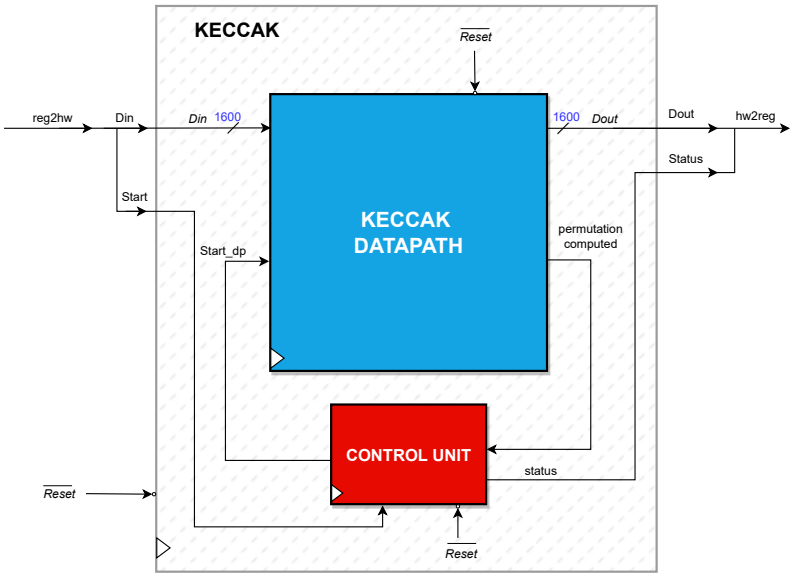


Figure 5.16: Connection between CU and Keccak accelerator

The Keccak accelerator is defined in the file *keccak.vhd*. Here datapath and control unit are instantiated and connected to each other. The resultant Keccak architecture has been verified in *QuestaSim*, and its test can be reproduced by running the script "*init_Questa.sh* "that can be found insubsection A.1.1.

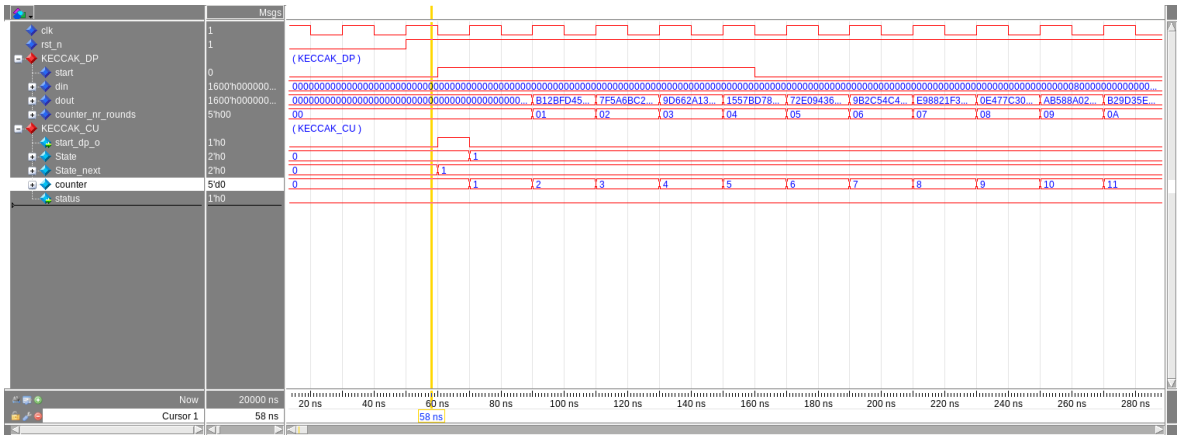


Figure 5.17: Start of the optimized Keccak accelerator

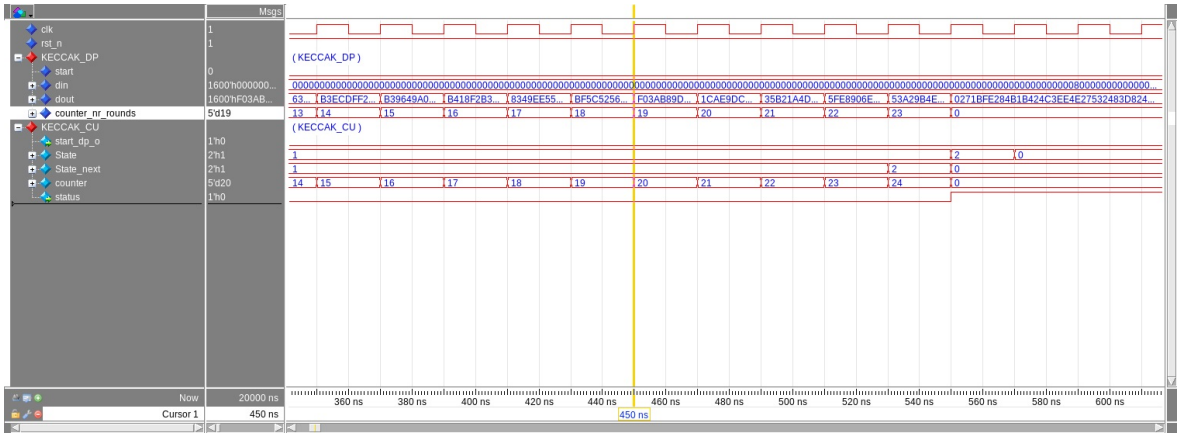


Figure 5.18: Permutation finish of the optimized Keccak accelerator

The Keccak permutation lasts 24 cycles since the designed accelerator operates directly on 1600-bit at the time.
The not optimized Keccak accelerator requires 76 cycles to perform the permutation, so the optimized architecture is **over 3 times faster**, this is due to the elimination of the register file latency.
Indeed, the input vector no longer needs to be saved in the internal buffer 64 bits at a time, and the result is no longer given in output by unpacking it 64 bits at a time. Since it takes 25 clock cycles to store a 1600 bit input 64 bits at a time in the buffer and 2 cycles to match the synchronization requirements of the register file, this explains the difference of 52 clock cycles.
The optimized Keccak is instantiated in the Keccak wrapper. The modifications done to *keccak_top.sv* are reported in listing 31.

```
    entity keccak is
    port (
        clk     : in  std_logic;
        rst_n   : in  std_logic;
        start   : in std_logic;
        din     : in  std_logic_vector(1599 downto 0);
        dout    : out std_logic_vector(1599 downto 0);
        status  : out std_logic);
end keccak;
```

Listing 31: code: top-entity of the optimized Keccak core

Lastly, the Bender manifest of *pulp_soc* is modified since the source tree of the Keccak has changed. The *keccak_buffer* and its packages are no longer used.
The modifications done to the Bender.yml of *pulp_soc* are reported in listing 32.

```
    sources:
  # keccak ip
  - files:
      - rtl/keccak_ip/gen_sv/keccak_reg_pkg.sv
      - rtl/keccak_ip/gen_sv/keccak_reg_top.sv
      - rtl/keccak_ip/keccak_globals.vhd
      - rtl/keccak_ip/Keccak_2to1mux.vhd
      - rtl/keccak_ip/Keccak_REG_rst_n.vhd
      - rtl/keccak_ip/reg_en_rst_n.vhd
```

```
10      - rtl/keccak_ip/keccak_round_constants_gen.vhd
11      - rtl/keccak_ip/keccak_round.vhd
12      - rtl/keccak_ip/keccak_cu.sv
13      - rtl/keccak_ip/keccak_datapath.vhd
14      - rtl/keccak_ip/keccak.vhd
15      - rtl/keccak_ip/keccak_top.sv
```

Listing 32: Modified manifest file of pulp_soc

The *keccak_top* portlist of the optimized architecture is no different from the un-optimized one. The driver's behavior is therefore the same.
The optimized architecture is completely integrated and no more modifications are needed.
The benchmark of the optimized architecture can be evaluated by simulating the Keccak algorithm. Table 5.5 summarizes the improvements obtained with the optimized Keccak architecture.
The algorithm execution time is retrieved by joining the information of the trace file and of the file generated by *pulp_trace* (as done in section 5.7).
Table 5.5 is presented a benchmark comparison of the two accelerator, the basic version and the optimized one.
The architectures have been synthesized with *Synopsys* using the "*Nangate 45nm Open Cell Library* ", which is a generic open-source library non-manufacturable.
The power estimation is evaluated with a drain voltage V_{dd} that is equal to 1.1 V and a frequency of 500 MHz (hence a clock period of 2 ns).
In the appendix can be found the scripts to replicate these syntheses (Listing A.3 for the Keccak accelerator, Listing A.4 for the optimized Keccak accelerator).

	Execution time (cycles)	Total area [µm^2]	Combinational area [µm^2]	Non-cominational area [µm^2]
Keccak	1348	40637	23512,8	17125
Optimized Keccak	1241	27506,5	8549,2	18957.3
Improvement	8.62%	47,2%	175%	-9.6%

Table 5.5: Optimized Keccak architecture benchmark

By removing the internal buffer, the optimized area architecture boasts a significantly smaller footprint, with a reduction of 47% compared to its predecessor. In addition, the elimination of the register file's latency has resulted in a modest 8% improvement in execution time.

5.9 FPGA implementation

In order to test and prototype the accelerator, the PULPissimo SoC with the developed accelerator has been synthesized with *Xilinx Vivado 2022.1*. The PULPissimo SoC is composed by:

- the CV32E40P core (formerly RI5CY) [26]
- L2 RAM of size 512kB RAM, composed of 4 banks each 128 kB
- private memory L1 of size 64 kB, composed of 2 banks each of 32 kB
- the Keccak accelerator

PULPissimo can be implemented for various Xilinx FPGA boards, the board selected in this thesis is the Xilinx ***Zedboard xc72020clg484-1***.
In PULPissimo Github[5] directory the PULP team provides precompiled bitstreams for the supported boards.
The PULPissimo bitstream for a supported target FPGA board can be generated by

1. In PULPissimo main folder the necessary synthesis *include scripts* are generated by calling the following command.

   ```
   $ make scripts
   ```

   This command parses the *Bender.yml*, using the PULP bender dependency management tool, to generate tcl scripts for all the IPs used in the project. These files are later on sourced by Vivado to generate the bitstream for PULPissimo.

2. In the FPGA subdirectory is started the appropriate make target to generate the bitstream with the following command.

   ```
   $ make <board_target>
   ```

If successful in FPGA directory are now contained two files.

"*pulpissimo_* < *board_target* > *.bit*" : the bitstream file for JTAG configuration of the FPGA.

"*pulpissimo_* < *board_target* > *.bin*" : the binary configuration file to flash to a non-volatile configuration memory.

Once the bitstream *pulpissimo_genesys2.bit* is generated in the FPGA folder, in Vivado the bitstream can be loaded into the FPGA or using the configuration File (*pulpissimo_genesys*2*.bin*) or by flashing it on-board.
In Vivado to Bitstream Flashing

```
Open Hardware Manager
Open Target
Program device
```

The FPGA is read to emulate PULPissimo.

[5] https://github.com/pulp-platform/pulpissimo

Table 5.6 shows the resource cost of the presented accelerator. It reports a factor of increase of 12% in LUTs and 8.5% in FFs occupation. The majority of the overhead is obtained because of the high dimension of the Keccak state.

	PULPissimo	**PULPssimo with acc.**	Increase factor
LUTs	49,945	57,072	+12%
FFs	40,494	44,276	+8.5%

Table 5.6: Resource occupation on Zedboard xc72020clg484-1

Table 5.7 reports the detailed area utilization of the main component of the Keccak architecture. Most of the resources, in particular about the 55%, are occupied by the core, and more than half of it is occupied by sequential components. For Vivado implementation, the memory blocks involved have

Name	Slice LUTs	Slice Registers	DPS	BRAM
Keccak IP	**6333**	**3615**	**0**	**0**
◇ Keccak core	3818	1629	0	0
◇ Register file	1874	1600	0	0
◇ AXI converter	640	386	0	0

Table 5.7: Resources utilization after implementation

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

CHAPTER 6

Results

The primary objective of this thesis is to accelerate in a deployment scenario the post-quantum cryptography algorithm CRYSTAL-Kyber, which is the winner of the NIST PQC standardization contest for general encryption[5]. Through the algorithm profiling done in section 2.3, it was determined that the Keccak sub-function would provide the greatest benefit in terms of computational cost, and as such, a dedicated hardware accelerator was designed for this sub-function.
The choice of application scenario has been influenced by the desire to contribute to the TRISTAN project, which is a European project that aims to mature and expand the RISC-V ecosystem.
To this end, the PULPissimo SoC[13] was chosen since it is an open-source microcontroller with a RISC-V core. The developed Keccak accelerator has then been integrated as an external hardware accelerator in PULPissimo and a dedicated driver has been created for it.
In this chapter, the benefits of the Keccak accelerator are summarized and conclusions are drawn based on the performance results obtained from running the CRYSTALS-Kyber algorithm with and without the accelerator. The comparison between these results highlights the significant improvements achieved with the accelerator, which are then contextualized by comparing them with state-of-the-art solutions.
To further enhance this thesis work, future improvements are proposed.

6.1 Simulating *CRYSTALS-Kyber* with Keccak accelerator

The impact of the accelerator has been tested for all the *CRYSTALS-Kyber* security levels.
The workflow followed for all versions is:

1. Run the algorithm on the default SoC (without the accelerator) and retrieve the algorithm execution time. This is the reference point to compare with.
2. Run the algorithm on the SoC with the accelerator and retrieve the algorithm execution time.
3. Compute the speed-up factor of the accelerator

In the "test"folder on Github[34], the three sub-folders present refer respectively to NIST-level of security I, III, and V.

- kyber512
- kyber768
- kyber1024

To perform benchmark testing of each algorithm, separate execution and profiling of the algorithm's three main components, namely key generation, encryption, and decryption, were conducted. Within the directory of each algorithm, three subfolders have been created, namely keygen, enc, and dec, which respectively contain the code implementation of each of these components.
The test of the algorithms can be reproduced by sourcing the shell scripts "run_kyber ", which can be found in appendix subsection A.1.5.
This script takes in input two parameters, the version of the algorithm and which part is wanted to run.
In the test folder, the simulation can be run with the following command.

```
$ source run_kyber $ALG $PART
```

ALG value can be : $ALG = 512, 768, 1024$

PART value can be : $PART = keygen, enc, dec$

CRYSTALS-Kyber algorithm changes to exploit the accelerator

The CRYSTALS-Kyber algorithm was benchmarked using the accelerator, and to enable this, modifications were made to the algorithm's C code. Specifically, the code was altered to ensure that when the Keccak function is called, the accelerator driver is utilized, rather than performing the Keccak permutation in software.
The Keccak function is called only in *fips202.c* file. The Keccak function is named $KeccakF1600\_StatePermute$ and is defined as follows.

```
1    void KeccakF1600_StatePermute (uint64_t State[25])
```

The accelerator is exploited by replacing the call to this function with a call to the device driver.
The *Keccak* driver to compute the permutation in HW, defined in section 5.6, is defined as follows.

```
1    void KeccakF1600_StatePermute (uint32_t Din[50], uint32_t Dout[50])
```

It differs from the software definition since it accepts two vector parameters, both of which are composed of 25 elements each element 32-bit width. One vector is the input value while the other is the output result.
The driver function replaces the call to the software function Keccak in "fips202.c".

$$KeccakF1600\_StatePermute(s) \xrightarrow{Replaced \quad with} KeccakF1600\_StatePermute(s, f)$$

After these modifications, the *CRYSTALS-Kyber* algorithms can be tested using the Keccak accelerator.
The cycle count results for all three levels of security of *CRYSTALS-Kyber* run on the PULPissimo RISC-V platform, with and without the accelerator, can be found in Table 6.1.

		Reference	Accelerated	Speed-up factor
Kyber512	KeyGen	1,101,598	395,495	2.79
	Encaps	1,435,915	552,827	2.60
	Decaps	1,432,310	726,049	1.97
Kyber768	KeyGen	1,772,967	663,059	2.67
	Encaps	2,280,600	856,258	2.66
	Decaps	2,258,939	1,083,818	2.08
Kyber1024	KeyGen	2,767,127	1,001,350	2.76
	Encaps	3,383,780	1,247,565	2.71
	Decaps	3,352,080	1,523,411	2.20

Table 6.1: Cycle counts for RISC-V PULPissimo platform [30]

As can be observed from the table, in all the security levels the Keccak accelerator improvement is impressive. **The algorithm execution time in all cases is more than halved**, up to a 2.79x speed-up in the Keygen algorithm with security level I.

6.2 Comparison with state-of-art solutions

There are not many prior works in the literature accelerating the CRYSTALS-Kyber algorithm with the standardized NIST parameter set. Therefore, it is hard to compare our design for each operation and parameter set with other works.
The following report works that better connect with this thesis and gives an overview of the SoA solutions.

In [35] is designed an optimized implementation of Kyber encryption schemes targeting the 64-bit ARM Cortex-A processors. The AES accelerator in the target board is used (Table 6.2).
Even though our designed accelerator is only for the permutation step and not the absorbing and squeezing phases of the hash algorithm, the results of our accelerator are better.

Noticeable is the work proposed in [36], the RISC-V architecture is enhanced by integrating a set of tightly coupled accelerators (application-specific functional unit) to speed up lattice-based PQC. The RISC-V platform used to test the accelerators is PULPino [29], which is the microcontroller version previous to PULPissimo. Compared to the pure software implementation on RISC-V, the accelerator implemented in this work shows a speedup factor of up to 9.6 for Kyber.

In [9] is proposed a dedicated *PostQuantum Arithmetic Logic Unit*, which is embedded directly in the pipeline of a RISC-V processor to execute the lattice-based algorithms CRYSTALS-Kyber and -Dilithium. In particular, this paper describes two different hardware accelerators that may be integrated together or singularly as two different FUs into the processor pipeline.
The first is a dedicated architecture to accelerate the polynomial multiplications which use the NTT and modular reductions, exploiting dedicated instructions for the fqmul and Barrett reduction in the case of Kyber, and the fqmul and reduce32 in the case of Dilithium.
The second is a dedicated architecture for the butterfly operations used in NTT and INTT computations for both algorithms.
Compared to the results obtained in this thesis work, these implementations achieve a lower speed-up of the Kyber algorithm. Though our architecture has a much higher resource consumption.

Another possible design approach is proposed in [11]. Here they designed a domain-specific vector co-processor, integrated with a RISC-V processor, focusing on the NTT transform for the Ring-LWE and Module LWE PQC algorithms.
To support the parallel computation of number theoretic transform (NTT) of different dimensions (from 64 to 2048), a vector NTT unit is implemented in VPQC. Besides, a vector sampler executing both uniform sampling and binomial sampling is also employed. Evaluated under TSMC 28nm technology, the vector coprocessor achieves almost an order of magnitude acceleration in computing KEM protocols, though this high performance is due to an area cost much larger of 942k gates.

There are also contributions targeting ASIC accelerators like the one presented in [37]. The ASIC-specific accelerator is developed over a 65nm standard cell library, this solution achieves high performance but at the price of more expensive hardware with less flexibility.

Comparing the solution proposed in this thesis with other works in the literature is a challenging task due to the scarcity of similar alternative solutions and the multitude of factors affecting the system's performance. The CPU and FPGA on which the systems were implemented are among the key factors influencing performance.
The aim of the results presented in Table 6.2 is not to provide a comparison metric but rather to review the various designs available and highlight their differences. Furthermore, it emphasizes that the solution proposed in this thesis is a valid option among the existing designs.

	Design methood	**Processor**	**Speed-up factor**			**FPGA**	**Complexity**			
			Kyber512	**Kyber758**	**Kyber1024**	**platform**	**LUT**	**FF**	**DSP**	**BRAM**
This work	HW loosely coupled	RISC-V CV32E40P	2.45	2.47	2.56	Xilinx Zedboard	6333	3615	0	0
[35]	HW/SW	ARM Cortex-A	1.96	1.89	1.82	NA	NA	NA	NA	NA
[9]	HW tightly coupled	RISC-V CVA6	1.83	1.84	1.83	Xilinx ZCU106	178	0	5	0.5
[36]	HW tightly coupled	RISC-V CV32E40P	7.68	7.77	9.63	Xilinx Zynq-7000	24.306	10.837	18	32
[11]	HW co-processor	ARM Cortex-M4	14.88	22.45	NA	NA	NA	NA	NA	NA

Table 6.2: SoA solutions review : results achieved with different design methodology

The report of [35] lacks of any synthesis benchmark. Instead, the vector co-processor of [11] has been implemented with Synopsys Design Compiler under TSMC 28nm HPC+ technology with a supply voltage of $0.9V$, it consumes 942k equivalent logic gates and 12KB memories.

In summary, the proposed solution in this thesis strikes a good balance between complexity and performance. Although it requires moderate resource consumption, this can be viewed as a reasonable trade-off for the impressive speed-up in performance. It is worth noting that the developed accelerator is system independent and highly versatile, thanks to its AXI interface, making it easy to attach to other microcontrollers.

6.3 Future improvements

The performance and robustness of the designed accelerator can be enhanced by various optimization techniques and design methodologies, possibilities that are left to be investigated as future work.
One potential optimization is writing the driver in assembly. Indeed, the compiled code is dependent on the compiler and may not be optimized for the specific processor and may not achieve the same level of performance as assembly code.
Another optimization involves using an interrupt after the accelerator completes the permutation. The driver developed in section 5.6 checks the status bit of the accelerator via the polling method, technique is discouraged since while the microcontroller continuously checks if the device has finished, it wastes cycles and power. Polling is only recommended for prototyping purposes.
Additionally, the accelerator's performance can be further improved by exploring techniques such as parallelization and pipelining.

Moreover, the standardization of CRYSTALS-Kyber makes it important to assess how well the accelerator withstands side-channel attacks. SCAs can retrieve information about a secret element of an algorithm by exploiting physical leakages eg. timing behavior, power consumption, or emanation in the electromagnetic field. In [12] is demonstrated a successful message (session key) recovery on a hardware implementation of CRYSTALS-Kyber, by deep learning-based power analysis. This result highlights the importance of the development of countermeasures against such attacks, as explained in [38] the masking (secret sharing method) may be the right way for protection against power analysis SCAs.

As for future work, the design presented in this thesis can be extended to perform all the SHA3 primitives. This can be done by modifying it to function as a reconfigurable SHA-3 hardware accelerator, which includes a Keccak accelerator with a reconfigurable rate and so able to support variable-length message digests. This modification provides a more adaptable and efficient solution for hash-based cryptography, as it can be adapted to different hash functions and input sizes depending on the requirements of the application, and may have applications in areas such as secure communication, digital signatures, and blockchain technology.

In this thesis work has been preferred to develop the accelerator "loosely coupled ", thus implementing it outside the processor cores. This solution makes the accelerator very versatile and, thanks to its AXI interface, easy to integrate into systems.
Inspired by the work of [36], the future accelerator can be tightly coupled and thus embedded within the RISC-V processor pipeline The tightly coupled accelerator is controlled by adding Instruction Set Architecture (ISA) extensions. It is expected that by using this design technique the communication and silicon area overhead will be reduced, though at the expense of less flexibility.

CHAPTER 7

Conclusion

The CRYSTALS-Kyber algorithm is a key encapsulation method (KEM) designed to withstand the assault of a future quantum computer, which could potentially crack the security used to protect privacy in the digital systems we rely on every day.
On 5 July 2022 NIST announced that the CRYSTALS-Kyber will be the new standard for post-quantum encryption, which is used when accessing secure websites.
This thesis proposes a hardware accelerator for the Kyber algorithm, which effectively demonstrates that hardware acceleration is an effective means to fulfill the high computation burden required by post-quantum cryptography algorithms. Indeed, the accelerator ensures high throughput and low latency.
After analyzing the Kyber algorithm, it was determined that the Keccak sub-function was the most beneficial part to accelerate. The proposed architecture, built upon Bertoni's design, has been integrated into the PULPissimo platform, which is a single-core RISC-V based microcontroller. The developed IP has been implemented as a loosely coupled accelerator, thus implemented outside the processor core, and is driven in a memory-map fashion style. It communicates with the SoC through an AXI interface, making it very versatile and easy to attach to other systems. The accelerator's area and performance have been optimized by removing unnecessary components such as Keccak's internal buffer.
After that, the SoC has been implemented on *Xilinx Zedboard xc7z020clg484-1* for gathering all the performance and area occupation results.
Testing the accelerator against software implementations, the results show that the accelerator is almost twenty times faster than the software at computing the Keccak permutation, with a factor of increase of 12% in LUTs and 8.5% in FFs occupation.
In Table 6.1, the speed-up achieved by the accelerator across all security levels of the Kyber algorithm is presented. On average, the accelerator achieves a 2.5 speed-up. In addition to the impressive speed gains, the algorithm's instruction count has also been reduced by 10%. This is a notable achievement as it implies that the solution chosen is more efficient and requires fewer instructions to execute the same task. The key to this improvement lies in the adoption of a simpler and hence smaller driver.
The experiment's results clearly demonstrate that the accelerator outperforms the software implementation, underscoring the effectiveness of hardware acceleration. The speed gains achieved by the accelerator, along with the reduction in instruction count, highlight hardware acceleration's potential to enable faster and more efficient algorithms.
It's important to note that the deployment, integration, and migration to quantum-safe security systems require time. The building of large quantum computers is no more just a concept but has become a mere engineering challenge. Although such technology is far from being commercialized, it's essential to act now to address the real threat posed by its inevitable existence in the future.

APPENDIX A

Appendix

A.1 Scripts

A.1.1 Init modelsim simulation

Listing A.1: Script to init a simulation

```
#!/bin/bash

echo "optional parameter. (1) vsim gui, (0) vsim without gui"

run=$1

source /eda/scripts/init_questa
#source /software/europractice-release-2019/scripts/init_questa10.7c

echo "Questasim has been setup"
echo "Working directort $PWD"
vlib ./work

target="keccak.tcl"

if [ "$run" == '1' ]
then
        vsim -do ./$target
elif [ "$run" == '0' ]
then
        vsim -c -do ./$target
else
        vsim -c
fi
```

A.1.2 PULPissimo wave file for Keccak accelerator

Listing A.2: Keccak waveform simulation in PULPissimo

```
onerror {resume}
quietly WaveActivateNextPane {} 0
add wave -noupdate -group keccak_wrap /tb_pulp/i_dut/soc_domain_i/pulp_soc_i/
    i_keccak_top/clk_i
add wave -noupdate -group keccak_wrap /tb_pulp/i_dut/soc_domain_i/pulp_soc_i/
    i_keccak_top/rst_ni
add wave -noupdate -group keccak_wrap /tb_pulp/i_dut/soc_domain_i/pulp_soc_i/
    i_keccak_top/test_mode_i

add wave -noupdate -group keccak_wrap -group reg_file /tb_pulp/i_dut/soc_domain_i/
    pulp_soc_i/i_keccak_top/i_regfile/reg_req_i
add wave -noupdate -group keccak_wrap -group reg_file /tb_pulp/i_dut/soc_domain_i/
    pulp_soc_i/i_keccak_top/i_regfile/reg_rsp_o
add wave -noupdate -group keccak_wrap -group reg_file /tb_pulp/i_dut/soc_domain_i/
    pulp_soc_i/i_keccak_top/i_regfile/reg2hw
add wave -noupdate -group keccak_wrap -group reg_file /tb_pulp/i_dut/soc_domain_i/
    pulp_soc_i/i_keccak_top/i_regfile/hw2reg

add wave -noupdate -group keccak_wrap -group keccak /tb_pulp/i_dut/soc_domain_i/
    pulp_soc_i/i_keccak_top/i_keccak/start
add wave -noupdate -group keccak_wrap -group keccak /tb_pulp/i_dut/soc_domain_i/
    pulp_soc_i/i_keccak_top/i_keccak/dout
add wave -noupdate -group keccak_wrap -group keccak /tb_pulp/i_dut/soc_domain_i/
    pulp_soc_i/i_keccak_top/i_keccak/din
add wave -noupdate -group keccak_wrap -group keccak /tb_pulp/i_dut/soc_domain_i/
    pulp_soc_i/i_keccak_top/i_keccak/status

TreeUpdate [SetDefaultTree]
WaveRestoreCursors {{Cursor 1} {0 ps} 0}
quietly wave cursor active 0
configure wave -namecolwidth 250
configure wave -valuecolwidth 100
configure wave -justifyvalue left
configure wave -signalnamewidth 1
configure wave -snapdistance 10
configure wave -datasetprefix 0
configure wave -rowmargin 4
configure wave -childrowmargin 2
configure wave -gridoffset 0
configure wave -gridperiod 1
configure wave -griddelta 40
configure wave -timeline 0
configure wave -timelineunits ns
update
run 10 ms
WaveRestoreZoom {8327340 ns} {8327440 ns}
```

A.1.3 Keccak accelerator synthesis script

Listing A.3: Synopsys synthesis script of the Keccak accelerator

```
define_design_lib work -path ./work

set search_path [list . /software/synopsys/syn_current_64.18/libraries/syn /software/
    dk/nangate45/synopsys ]
set link_library [list "*" "NangateOpenCellLibrary_typical_ecsm_nowlm.db" "
    dw_foundation.sldb" ]
set target_library [list "NangateOpenCellLibrary_typical_ecsm_nowlm.db" ]
```

```
set synthetic_library [list "dw_foundation.sldb" ]

analyze -library work -format vhdl {../rtl/src_64bit/with_FSM/keccak_globals.vhd}
analyze -library work -format vhdl {../rtl/src_64bit/with_FSM/Keccak_2to1mux.vhd}
analyze -library work -format vhdl {../rtl/src_64bit/with_FSM/Keccak_REG_rst_n.vhd}
analyze -library work -format vhdl {../rtl/src_64bit/with_FSM/reg_en_rst_n.vhd}
analyze -library work -format vhdl {../rtl/src_64bit/with_FSM/
    keccak_round_constants_gen.vhd}
analyze -library work -format vhdl {../rtl/src_64bit/with_FSM/keccak_round.vhd}
analyze -library work -format vhdl {../rtl/src_64bit/with_FSM/keccak_buffer.vhd}
analyze -library work -format vhdl {../rtl/src_64bit/with_FSM/keccak.vhd}

set power_preserve_rtl_hier_names true

elaborate keccak -architecture RTL -library WORK > elaborate.txt

uniquify
link

create_clock -name my_clk -period 1.5 {clk}
set_dont_touch_network my_clk
set_clock_uncertainty 0.07 [get_clock my_clk]
set_input_delay 0.5 -max -clock my_clk [remove_from_collection [all_inputs] clk]
set_output_delay 0.5 -max -clock my_clk [all_outputs]
compile

report_timing > results/notOpt/report_timing.txt
report_design > results/notOpt/report_design.txt
report_resources > results/notOpt/report_resources.txt
report_cell > results/notOpt/report_cell.txt
report_bus > results/notOpt/report_bus.txt
report_clock > results/notOpt/report_clock.txt
report_constraint > results/notOpt/report_constraint.txt
report_area -hierarchy > results/notOpt/report_area.txt
report_hierarchy > results/notOpt/report_hierarchy.txt
report_power > results/notOpt/report_power.txt
```

A.1.4 Optimized Keccak accelerator synthesis script

Listing A.4: Synopsys synthesis script of the optimized Keccak accelerator

```
define_design_lib work -path ./work

set search_path [list . /software/synopsys/syn_current_64.18/libraries/syn /software/
    dk/nangate45/synopsys ]
set link_library [list "*" "NangateOpenCellLibrary_typical_ecsm_nowlm.db" "
    dw_foundation.sldb" ]
set target_library [list "NangateOpenCellLibrary_typical_ecsm_nowlm.db" ]
set synthetic_library [list "dw_foundation.sldb" ]

analyze -library work -format vhdl {../rtl/src_64bit/with_FSM_noBuffer/
    keccak_globals.vhd}
analyze -library work -format vhdl {../rtl/src_64bit/with_FSM_noBuffer/
    Keccak_2to1mux.vhd}
analyze -library work -format vhdl {../rtl/src_64bit/with_FSM_noBuffer/
    Keccak_REG_rst_n.vhd}
analyze -library work -format vhdl {../rtl/src_64bit/with_FSM_noBuffer/
    reg_en_rst_n.vhd}
analyze -library work -format vhdl {../rtl/src_64bit/with_FSM_noBuffer/
    keccak_round_constants_gen.vhd}
analyze -library work -format vhdl {../rtl/src_64bit/with_FSM_noBuffer/
    keccak_round.vhd}
```

```
analyze -library work -format vhdl {../rtl/src_64bit/with_FSM_noBuffer/
    keccak_datapath.vhd}

set power_preserve_rtl_hier_names true

elaborate keccak_dp -architecture RTL -library WORK > elaborate.txt

uniquify
link

create_clock -name my_clk -period 1.5 {clk}
set_dont_touch_network my_clk
set_clock_uncertainty 0.07 [get_clock my_clk]
set_input_delay 0.5 -max -clock my_clk [remove_from_collection [all_inputs] clk]
set_output_delay 0.5 -max -clock my_clk [all_outputs]
compile

report_timing > results/Opt/report_timing.txt
report_design > results/Opt/report_design.txt
report_resources > results/Opt/report_resources.txt
report_cell > results/Opt/report_cell.txt
report_bus > results/Opt/report_bus.txt
report_clock > results/Opt/report_clock.txt
report_constraint > results/Opt/report_constraint.txt
report_area -hierarchy > results/Opt/report_area.txt
report_hierarchy > results/Opt/report_hierarchy.txt
report_power > results/Opt/report_power.txt
```

A.1.5 Shell file to simulate Kyber

In the pqc_riscv directory on GitHub[1].

Listing A.5: Shell script to run Kyber

```
#!/bin/bash

echo "Insert the version to run. (0) Kyber-512, (1) Kyber-768, (22) Kyber-1024"
read version_to_run
if [ "$version_to_run" -eq '0' ]
then
version_to_run="kyber512"
elif [ "$version_to_run" -eq '1' ]
version_to_run="kyber768"
else
version_to_run="kyber1024"
fi


echo "Insert the version to run. (0) KeyGen, (1) Encaps, (22) Decaps"
read part
if [ "$part" -eq '0' ]
then
part="keygen"
elif [ "$part" -eq '1' ]
part="enc"
else
```

[1] https://github.com/aledolme/pqc_riscv/tree/main/test

```
part=”dec”
fi

source /eda/scripts/init_questa
export PULP_RISCV_GCC_TOOLCHAIN=$HOME/riscv-pulp
export PATH=$PULP_RISCV_GCC_TOOLCHAIN/bin:$PATH
export VSIM_PATH=$pwd/pulpissimo
export RT_DIR=$pwd/pulp-runtime

source pulp-runtime/configs/pulpissimo.sh

cd pulpissimo
./bender update
make checkout
source setup/vsim.sh
env | grep VSIM
make clean
make script
make build
cd ../test/$version_to_run/$part
echo ”RUN simulation : (0) vsim withouth gui. (1) vsim with gui”
read gui
make clean all
make dis > keccak.s
if [ ”$gui” -eq '0' ]
then
make run
cd build
.$RT_DIR/scripts/pulptrace trace_core_000003e0.log main/main > processed_trace.log
else
make run gui=1
fi
```

Bibliography

[1] Arjen K Lenstra. Integer factoring. *Towards a Quarter-Century of Public Key Cryptography: A Special Issue of DESIGNS, CODES AND CRYPTOGRAPHY An International Journal. Volume 19, No. 2/3 (2000)*, pages 31–58, 2000.

[2] Neal Koblitz. Elliptic curve cryptosystems. *Mathematics of computation*, 48(177):203–209, 1987.

[3] Peter W Shor. Algorithms for quantum computation: discrete logarithms and factoring. In *Proceedings 35th annual symposium on foundations of computer science*, pages 124–134. Ieee, 1994.

[4] Daniel J Bernstein and Tanja Lange. Post-quantum cryptography. *Nature*, 549(7671):188–194, 2017.

[5] NIST CSRC. Post-quantum cryptography standardization–post-quantum cryptography, 2017.

[6] David Joseph, Rafael Misoczki, Marc Manzano, Joe Tricot, Fernando Dominguez Pinuaga, Olivier Lacombe, Stefan Leichenauer, Jack Hidary, Phil Venables, and Royal Hansen. Transitioning organizations to post-quantum cryptography. *Nature*, 605(7909):237–243, 2022.

[7] Guido Bertoni, Joan Daemen, Michaël Peeters, and Gilles Van Assche. Keccak. In *Advances in Cryptology–EUROCRYPT 2013: 32nd Annual International Conference on the Theory and Applications of Cryptographic Techniques, Athens, Greece, May 26-30, 2013. Proceedings 32*, pages 313–314. Springer, 2013.

[8] Prashanth Mohan, Wen Wang, Bernhard Jungk, Ruben Niederhagen, Jakub Szefer, and Ken Mai. Asic accelerator in 28 nm for the post-quantum digital signature scheme xmss. In *2020 IEEE 38th International Conference on Computer Design (ICCD)*, pages 656–662. IEEE, 2020.

[9] Pietro Nannipieri, Stefano Di Matteo, Luca Zulberti, Francesco Albicocchi, Sergio Saponara, and Luca Fanucci. A risc-v post quantum cryptography instruction set extension for number theoretic transform to speed-up crystals algorithms. *IEEE Access*, 9:150798–150808, 2021.

[10] Mojtaba Bisheh-Niasar, Reza Azarderakhsh, and Mehran Mozaffari-Kermani. Instruction-set accelerated implementation of crystals-kyber. *IEEE Transactions on Circuits and Systems I: Regular Papers*, 68(11):4648–4659, 2021.

[11] Guozhu Xin, Jun Han, Tianyu Yin, Yuchao Zhou, Jianwei Yang, Xu Cheng, and Xiaoyang Zeng. Vpqc: A domain-specific vector processor for post-quantum cryptography based on risc-v architecture. *IEEE transactions on circuits and systems I: regular papers*, 67(8):2672–2684, 2020.

[12] Yanning Ji, Ruize Wang, Kalle Ngo, and Elena Dubrova. A side-channel attack on a hardware implementation of crystals-kyber. In *2023 IEEE European Test Symposium (ETS'23)*, 2023.

[13] Pasquale Davide Schiavone, Davide Rossi, Antonio Pullini, Alfio Di Mauro, Francesco Conti, and Luca Benini. Quentin: an ultra-low-power pulpissimo soc in 22nm fdx. In *2018 IEEE SOI-3D-Subthreshold Microelectronics Technology Unified Conference (S3S)*, pages 1–3, 2018.

[14] Ronald L Rivest. Cryptography. In *Algorithms and complexity*, pages 717–755. Elsevier, 1990.

[15] Aditya Anand. Breaking down : Sha-1 algorithm. `https://infosecwriteups.com/breaking-down-sha-1-algorithm-c152ed353de2`.

[16] Cheap SSL Security. What is public key and private key cryptography, and how does it work? `https://cheapsslsecurity.com/p/what-is-public-key-and-private-key-cryptography-and-how-does-it-work/`.

[17] Oded Regev. The learning with errors problem (invited survey). In *2010 IEEE 25th Annual Conference on Computational Complexity*, pages 191–204. IEEE, 2010.

[18] Roberto Avanzi, Joppe Bos, Léo Ducas, Eike Kiltz, Tancrède Lepoint, Vadim Lyubashevsky, John M Schanck, Peter Schwabe, Gregor Seiler, and Damien Stehlé. Crystals-kyber algorithm specifications and supporting documentation. *NIST PQC Round*, 2(4):1–43, 2019.

[19] Alessandra Dolmeta. *Hardware architecture for CRYSTALS-Kyber cryptographic primitives*. PhD thesis, Politecnico di Torino, 2022.

[20] Chad Boutin. Nist selects winner of secure hash algorithm (sha-3) competition. *Press release., October*, 2, 2012.

[21] Guido Bertoni, Joan Daemen, Michaël Peeters, and Gilles Van Assche. Sponge functions. In *ECRYPT hash workshop*, volume 2007, 2007.

[22] 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 Oliver C. S. Choy, Ray C. C. Cheung, Peter Athanas, and Kentaro Sano, editors, *Reconfigurable Computing: Architectures, Tools and Applications*, pages 372–378, Berlin, Heidelberg, 2012. Springer Berlin Heidelberg.

[23] Guido Bertoni, Joan Daemen, Michaël Peeters, GV Assche, and RV Keer. 1001 ways to implement keccak. In *Third SHA-3 candidate conference*, 2012.

[24] Andrew Waterman, Yunsup Lee, David Patterson, Krste Asanovic, Volume I User level Isa, Andrew Waterman, Yunsup Lee, and David Patterson. The risc-v instruction set manual. *Volume I: User-Level ISA', version*, 2, 2014.

[25] Florian Zaruba, Fabian Schuiki, Torsten Hoefler, and Luca Benini. Snitch: A tiny pseudo dual-issue processor for area and energy efficient execution of floating-point intensive workloads. *IEEE Transactions on Computers*, 70(11):1845–1860, 2020.

[26] OpenHW Group. Cv32e40p. `https://github.com/openhwgroup/cv32e40p`.

[27] ETH Zurich. Cva6 core. `https://github.com/openhwgroup/cva6`.

[28] lowRISC. Ibex. `https://github.com/lowRISC/ibex`.

[29] Andreas Traber, Florian Zaruba, Sven Stucki, Antonio Pullini, Germain Haugou, Eric Flamand, Frank K Gurkaynak, and Luca Benini. Pulpino: A small single-core risc-v soc. In *3rd RISCV Workshop*, 2016.

[30] Alessandra Dolmeta, Mattia Mirigaldi, Maurizio Martina, and Guido Masera. Implementation and integration of keccak accelerator on risc-v for crystals-kyber. pages 1–2, 2023.

[31] PULP group. Generic_register_interface. `https://github.com/pulp-platform/register_interface`.

[32] Bender. Bender. `https://github.com/pulp-platform/bender`.

[33] The PULP team. *PULPissimo datasheet*, 3 2021.

[34] Mattia Mirigaldi Alessandra Dolmeta. Pqc_riscv. `https://github.com/aledolme/pqc_riscv`.

[35] 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. Cryptology ePrint Archive, Paper 2021/561, 2021. `https://eprint.iacr.org/2021/561`.

[36] Tim Fritzmann, Georg Sigl, and Johanna Sepúlveda. Risq-v: Tightly coupled risc-v accelerators for post-quantum cryptography. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, pages 239–280, 2020.

[37] Malik Imran, Zain Ul Abideen, and Samuel Pagliarini. A systematic study of lattice-based nist pqc algorithms: From reference implementations to hardware accelerators. *arXiv preprint arXiv:2009.07091*, 2020.

[38] Guido Bertoni, Joan Daemen, Michaël Peeters, and Gilles Van Assche. Building power analysis resistant implementations of keccak. In *Second SHA-3 candidate conference*, volume 142. Citeseer, 2010.