ISTANBUL TECHNICAL UNIVERSITY
ELECTRICAL-ELECTRONICS FACULTY

SYSTEM ON CHIP DESIGN FOR POST-QUANTUM CRYPTOGRAPHY USING RISC-V PROCESSOR

SENIOR DESIGN PROJECT

Ekin Türkü ERDOĞAN
Telat IŞIK

ELECTRONICS AND COMMUNICATION ENGINEERING DEPARTMENT

Berna Örs Yalçın
Uygundur
09.01.2024

JANUARY 2024

ISTANBUL TECHNICAL UNIVERSITY
ELECTRICAL-ELECTRONICS FACULTY

SYSTEM ON CHIP DESIGN FOR POST-QUANTUM CRYPTOGRAPHY USING RISC-V PROCESSOR

SENIOR DESIGN PROJECT

Ekin Türkü ERDOĞAN
040180624

Telat IŞIK
040180082

ELECTRONICS AND COMMUNICATION ENGINEERING DEPARTMENT

Project Advisor: Prof. Dr. Sıddıka Berna Örs YALÇIN

JANUARY 2024

İSTANBUL TEKNİK ÜNİVERSİTESİ
ELEKTRİK-ELEKTRONİK FAKÜLTESİ

RISC-V İŞLEMCİ KULLANARAK KUANTUM SONRASI KRİPTOGRAFİ İÇİN ÇİP ÜZERİNDE SİSTEM TASARIMI

LİSANS BİTİRME TASARIM PROJESİ

Ekin Türkü ERDOĞAN
040180264

Telat IŞIK
040180082

Proje Danışmanı: Prof. Dr. Sıddıka Berna ÖRS YALÇIN

ELEKTRONİK VE HABERLEŞME MÜHENDİSLİĞİ BÖLÜMÜ

OCAK, 2023

We are submitting the Senior Design Project Report entitled as “SYSTEM ON CHIP DESIGN FOR POST-QUANTUM CRYPTOGRAPHY USING RISC-V PROCESSOR”. The Senior Design Project Report has been prepared as to fulfill the relevant regulations of the Electronics and Communication Engineering Department of Istanbul Technical University. We hereby confirm that we have realized all stages of the Senior Design Project work by ourselves and we have abided by the ethical rules with respect to academic and professional integrity.

Ekin Türkü ERDOĞAN .....
040180264

Telat IŞIK .....
040180082

FOREWORD

First of all, we would like to express our sincerest gratitude to our advisor, Professor Dr. Berna Örs Yalçın, for her invaluable guidance, unwavering support, and expert mentorship throughout the completion of this project. Her dedication to excellence, profound insights, and encouragement have been very important in shaping the development of this work. We are truly thankful for the opportunity to learn under her support and for the inspiration she provided throughout this academic journey.

We would also like to express our appreciation to my friends and family. Their encouragement, understanding, and unwavering support have been the pillars that sustained us throughout this academic period.

In addition, We would like to acknowledge how proud We are for successfully completing the Electronics and Communication Engineering program of Istanbul Technical University. This accomplishment represents not only the culmination of our academic efforts but also a testament to the skills and knowledge we have gained during our time at the university.

January 2024

Ekin Türkü ERDOĞAN
Telat IŞIK

TABLE OF CONTENTS

Page

ABBREVIATIONS

SoC **:** System on Chip
RISC **:** Reduced Instrcution Set Computer
PQC **:** Post-Quantum Cryptography
RSA **:** Rivest-Shamir-Adleman
ECC **:** Elliptic Curve Cryptography
NIST **:** National Institure of Standards and Technology
VPN **:** Virtual Private Network
SVP **:** Shortest Vector Problem
CVP **:** Closest Vector Problem
ISA **:** Instruction Set Architecture
TCL **:** Tool Command Language
MHz **:** Megahertz
Gbps **:** Gigabits per second

s

SYMBOLS

$\boldsymbol{\tau}$	**:** Tau
γ	**:** Gamma
$\boldsymbol{\beta}$	**:** Beta
r	**:** Bit-rate
c	**:** Capacity

LIST OF TABLES

Page

LIST OF FIGURES

Page

SYSTEM ON CHIP DESIGN FOR POST-QUANTUM CRYPTOGRAPHY USING RISC-V PROCESSOR

SUMMARY

This project revolves around the development of a System-on-Chip (SoC) design intended for the robust implementation of the post-quantum cryptography algorithm known as CRYSTALS-Dilithium. The overarching goal is to seamlessly integrate this algorithm into a RISC-V processor-based system while ensuring optimal performance and security. The project unfolds through several key phases:

Understanding the Significance of Post-Quantum Cryptography:

In the initial phase, the project delves into the realm of post-quantum cryptography, investigating its crucial importance in the face of evolving quantum computing threats. By scrutinizing the vulnerabilities inherent in existing cryptographic systems, the project aims to establish the necessity of transitioning to post-quantum algorithms.

Comprehensive Exploration of CRYSTALS-Dilithium Algorithm:

A thorough examination of the CRYSTALS-Dilithium algorithm forms the foundation of the project. This includes acquiring in-depth knowledge of its mathematical foundations and cryptographic principles. The objective is to grasp the intricacies that make CRYSTALS-Dilithium a compelling choice for secure communication in the post-quantum era.

Source Code Examination and Compilation:

The project proceeds to acquire and meticulously review the source code associated with the CRYSTALS-Dilithium algorithm. This step is crucial to gain insights into the software aspects of the algorithm. Subsequently, the source code is compiled to ensure its proper functionality within the system.

Profiling and Performance Analysis:

A pivotal aspect of the project involves profiling the compiled source code to pinpoint specific functions or modules that consume a significant portion of the running time. Through performance analysis, the project endeavors to identify areas for optimization, paving the way for enhanced efficiency.

Hardware Design Research for RISC-V Core Integration:

With a focus on hardware solutions, the project embarks on researching and selecting suitable designs for integration with a RISC-V processor. The aim is to identify hardware acceleration techniques and optimizations that align seamlessly with the requirements of the CRYSTALS-Dilithium algorithm. The ultimate objective is to augment the overall system performance.

Essentially, during the hardware design research phase, we're not just looking into different options; we're carefully evaluating each one based on how straightforward, efficient, and compatible it is. This method guarantees that the chosen hardware not only improves the overall system performance but also does it in a way that is easy to understand and in harmony with the specific requirements of the CRYSTALS-Dilithium algorithm. The ultimate goal is to create a secure and optimized System-on-Chip (SoC) design that can effectively tackle the challenges presented by the imminent threats of quantum computing.

In conclusion, this project aspires to not only comprehend the theoretical foundations of post-quantum cryptography and the intricacies of the CRYSTALS-Dilithium algorithm but also to manifest a tangible, secure, and optimized SoC design. By leveraging the capabilities of a RISC-V processor, the project seeks to address the imminent challenges posed by quantum computing threats, resulting in a robust and efficient solution.

RISC-V İŞLEMCİ KULLANARAK KUANTUM SONRASI KRİPTOGRAFİ İÇİN ÇİP ÜZERİNDE SİSTEM TASARIMI

ÖZET

Her geçen gün, kuantum bilgisayarlarının kullanımının artması; beraberinde bazı güvenlik sorunlarını da beraberinde getirmiştir. Günümüzde kullanılan kriptografi algoritmalarının bu sorunlar karşısında yetersiz kalması yeni arayışları doğurmuştur. CRYSTALS-Dilithium algoritması bu arayışlar sonucu ortaya çıkan, yakın gelecekteki güvenlik problemlerine çözüm olabilecek bir Post-Quantum Cryptography algoritmasıdır. Bu algoritma NIST kurumunun başlattığı projeler doğrultusunda oluşturulmuş bir algoritmadır. Bizim projemizin amacı ise bir RISC-V işlemcisi kullanarak bu algoritmayı koşturabilecek bir SoC dizayn oluşturmaktır. Sadece günümüzü değil, gelecekte oluşabilecek durumları da değerlendirmek amacıyla bu proje oldukça kiritiktir.

Bu proje sürecinde çeşitli adımlar ve araştırmalar yapılmıştır. Öncelikli olarak, PQC konseptini ve önemini idrak etmek amacıyla araştrmalar yapılmıştır. Veri güvenliği şüphesiz her alanda çok kritiktir. Yapılan araştırmalar doğrultusunda da bu konudaki önemin her geçen gün daha da artacağı gözlemlenmiştir. Özellikle bankacılık, bulut sistemleri ve tüketici bilgileri gibi konularda gereken güvenlik koşulları, günümüz algoritmalarıyla güvende olmayacağı için; PQC algoritmalarının yeri oldukça kritiktir. PQC algoritmalarının genel özellikleri ve önemi anlaşıldıktan sonra, seçilen algoritma olan CRYSTALS-Dilithium algoritmasının özellikleri ve çalışma prensiplerini anlamak amacıyla araştırma yapılmıştır. Bu algoritmanın matematiksel modelleri ve içerisinde barındırdığı problem tipleri araştırılıp öğrenmilmiştir. Yapılan incelemeler sonucu, bu algoritmanın diğer PQC algoritmaları gibi günümüz algoritmalarından çok daha güvenli olduğu, quantum siber saldırıları karşısında oldukça güçlü olduğu görünmüştür.

Bu algoritmanın kaynak kodları incelenmiş ve koşturulmuştur. Bu adım algoritmanın yazılımsal özelliklerini görmek adına önemli bir kaynak olmuştur. Algoritmada kullanılan fonskiyonlar tek tek derlenip sonuçları incelenip karşılaştırılmıştır. Daha sonrasında gerçekelştirilen yazılım profili sonucunda, algoritmanın hangi fonksiyonunun en çok zaman harcadığı ve en fazla çağrıldığı tespit edilmiştir. Bunun amacı şudur: İşlem zamanını en çok harcayan fonksiyon için tasarlanacak ve RISC-V işelmcisine uyumlu olacak bir donanım; bütün sistemin daha hızlı çalışmasını sağlayacaktır.

Esasen, donanım tasarımı araştırma aşaması sadece mevcut seçeneklerin keşfini içermiyor; aynı zamanda her birini basitlik, verimlilik ve uygunluk temelinde titiz bir değerlendirme sürecinden geçiriyoruz. Bu yaklaşım, seçilen donanım çözümünün sadece genel sistem performansını artırmakla kalmayıp aynı zamanda CRYSTALS-Dilithium algoritmasının şifreleme incelikleriyle de uyumlu bir şekilde çalışmasını sağlar. Bu çabaların sonucunda, yakın gelecekteki kuantum hesaplama tehditleriyle başa çıkabilen güvenli ve optimize edilmiş bir System-on-Chip (SoC) tasarımı ortaya çıkması beklenmektedir.

Profilleme sonucunda en çok zaman harcayan fonksiyon KeccakF1600_StatePermute fonskyionu olmuştur. Bu fonksiyon, PQC algoritmaları arasında yaygın olarak kullanılan bir fonksiyon olduğu için öncelikli hedef literatürdeki donanım tasarımlarını araştırmak olarak belirlenmiştr. Yapılan araştırmalarla birlikte donanın opsiyonları incelenmiş ve bir tanesi seçilmiştir. Daha sonra seçilen donanım ile ilgili daha detaylı bir araştırma yapılıp dokümente edilmiştir.

1. INTRODUCTION

As this project unfolds within a collaborative effort involving Istanbul Technical University, Sabancı University, and TÜBİTAK, facilitated by the 1001 Project, the carefully planned trajectory of our study has been charted through regular, weekly online meetings within the project group. At its core, the overarching goal of this endeavor is the development of a RISC-V processor capable of executing Post-Quantum Cryptography (PQC) algorithms. These PQC algorithms stem from a competitive initiative sponsored by the National Institute of Standards and Technology (NIST), underscoring their significance in the landscape of cryptographic advancements.

In accordance with our project advisor's guidance, we have chosen the Dilithium algorithm as the focal point of our graduation project. This deliberate choice has led us to engage in an in-depth exploration of the Dilithium algorithm, dissecting its intricacies, comprehending the underlying code structures, and scrutinizing the outcomes produced by this cryptographic mechanism. The ensuing chapters of our thesis will unfold the layers of our research, illuminating the complexities and insights gained during our examination of the Dilithium algorithm, thereby contributing to the broader field of Post-Quantum Cryptography.

2. POST-QUANTUM CRYPTOGRAPHY

2.1 What Is PQC?

Post-Quantum Cryptography (PQC) is a branch of cryptography that mainly focuses on creating cryptographic algorithms that can withstand attacks from quantum computers [1]. Many contemporary cryptographic techniques, such as Rivest-Shamir-Adleman (RSA) and elliptic curve cryptography (ECC), are vulnerable to quantum computers. Since an attack made from a quantum compute is powerful and efficient when it comes to solving specific mathematical issues, the traditional cryptography algorithms fall short against attacks from quantum computers.

To address this issue, PQC algorithms are based on mathematical problems that are thought to be difficult to solve even by quantum computers. These algorithms are designed to provide security in a future with large-scale quantum computers. Lattice-based cryptography, code-based cryptography, multivariate polynomial cryptography, hash-based cryptography, and many other methodologies are common in various PQC algorithms [2].

National Institute of Standards and Technology (NIST) of the United States launched a program and competition in en effort to standardize PQC algorithms [3]. The NIST PQC Standardization process, which began in 2016, involves the evaluation and selection of novel cryptographic algorithms capable of defending against attacks from both classical and quantum computers. Multiple rounds of submissions, reviews, and public scrutiny are part of the process. The goal of NIST in this standardization process is to replace traditional algorithms with new PQC algorithms that will be used globally.

Because it protects the long-term security and integrity of digital communications and data in the post-quantum age, PQC is an important area of research. PQC algorithms seek to deliver secure cryptographic solutions in the future by inventing algorithms that are resistant to quantum computer attacks [4].

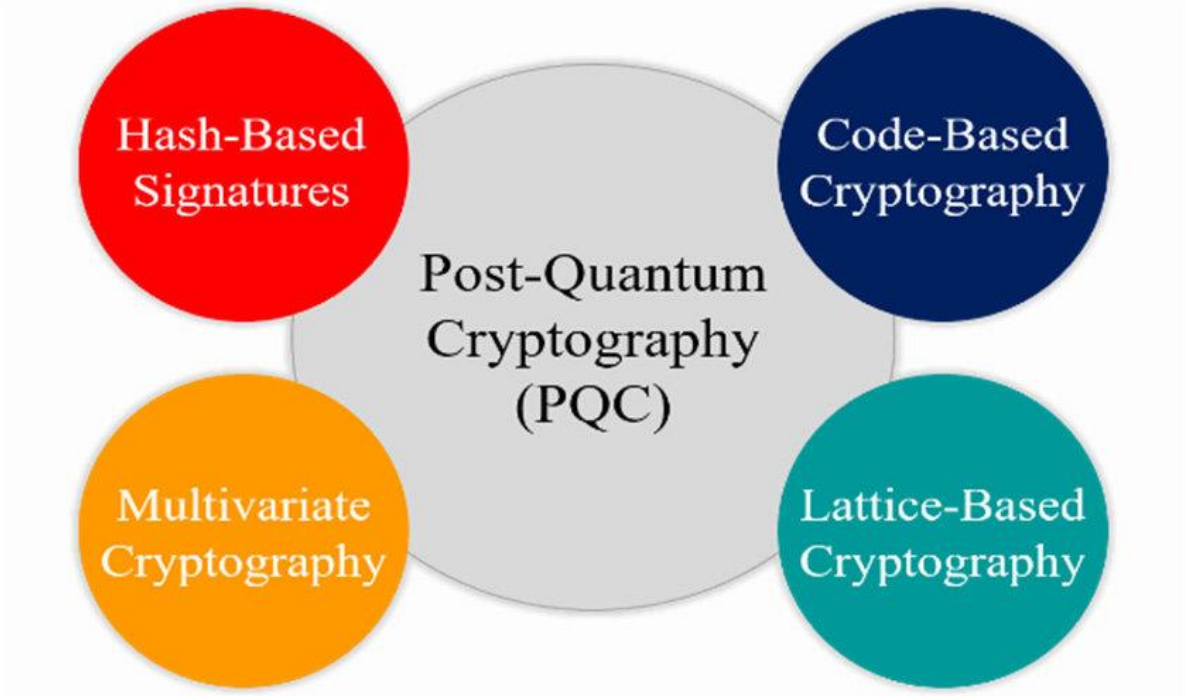

Figure 2.1 : Basic Types of PQC Algorithms

2.1.1 The importance of PQC algorithms

The significance of PQC algorithms rests in mitigating the potential danger to data security posed by quantum computers. It guarantees data security even in the presence of sophisticated quantum computing capabilities. PQC seeks to provide sensitive data with long-term security. Organizations and individuals can future-proof their cryptographic systems and ensure the secrecy, integrity, and authenticity of their data over time by switching from traditional algorithms to PQC algorithms.

2.1.2 Usage areas of PQC algorithms

PQC techniques can play an important role in many domains where data integrity is critical. Some examples of various application areas include insdustries such as defense, banking, manufacturing, retail and many more. Digital signatures are commonly used to authenticate digital documents. PQC algorithms can provide post-quantum secure signature techniques, ensuring message non-repudiation and protecting against cyber threats.

PQC algorithms can be used to secure and assure the integrity of stored data. This is especially important in instances where sensitive information must be kept secure for an extended period of time, such as bank records, healthcare databases, or vital infrastructure systems. Virtual private network (VPN) solutions can use PQC algorithms to offer safe

and private communication between remote networks or persons. The integrity of data transported across the VPN can be preserved by incorporating PQC algorithms into VPN protocols.

PQC algorithms can also be employed in communication protocols to safeguard data and allow it to be safely transported across networks.

2.2 Lattice Based Cryptography

2.2.1 What is lattice?

It is necessary to comprehend the meaning of lattice before delving into the explanation of lattice-based cryptography. A collection of points having a periodic structure inside an n-dimensional space is called a lattice structure. Formally speaking, the collection of vectors formed by n-linearly independent vectors b1,...,bn Rn is called the lattice [8]. Figure 2.2 shows an example of a 2D lattice created using three distinct basis vectors.

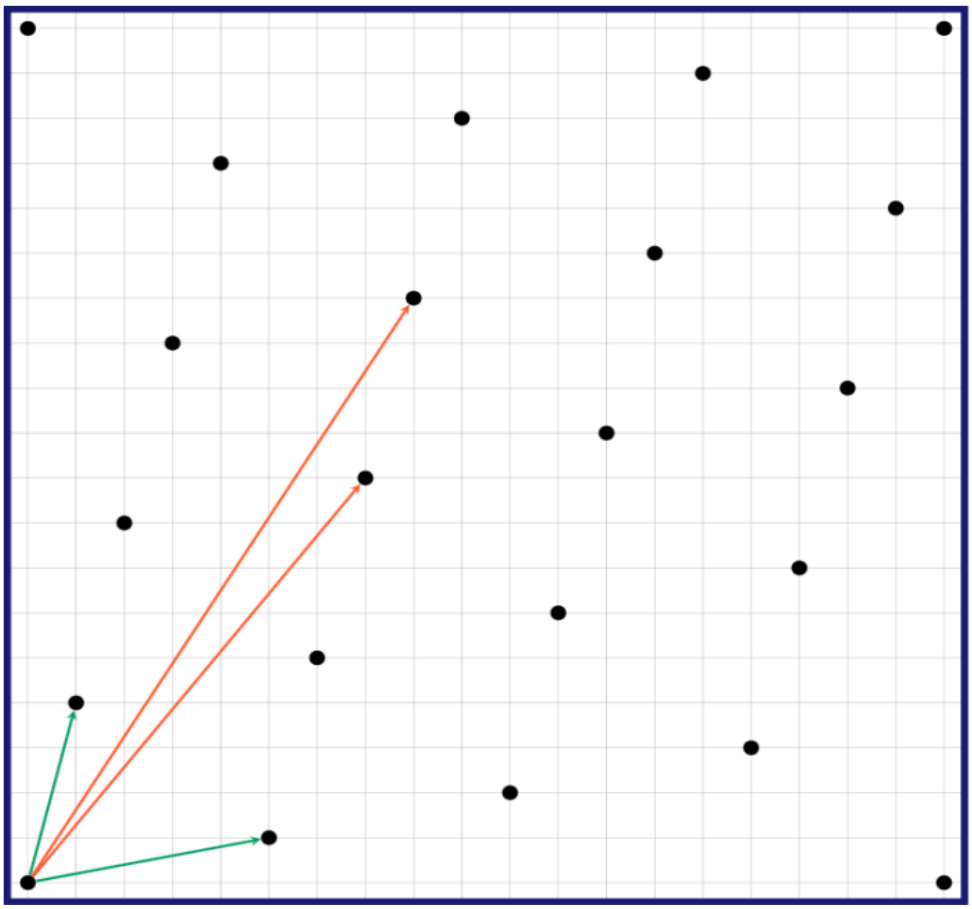

Figure 2.2: 2D lattice generated by 3 different base vectors

2.2.2 What is lattice based cryptography?

Lattice-based techniques that rely on solving computational lattice issues are widely acknowledged as being secure and challenging. Because of this, the field of PQC algorithms frequently use these techniques. As it stated in [8] there is no polynomial time algorithm that approximates lattice problems to within polynomial factors. Altough there are many lattice-based problems, there are two important problems that stand out for cryptography algorithms which are Shortest Vector Problem (SVP) and Closest Vector Problem (CVP).

2.2.2.1 Shortest vector problem

The SVP has three different versions as well. The first variant involves finding the shortest non-zero vector, the second aims to determine the length of the shortest non-zero vector, and the last one focuses on verifying if the length of the shortest non-zero vector is shorter than a specified real number [9]. An example to SVP can be seen in Figure 2.3 below.

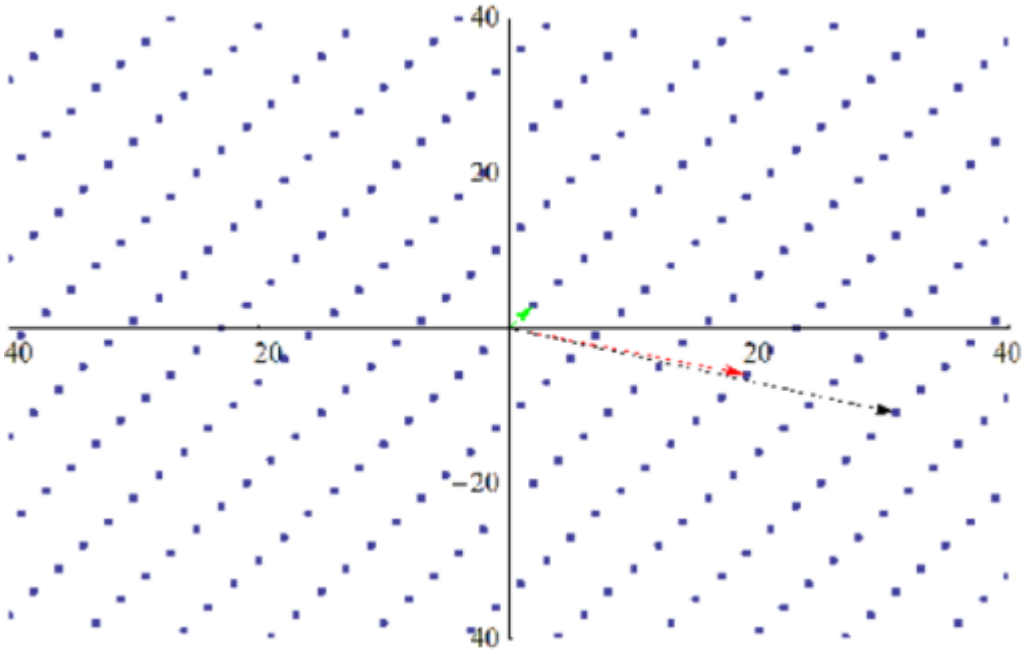

Figure 2.3: SVP example [10]

2.2.2.2 Closest vector problem

The SVP and CVP are very similar. The distinction is that the issue seeks to locate the lattice point that is closest to the target vector given a basis of a lattice and a target. An illustration of CVP is provided in Figure 2.4 below.

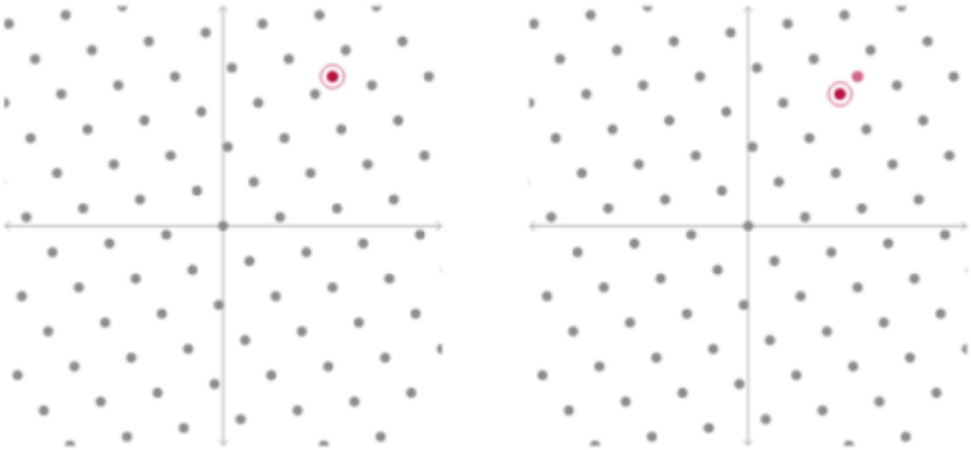

Figure 2.4: CVP example [11]

2.2.3 Lattice based cryptography and PQC

As stated earlier, lattice based problems form the basis for many algorithms in the field of PQC. The general reason behind this is certain lattice problems are difficult even for quantum computers. With that motivation, it is believed that quantum computers are not capable enough to solve lattice problems. Schemes based on lattice problems are being actively researched as potential replacements for traditional cryptography algorithms.

Like the CRYSTALS-Dilithium algorithm, the NIST is leading an effort to standardize the PQC algorithms and mostly lattice based algorithms. It is a sure thing that PQC algorithms need to be secure but also they need to be efficient. Lattice based algorithms have a big advantage when it comes to efficiency.

To summarize, lattice-based problems are central to the development of PQC algorithms. The algorithm which we study on is also a lattice based PQC algorithm of NIST.

3. LITERATURE REVIEW ON RISC-V PROCESSORS

3.1 What Is RISC-V?

RISC-V is an open-source instruction set architecture (ISA) used for the development of custom processors that are aiming a variety of applications. RISC stands for Reduced Instruction Set Computer and the RISC-V is designed for the purposes like high performance and power efficiency. The start of this project goes back to 2010 in University of California, Berkeley. In the default version, there are four types of instructions which are called base instructions: R-, I-, S- and U- Type. The structures of these instructions are shown in Figure3.1.

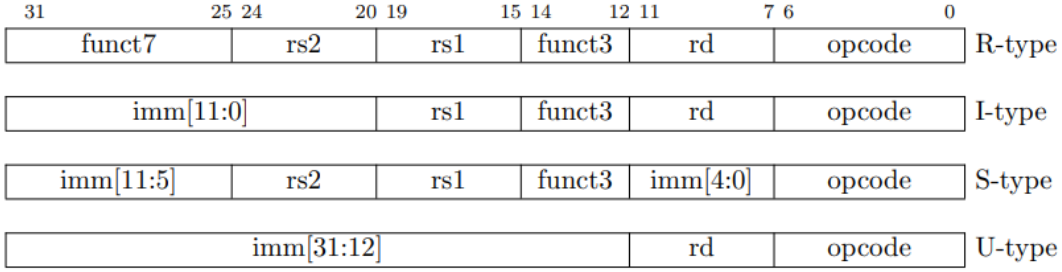

Figure 3.1: RISC-V base instruction set format [6]

3.2 RISC-V Applications

The versatility and open architecture of RISC-V make it a good choice for various applications across different industries. The RISC-V specifications are maintained by the RISC-V Foundation, an international consortium of members dedicated to creating an open ISA. Both hardware and software studies are also available in the website of the RISC-V Foundation [6].

Since the RISC-V ISA is an open source and flexible, it is suitable to many applications. It is suitable for embedded system designs due to its simplicity and customizable nature. It is a great option where an efficient and small sized processor is required. Thanks to its power efficiency RISC-V processors are also good options for IoT devices like sensors. This ISA is also widely used for academic purposes because it is free and simple. Last but not least, RISC-V processors are good options

for security purposes like in this project. Realizing PQC algorithms and building cloud computing systems can be great examples to security purposed applications of RISC-V processors.

3.3 Hornet RISC-V Core

The Hornet RISC-V core is a processor core that uses the RISC-V ISA with the "M" extension. Yavuz Selim Tozlu and Yasin Yilmaz created it as part of their graduation project at Istanbul Technical University [7]. The core is intended for microcontroller applications, providing an efficient and adaptable solution.

The Hornet RISC-V core strives to optimize performance and simplicity by adhering to the principles of RISC design. It is built on the RISC-V ISA, which enables a modular and adaptable architecture. The RISC-V ISA's "M" extension is specifically designed to handle microcontroller functionality such as interrupts, low-power modes, and efficient exception handling.

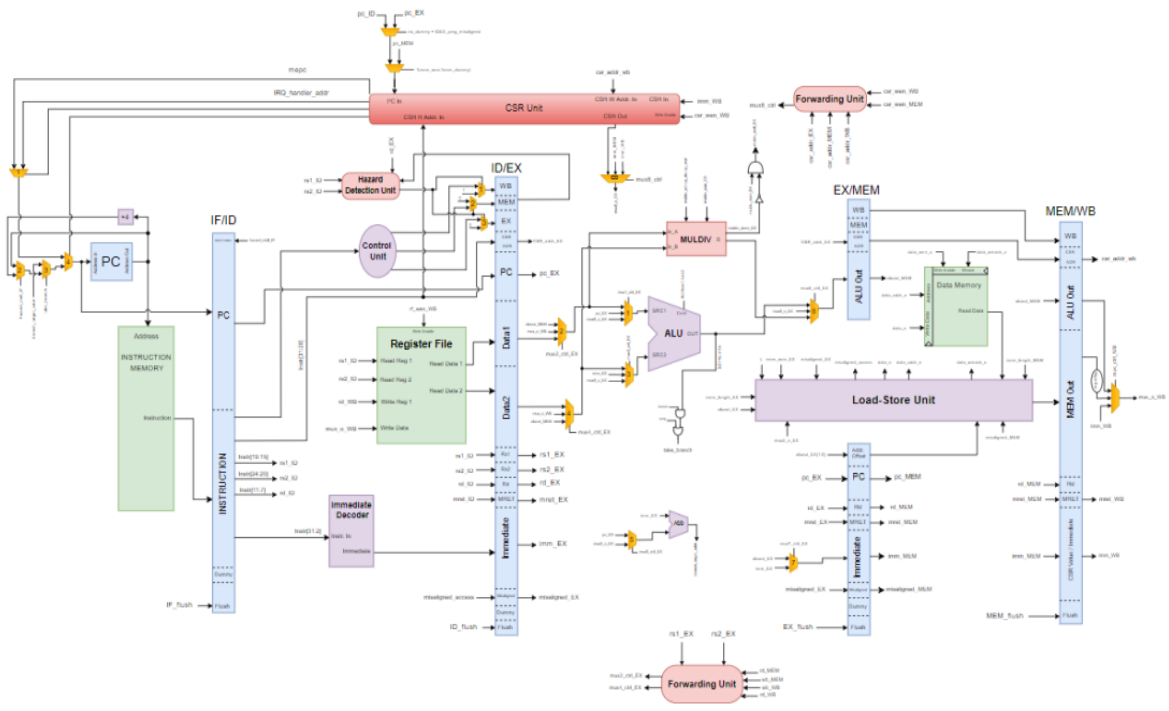

Figure 3.2: Pipeline Diagram of the Hornet Core

4. IMPLEMENTATION AND TESTING OF HORNET CORE

4.1 Implementation on Xilinx Vivado

To implement the Hornet core, Xilinx Vivado program was chosen. The installation of Vivado can be achieved form the official Xilinx website which is shown in Figure 4.1.

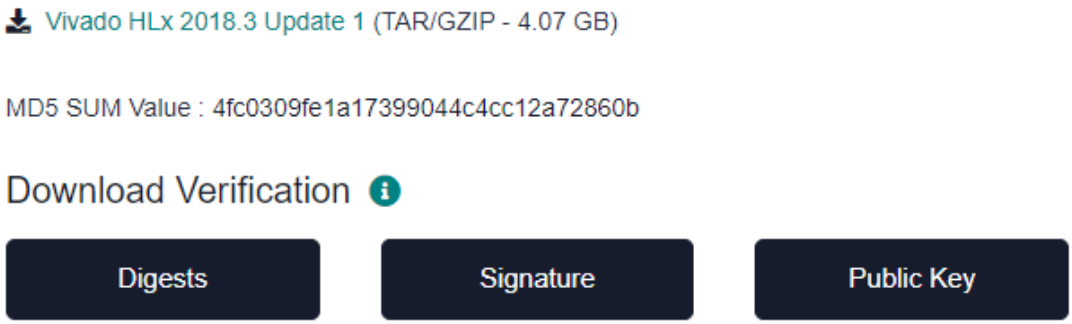

Figure 4.1: Xilinx website

It should be noted that there are multiple versions of Vivado. For this project version 2018.3 should suffice considering the recent versions will take too much disk space. After installation of Vivado, the initial program screen should look like as given in Figure 4.2.

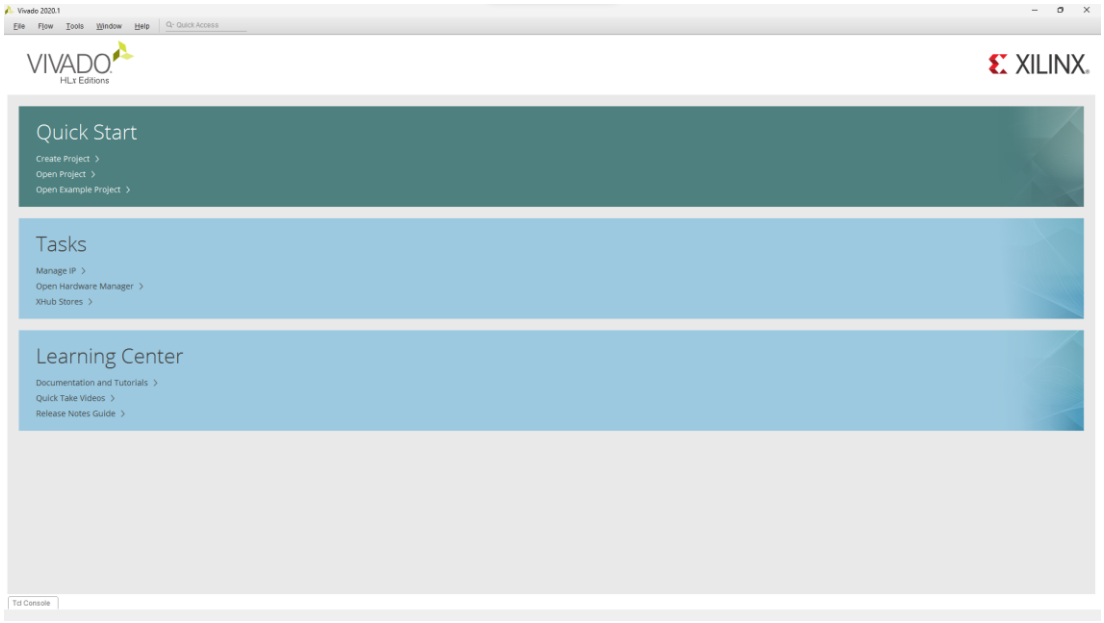

Figure 4.2: Vivado interface

To implement the Hornet core, codes of the project should be downloaded from the GitHub repository available in [yavuz tozlu]. The github repository can be seen in Figure 4.3. After the download is complete, from the Vivado interface a new project should be created and every file shown Figure 4.4 should be addded into the newly created Vivado project. After adding the souce files mentioned and opening the project the hierarchy of the project should look like as given in Figure 4.4.

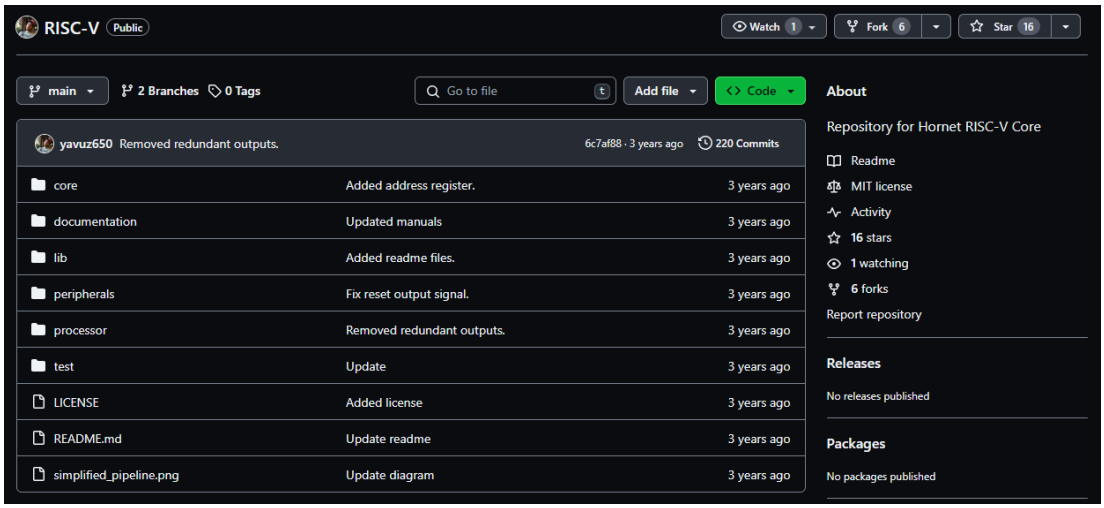

Figure 4.3: Sources of the Hornet Core vivado projects

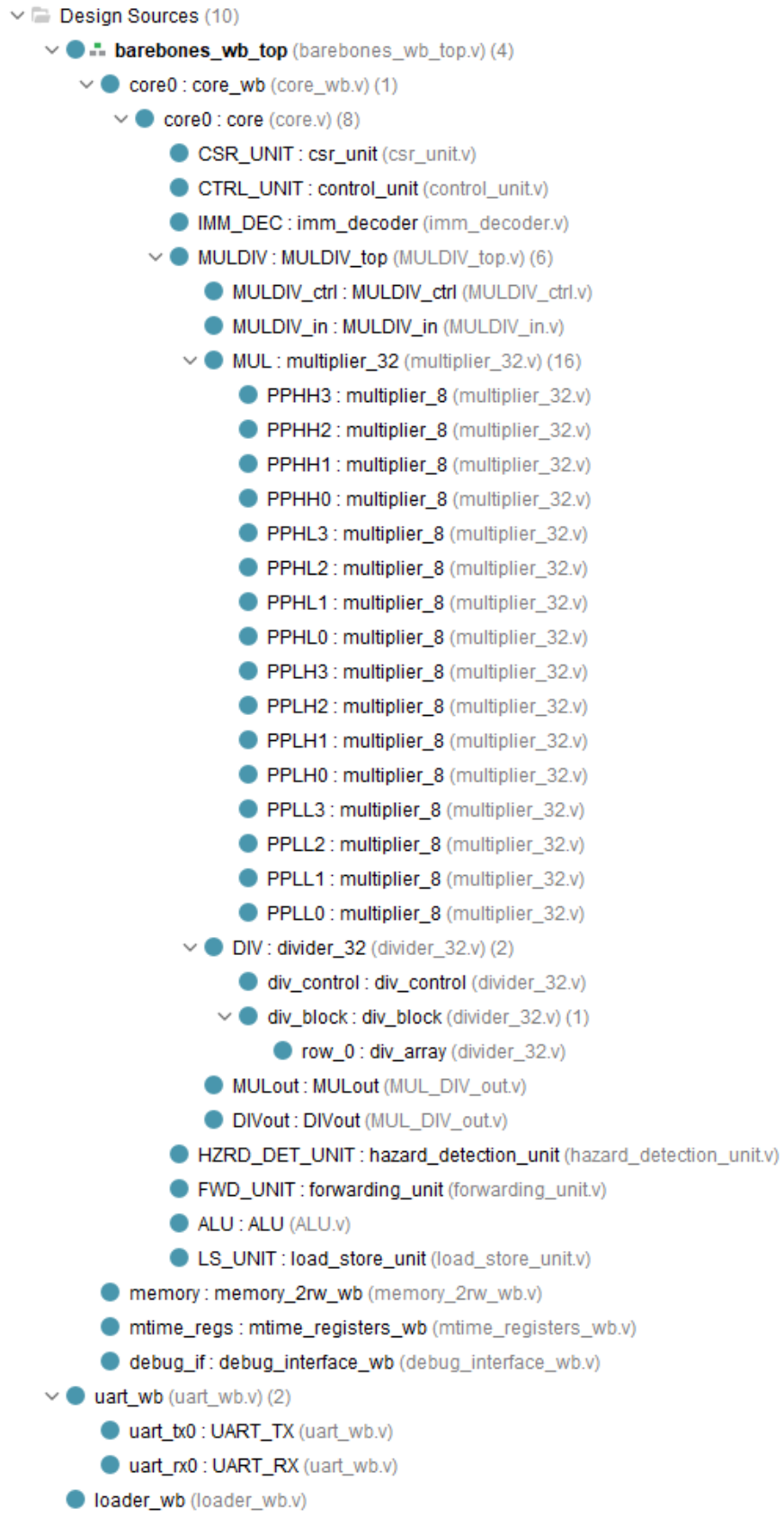

Figure 4.4: Sources of the Hornet Core vivado project

4.2 Synthesis and Implementation Steps

The Hornet Core, developed by Istanbul Technical University's Yavuz Selim TOZLU and Yasin YILMAZ, was specifically designed and optimized for RISC-V processor implementation. Choosing the "Hornet" Core takes use of the university's experience and research background in the field of RISC-V architecture. We can benefit from the experience and knowledge gained through the creation of an established and well-tested core, assuring a reliable and efficient foundation for our PQC processor.

To take synthesis and implementation of the project simply “Run Synthesis” and “Run Implementation” as fiven in Figure 4.5. A successfull implementation will ensure the project is working corretly. The option to view the RTL schematic is also available and shown in Figure 4.5. RTL scheme of the core and the whole system are shown in Figure 4.6 and Figure 4.7.

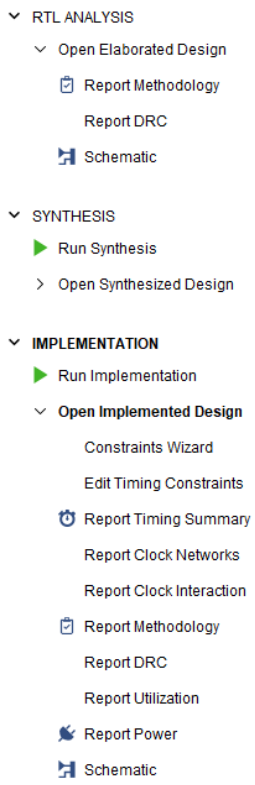

Figure 4.5: RTL schematic, synthesis and implementation tools of Vivado

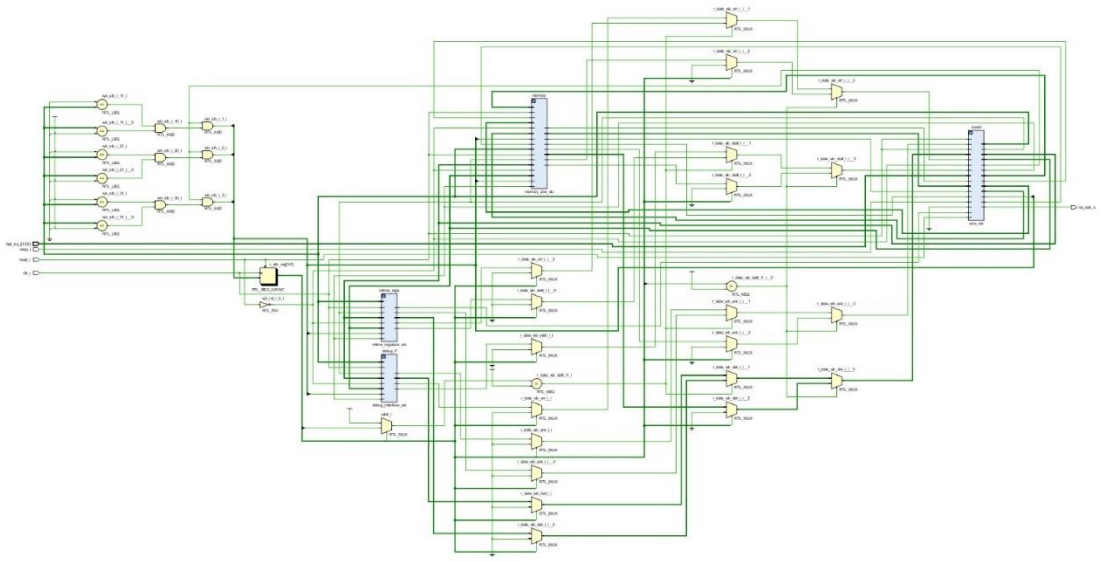

Figure 4.6: RTL schematic of the whole project

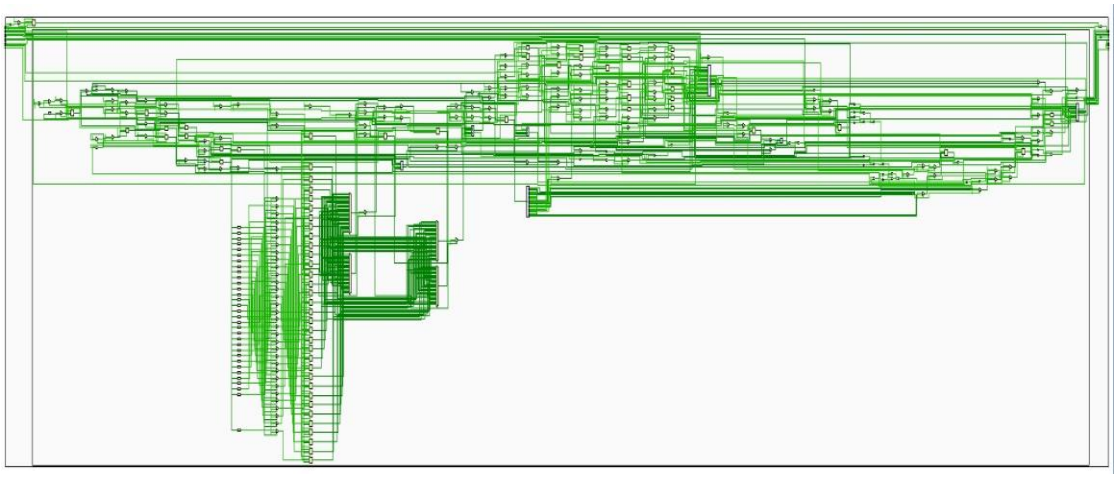

Figure 4.7: RTL schematic of the core module

The post implementation layout of the system on the AMD Arty Artix-7 100T FPGA board is shown in Figure 4.8. The post-implementation utilization report in Figure 4.9 shows the utilized assets of the used FPGA.

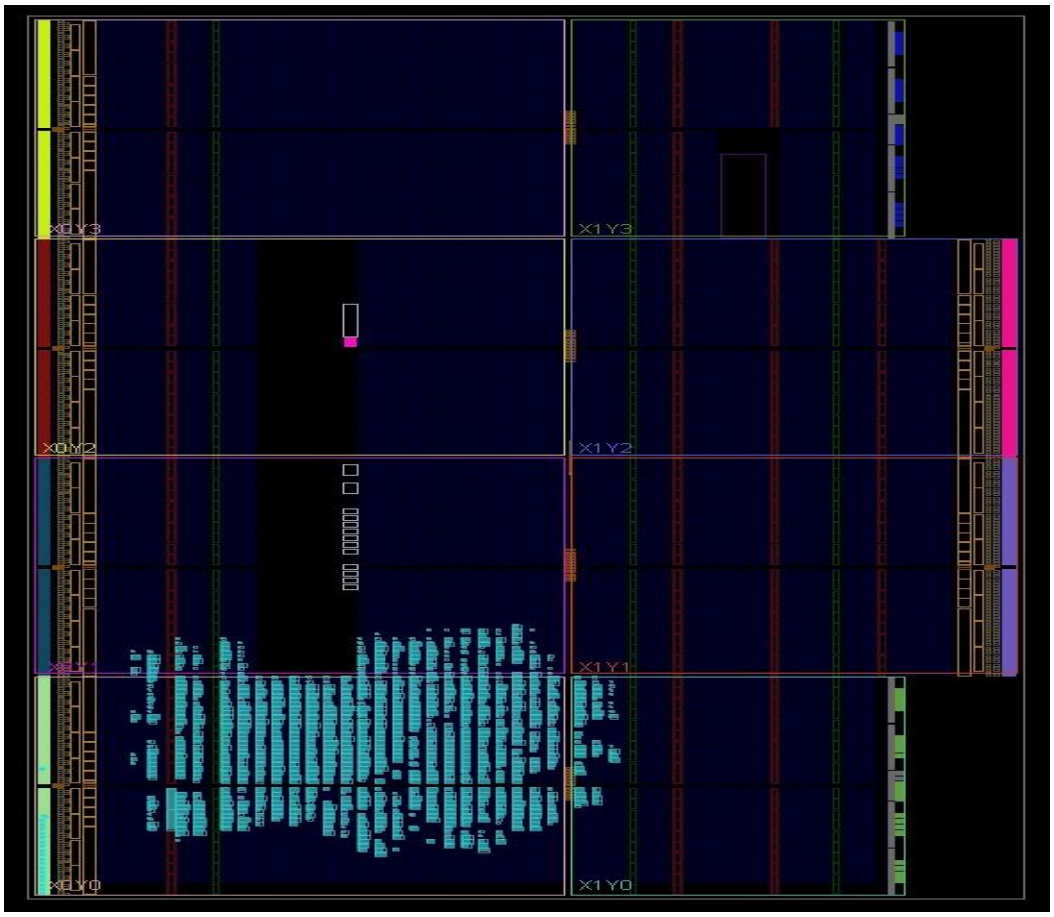

Figure 4.8: Post implementation layout of the Hornet Core System

Resource	Utilization	Available	Utilization %
LUT	5393	63400	8.51
FF	2485	126800	1.96
BRAM	2	135	1.48
IO	20	300	6.67

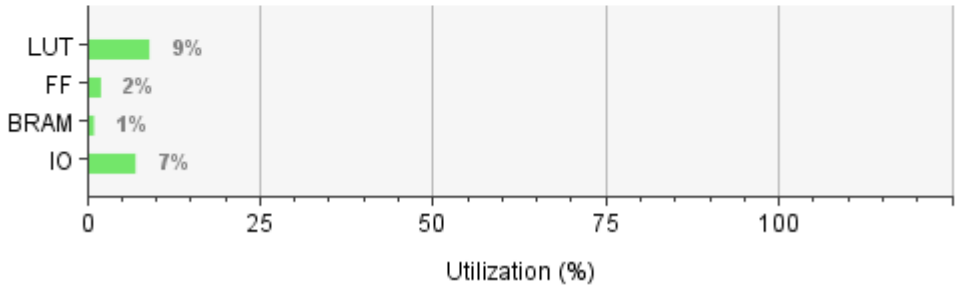

Figure 4.9: Post implementation utilization report Hornet Core System

4.3 Simulation with Available Souces

The simulation and testing of the Hornet Core can be done from the waveform which is given in Figure 4.10. Looking through the simulation waveform can prove helpful when specific signals need to be viewed. Although this method can be useful, the provided testbench in the Hornet files automatically checks whether the run code works correctly or not. This way the programs run can be checked directly from the TCL console.

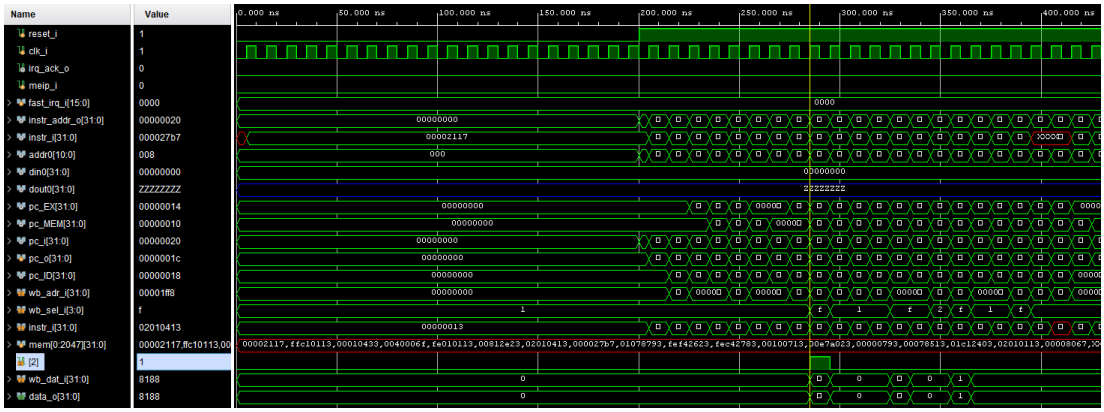

Figure 4.10: Post implementation utilization report Hornet Core System

As it will be mentined in the following sections, considering the way the testbench works it is extremely helpful to manipulate a single address in the C codes that are run the Hornet core. The value of the address 0x00002010 will automatically be checked form the testbench. If the result is correct the testbench expects a value of 1, and 0 if unsuccessful. The TCL console output and the testbench are seen in Figure 4.11 and Figure 4.12 repectively.

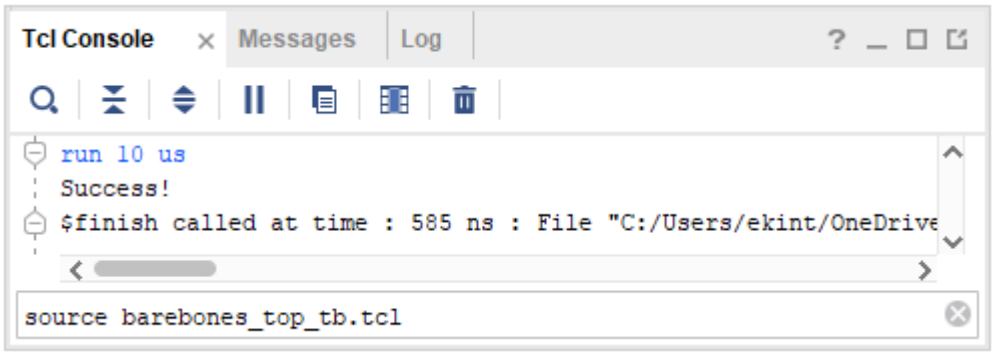

Figure 4.11: TCL console output

```
always @(posedge wb_clk_i or posedge wb_rst_i)
begin
    if(wb_rst_i) begin end

    else
    begin
        if(wb_cyc_i && wb_stb_i && wb_we_i)
        begin
            if(wb_dat_i == 32'b1)
            begin
                $display("Success!");
                $finish;
            end

            else
            begin
                $display("Failure!");
                $finish;
            end
        end
    end
end
```

Figure 4.12: Testbench code

4.4 RISC-V GNU Toolchain

Any text editor can be used to edit the algorithm's codes. However, the RISC-V GNU Toolchain is needed for the compilation process, and it can only be found on Linux systems.

4.4.1 Setup

Ubuntu is chosen for this project since it is based on Linux. As we did in this project, the most popular approach to utilize Ubuntu is to put it up as a virtual machine on top of a Windows operating system. Oracle's VirtualBox software is used as our Ubuntu virtual machine [12]. Once the Ubuntu virtual machine is operational, the project environment needs to be configured.

In order to install the toolchain, command lines have been used like the most software installations in Linux systems. These commans can be accessed through a terminal application on the Linux machine.

As a first step of installing the toolchain on the system, the repository must be cloned with the command below:

- git clone https://github.com/riscv/riscv-gnu-toolchain

Other prerequired software can be installed with the command below:

- sudo apt-get install autoconf automake autotools-dev curl python3 libmpc-dev libmpfr-dev libgmp-dev gawk build-essential bison flex texinfo gperf libtool patchutils bc zlib1g-dev libexpat-dev

After that, current directory should be changed with the directory of the toolchain with the command below:

- cd riscv-gnu-toolchain

The command down below will configure the toolchain:

- ./configure –-prefix=/opt/riscv --with-multilib-generator="rv32iilp32--;rv32im-ilp32--"

And lastly, the toolchain can be installed by the following command:

- sudo make

This installation appproximately lasts an hour.

4.4.2 Verification of the setup

In order to verify the toolchain, we compiled a test C program and generated a binary file for the RISC-V processor. The test C program is called bubble sort program and used from an earlier graduation project of our professor [7]. This program is a simple program based on the bubble sort game. The source code bubble_sort.c is shown in Figure 4.13.

```c
#include "string.h"
#include <stdio.h>
#define DEBUG_IF_ADDR 0x00002010

void bubble_sort(int* arr, int len)
{
    int sort_num;
    do
    {
        sort_num = 0;
        for(int i=0;i<len-1;i++)
        {
            if(*(arr+i) > *(arr+i+1))
            {
                int tmp = *(arr+i);
                *(arr+i) = *(arr+i+1);
                *(arr+i+1) = tmp;
                sort_num++;
            }
        }
    }
    while(sort_num!=0);
}

int main()
{


    int unsorted_arr[] = {195,14,176,103,54,32,128};
    int sorted_arr[] = {14,32,54,103,128,176,195};
    bubble_sort(unsorted_arr,7);

    int *addr_ptr = (int *)DEBUG_IF_ADDR;
    *addr_ptr = 2;
    printf("0x%08x\n", addr_ptr);

    if(memcmp((char*) sorted_arr, (char*) unsorted_arr, 28) == 0)
    {
        //success
        //*addr_ptr = 1;
        printf("success\n");
    }
    else
    {
        //failure
        //*addr_ptr = 0;
        printf("failure\n");
    }
    printf("0x%08x\n", addr_ptr);
    printf("%d\n", *addr_ptr);
    return 0;
}
```

Figure 4.13: Code of the bubble sort program

The following steps have been followed to test the toolchain. Firstly, the repository of the project needs to be cloned:

- git clone https://github.com/yavuz650/RISC-V.git

Changing the directory:

- cd RISC-V/test/bubble_sort

In order to achieve the binary file, first we need to achieve the Executable and Linkable File format (.elf) of this program:

- riscv64-unknown-elf-gcc bubble_sort.c ../crt0.s –march=rv32i – mabi=ilp32 –T ../linksc.ld –nostartfiles –ffunction-sections –fdatasections –Wl,--gc sections –o bubble_sort.elf

And lastly, the binary file gets generated from the .elf file with the command below:

- riscv64-unknown-elf-objcopy –O binary –j .init –j .text –j .rodata bubble_sort.elf bubble_sort.bin

Once the binary file has been generated, the program is compiled successfully. But there is one more step before this file gets loaded into the processor's memory. This binary file needs to be converted into another format called .data file. To do that, there is a simple C program called rom_generator in the same repository.

The command sequence down below is set to generate the .data file:

- cd ..

g++ rom_generator.c –o rom_generator

cd bubble_sort

../rom_generator bubble_sort.bin

Now, the .data file is ready for loading into the processor`s memory.

5. CRYSTALS-DILITHIUM PQC ALGORITHM

5.1 What Is CRYSTALS-Dilithium Algorithm?

CRYSTALS-Dilithium algorithm is one of the post-quantum cryptography algorithms submitted to the NIST. As it stated in [13], CRYSTALS-Dilithium is a lattice based digital signature scheme whose security is based on the hardness of finding short vectors in lattices.

5.2 Design Criterias And Characteristics Of CRYSTALS-Dilithium

There are some design criterias of the CRYSTALS-Dilithium signature algorithm. The main idea behind these design criterias is to make the algorithm efficient and easy adaptive. Main criterias and their descriptions are down below [13].

- **Simple to implement:** This is one of the most cruical aspects of the CRYSTALS-Dilithium algorithm beacuse; as we mentioned before, the threats that cyber attacks with quantum computers will be very common in the short future based on the studies about PQC. So implementation of such an algorithm needs to be simple for a wider usage.
- **Conservative with parameters**: Selecting a conservative approach to choosing parameters for algorithms allows long-term security. This criteria makes the algorithm be adaptive over a long time of period.
- **Minimum size of (key + signature):** The sum of these parameters are designed to be minimum because many applications of the algorithm require the transmission of both public key and the signature. As it stated in [13], the CRYSTALS-Dilithium algorithm has the smallest size when it comes the combination of the public key and signature sizes amongs the lattice based schemes with the same level of security.
- **Being modular:** With thise criteria of the algorithm and its functions, the usage are of the algorithm varies. This enables the algorithm to get implemented in many areas where the security is needed.

5.3 Overview Of The Basic Approach Of The CRYSTALS-Dilithium

According to the [13], the design of the scheme is based on the "Fiat-Shamir with Aborts" approach. And the simpilfied verison of the scheme is shown in Figure 5.1.

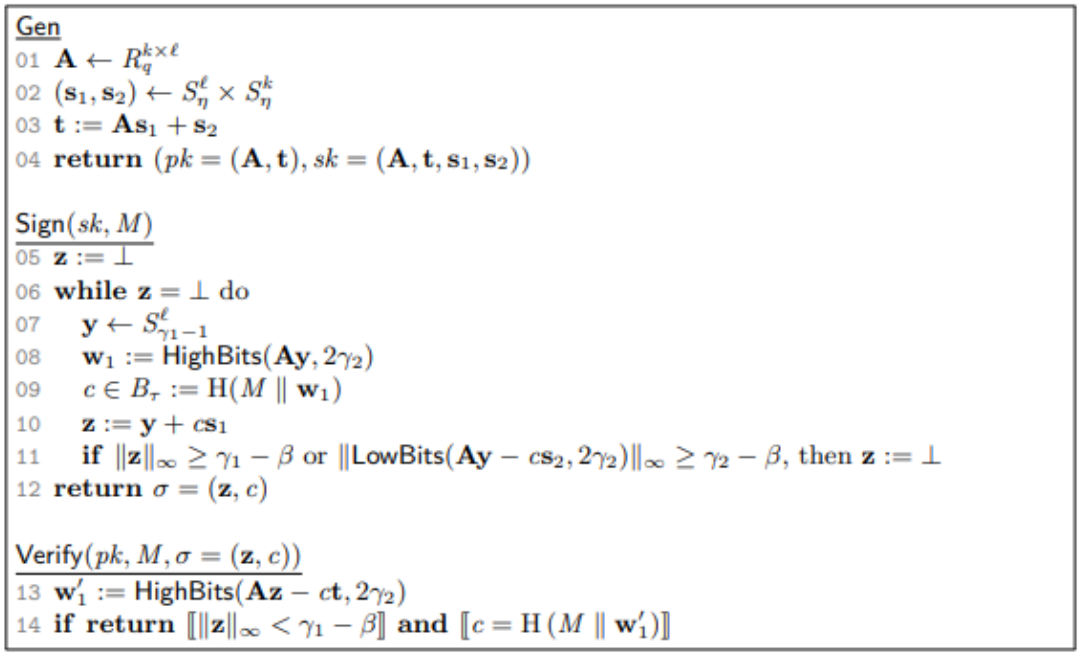

Figure 5.1: Template of the signature scheme without public key compression

5.4 Key Generation And Signature In The CRYSTALS-Dilithium

Since the CRYSTALS-Dilithium algorithm is a signature scheme, the two main parts of the algorithm are public key generation and signing procedure. Of course the algorithm itself is very much complicated than that but since this study is about implementing the algorithm on a RISC-V processor, we will not go through all of the details of the algorithm. All the information about the CRYSTALS-Dilithium algorithm can be found in [13].

5.4.1 Key generation

The main idea behind the key generation algorithm is to generate a matrix that is sized $k \times l$ which will be named A. Each entry of matrix A will be a polynomial in the ring equation given below.

$$R_q = \frac{\mathbb{Z}_q[X]}{X^n + 1} \tag{5.1}$$

Later on, the algorithm samples 2 random vectors called s_1 and s_2. The coefficient of the vectors is an element of R_q. Lastly, the second part of the public key is generated as given in the equation below.

$$\mathbf{t} = \mathbf{A}s_2 + s_1 \tag{5.2}$$

5.4.2 Signing procedure

In the signing part of the algorithm, a masking vector of polynomials y with coefficients less then γ_1 is generated. The parameter γ_1 is chosen strategically to ensure the signature remains zero-knowledge regarding the secret key and is difficult to forge.

The algorithm computes Ay and extracts the "high-order" bits w_1 from the coefficients in this vector. The challenge c is then created as the hash of the message and w_1, resulitng in a polynomial in R_q with exactly $\tau \pm 1$ non-zero coefficients. The potential signature is computed as given in equation 3. But to prevent leaking the secret key, rejection sampling is employed.

$$\mathbf{z} = \mathbf{y} + c\mathbf{s_1} \tag{5.3}$$

The parameter β is set to the maximum possible coefficient of cs1, and if any coefficient of z exceeds γ_1 - β, or if any coefficient of the low-order bits **Az - ct** surpasses γ_2 - β, the signing procedure is restarted. The loop continues until both conditions are met. Parameters are chosen to limit the expected number of repetitions.

5.5 Code Profiling Of The CRYSTALS-Dilithium

Code profiling is a method used to study how a program behaves when it runs. Its purpose is to measure how well the program performs, pinpoint areas where it may be slowing down, and comprehend how resources are being used. This involves gathering information about different aspects of the program's execution, like the time each function takes, how much memory is being used, and how often functions are called. The main objective of code profiling is to enhance and make the code more efficient.

In order to make the CRYSTALS-Dilithium algorithm run faster on the Hornet core, a custom hardware block is planned to be designed. Detecting the most repeated function on the algorithm is highly important because realizing that function with a external hardware block will increase the speed of the algorithm on the Hornet core. To find out which function is the most repeated, the code profiling method is applied.

The profiling operation script in the Makefile shown in Figure 5.2 and the result of the code profiling is shown in Figure 5.3.

```
profile: test/test_vectors.c $(KECCAK_SOURCES) $(KECCAK_HEADERS)
	$(CC) $(CFLAGS) -pg -O0 -o profile_test_vectors $< $(KECCAK_SOURCES)
	./profile_test_vectors
	$(PROF) -p profile_test_vectors gmon.out
```

Figure 5.2: Profiling script in the Makefile

```
Each sample counts as 0.01 seconds.
  %   cumulative   self              self     total
 time   seconds   seconds    calls  us/call  us/call  name
 21.79      1.63     1.63  1174176     1.39     1.39  KeccakF1600_StatePermute
 13.90      2.67     1.04                             main
 10.29      3.44     0.77                             looptop
  7.89      4.03     0.59   640000     0.92     0.92  pqcrystals_dilithium2_avx2_rej_uniform_avx
  5.08      4.41     0.38   271568     1.40     1.40  keccakx4_squeezeblocks
  4.28      4.73     0.32   252088     1.27     1.27  pqcrystals_dilithium2_avx2_decompose_avx
  3.34      4.98     0.25                             pqcrystals_dilithium2_avx2_invntt_avx
  2.67      5.18     0.20 10540000     0.02     0.02  load64
  2.27      5.35     0.17    50000     3.40     3.40  pqcrystals_dilithium2_avx2_polyt0_unpack
  2.01      5.50     0.15                             pqcrystals_dilithium2_avx2_ntt_avx
  1.87      5.64     0.14 40960000     0.00     0.00  nttidx
  1.87      5.78     0.14   506321     0.28     0.28  pqcrystals_dilithium2_avx2_poly_chknorm
  1.87      5.92     0.14   262088     0.53     0.53  pqcrystals_dilithium2_avx2_polyz_unpack
  1.60      6.04     0.12   292088     0.41     0.41  pqcrystals_dilithium2_avx2_poly_caddq
  1.60      6.16     0.12   209066     0.57     2.69  keccak_absorb
  1.60      6.28     0.12    50000     2.40     2.40  pqcrystals_dilithium2_avx2_polyt0_pack
  1.47      6.39     0.11   306321     0.36     0.36  pqcrystals_dilithium2_avx2_poly_reduce
  1.34      6.49     0.10   120000     0.83     0.83  pqcrystals_dilithium2_avx2_rej_eta_avx
  1.34      6.59     0.10                             _init
  1.07      6.67     0.08    50000     1.60     1.60  pqcrystals_dilithium2_avx2_polyz_pack
  1.07      6.75     0.08                             _looptop2
  0.67      6.80     0.05    80000     0.62     0.62  pqcrystals_dilithium2_avx2_power2round_avx
  0.67      6.85     0.05    50000     1.00     1.00  pqcrystals_dilithium2_avx2_polyt1_pack
  0.67      6.90     0.05    40000     1.25     2.52  pqcrystals_dilithium2_avx2_use_hint_avx
  0.60      6.95     0.04    90000     0.50     0.50  pqcrystals_dilithium2_avx2_polyeta_pack
  0.60      6.99     0.04    90000     0.50     0.50  pqcrystals_dilithium2_avx2_polyeta_unpack
  0.53      7.03     0.04  1071374     0.04     0.04  store64
  0.53      7.07     0.04   236034     0.17     0.17  pqcrystals_dilithium2_avx2_poly_add
  0.53      7.11     0.04   222088     0.18     0.18  pqcrystals_dilithium2_avx2_polyw1_pack
  0.53      7.15     0.04   173022     0.23     1.62  keccak_squeeze
  0.53      7.19     0.04   100000     0.40    11.01  keccak_absorb_once
  0.40      7.22     0.03    50000     0.60     0.60  pqcrystals_dilithium2_avx2_polyt1_unpack
  0.40      7.25     0.03                             nttunpack128_avx
  0.27      7.27     0.02   243022     0.08     0.08  keccakx4_absorb_once
  0.27      7.29     0.02   136044     0.15     0.15  keccak_init
  0.27      7.31     0.02   114184     0.18     0.18  rej_eta
  0.27      7.33     0.02    10000     2.00   100.78  pqcrystals_dilithium2_avx2_keypair
  0.13      7.34     0.01   640000     0.02     0.02  pqcrystals_dilithium2_avx2_poly_nttunpack
  0.13      7.35     0.01   209066     0.05     2.73  pqcrystals_dilithium_fips202_avx2_shake256_absorb
  0.13      7.36     0.01   160000     0.06     5.29  pqcrystals_dilithium2_avx2_poly_uniform_4x
  0.13      7.37     0.01   160000     0.06     1.46  pqcrystals_dilithium_fips202x4_avx2_shake128x4_squeezeblocks
```

Figure 5.3: Profiling results

Like the most of the cryptography algorithms, CRYSTALS-Dilithium algorithm is generally based on randomness. For that reason; to get the most precised results, the algorithm has been runned in a loop. Based on the results that shown in Figure 5.3, KeccakF1600_StatePermute function is the one that takes the most of the running time of the algorithm, it is also the one that gets called most in the algorithm.

In conclusion, the test_vectors algorithm has been run in a loop and based on the results KeccakF1600_StatePermute function will be the function that will run on the custom hardware block.

Since the KeccakF1600_StatePermute function is a widely used function in cryptography algorithms, doing a research on the earlier studies about implementing a new design for KeccakF1600_StatePermute function is decided as the first thing to do. Based on the results of the researching, hardware designs in the literature will be explained in detail later on this paper.

One hardware design is planned to be selected to add in this project. The design requires not only be able to run the KeccakF1600_StatePermute function, but also it requires to be suitable for the Hornet RISC-V core.

5.5.1 Keccak algorithm

Before getting in the detail of hardware designs of KeccakF1600_StatePermute function, it will be nice to know the keccak algorithm itself. The algorithm based on the permutation function f. It is designed for Keccak states with a constant length of b, where b = c + r bits. In this equation r is the bit-rate and the c is capacity. In order to generate a new message length that is a multiple of r, the algorithm first pads the input message. The data output block is the first r bits of the state, and compression is done at the conclusion. Some of the arguments for the KeccakF1600_StatePermute function are listed below [14].

- c = 1024
- r= 576
- f = 1600

6. HARDWARE DESIGNS FOR KECCAK ALGORITHM

Our analysis indicates that the Micromachines KECCAK implementation is the first hardware alternative. The proposed pipelined hardware configuration for the Keccak algorithm is shown in Figure 6.1.

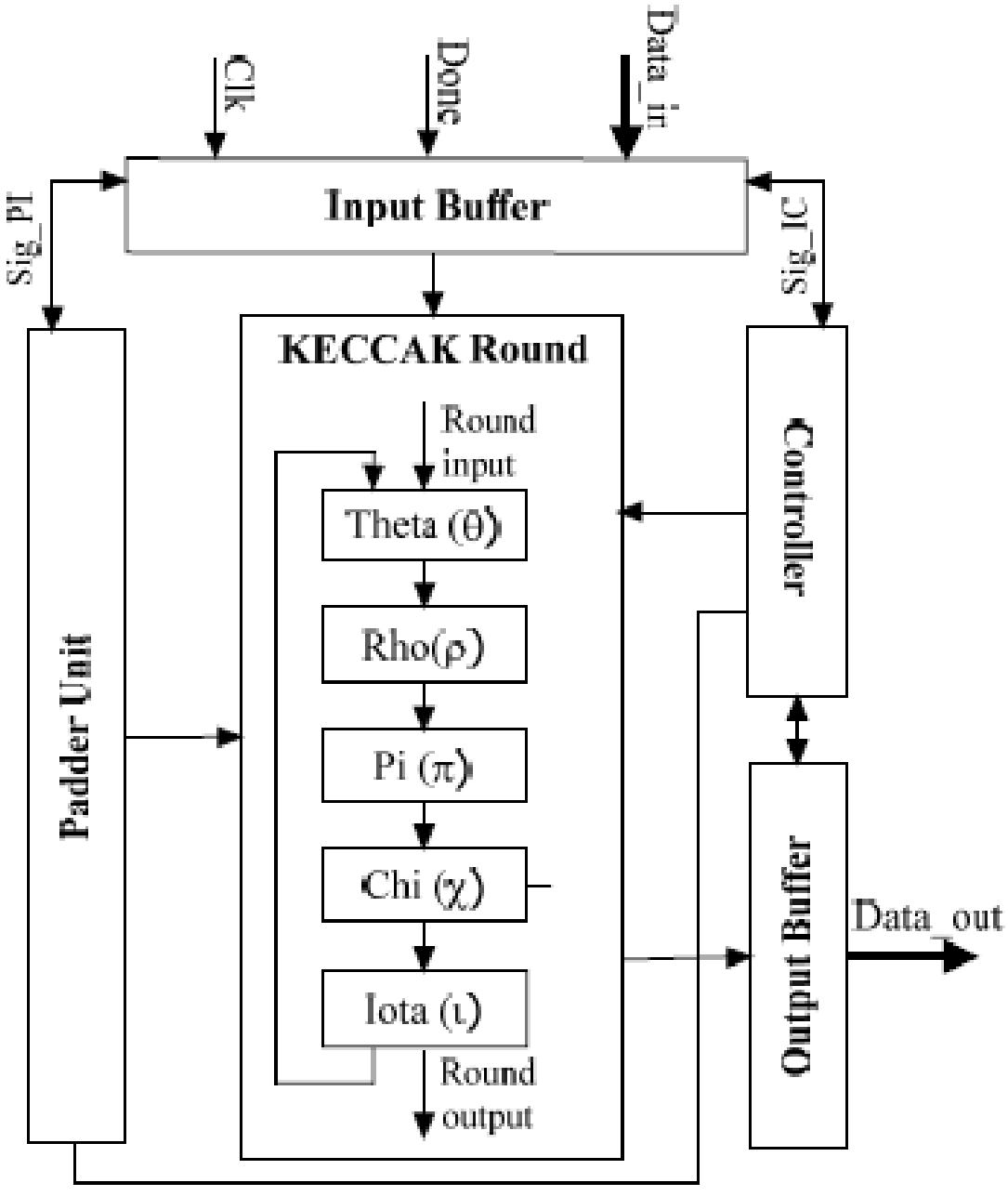

Figure 6.1: KECCAK Architecture of MicroMachines[14]

Second option is the study of wang shown in Figure 6.2.

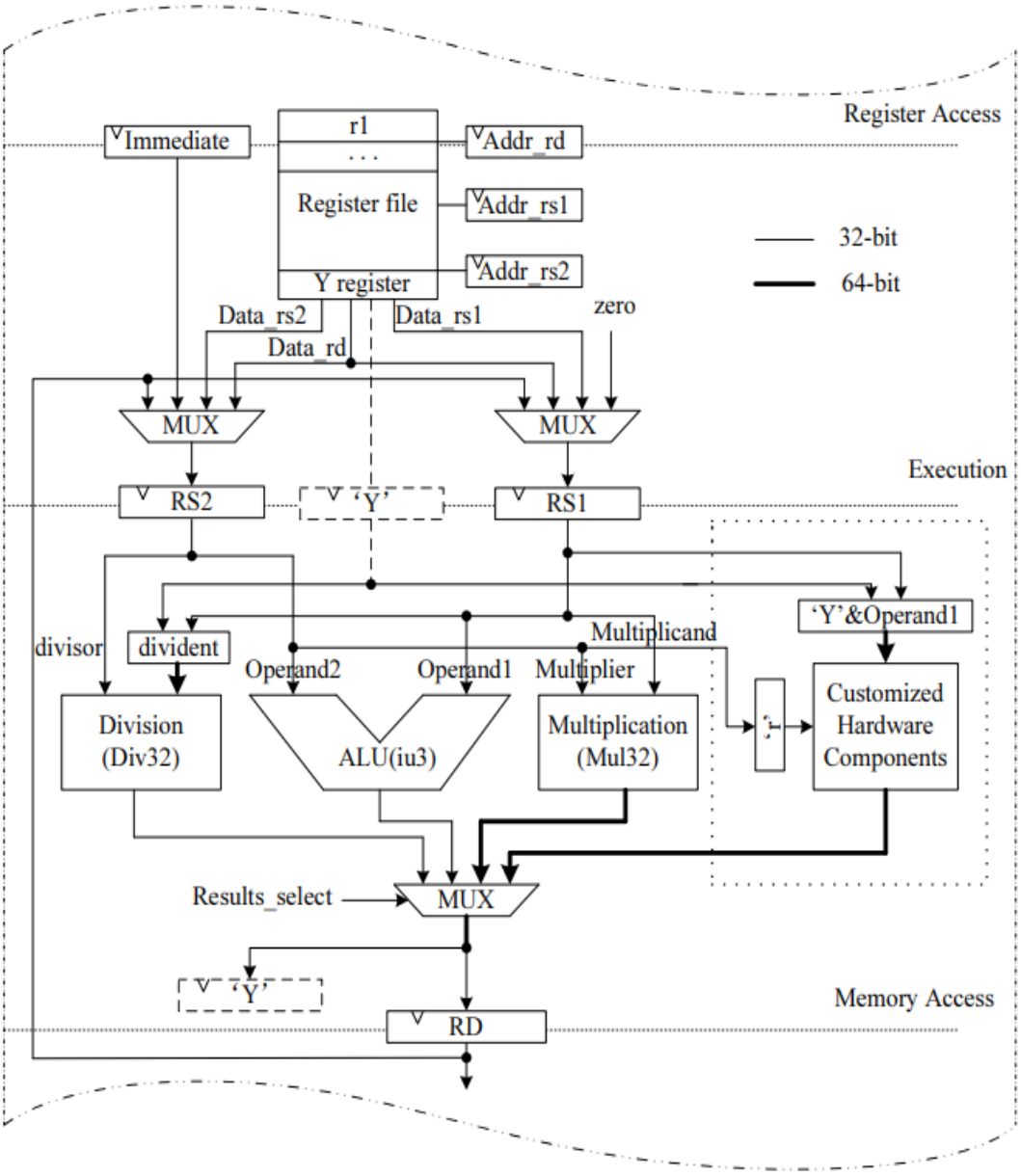

Figure 6.2: KECCAK Architecture of Wang [15]

Finally, the last solution is the design of Team Keccak, the RTL scheme shown in Figure 6.3 is generated with the source codes from their website using Vivado.

Figure 6.3: KECCAK Architecture of Team Keccak

All the designs mentioned above are quite similar to each other. Due to its simplicity, and high performance analysis, the study of Micromachines stands out the most. The explanations of the blocks shown in Figure 6.3 are below.

6.1 The Chosen Keccak Hardware Architecture

- **Input Buffer:** Provides connectivity for the Keccak round-input extarnel module in this design.
- **Padder Unit:** Carries out the padding operation and byte-by-byte inversion. The data flow between the Padder Unit and the Keccak core components is managed by a 2x1 multiplexer.

- **Controller:** Performs the synchronization of all the Keccak modules and their data transmission.
- **Keccak Round:** This is the core component of Keccak. The chain of operations get executed in this core.
- **Output Buffer:** Ensures the Keccak round-output connectivity in this architecture.
- **Theta Operation:** This will perform exclusive or operation on five columns of input bits.
- **Rho Operation:** Carries out a left rotation for each lanes uniqe member of positions.
- **Pi Operation:** Adjusts the lanes location in the Keccak.
- **Chi Operation:** A matrix consists of five rows and five lanes.
- **Iota Operation:** Executes an exclusive or operation on the initial lane.

FPGA hardware implementation details are shown in Table 5.1.

Design	**Area (Slice)**	**Freq. (MHz)**	**Throu. (Gbps)**	**Eff. (Mbps/Slice)**
[16]	5363	110	-	-
[17]	1365	326.38	7.83	5.73
[18]	1102	223	5.35	4.49
Micromachines Proposed	1370	258.6	10.77	7.96

Table 6.1: Implementation details of the KECCAK Architecture [14]

The KECCAK implementation in this study utilizes 1370 slices at a frequency of 258.6 MHz. The proposed design achieves a throughput of 10.77 Gbps with an efficiency of 7.96 Mbps per slice.

This architecture has been used to analyze the impact of fault attacks on the Keccak process. Tests were carried out by inserting single and multiple defects into KECCAK's input processes. The quantity of false bits produced by each operation at its output was then counted. The analysis's findings are displayed in Figure 6.4.

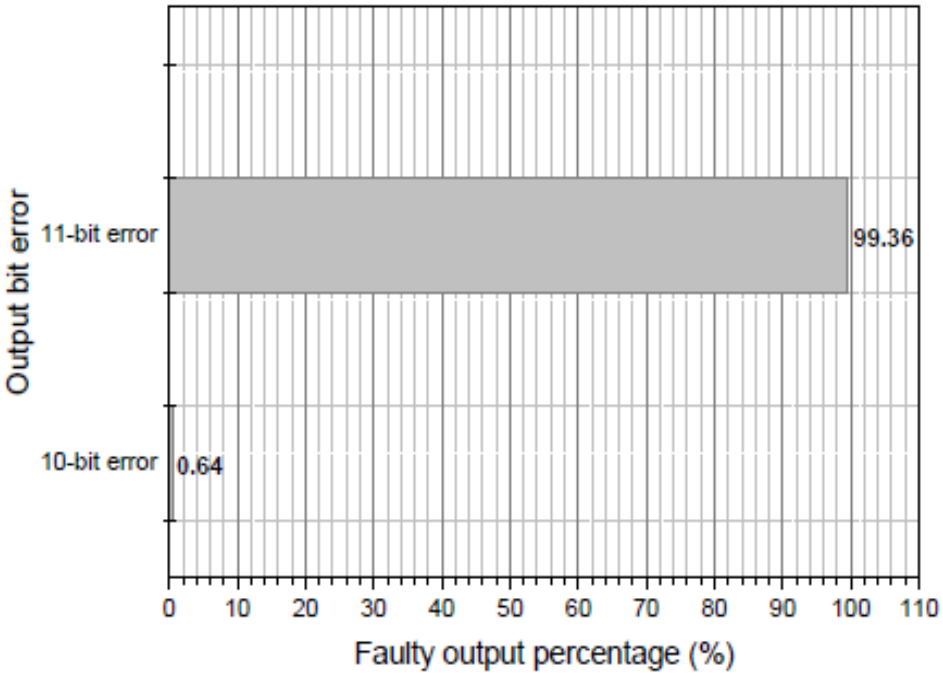

Figure 6.4: Fault analyze of the architecture [14]

7. REALISTIC CONSTRAINTS AND CONCLUSIONS

7.1 Practical Applications of this Project

The significance of purpose-specific hardware designs is growing steadily. Consequently, processors are crafted for distinct purposes within various architectures. In the case of the Hornet RISC-V processor, the addition of a PQC hardware will help improve the usage of the core significantly. The design is open for utilization and expansion, serving research, education, and personal projects alike.

7.2 Realistic Constraints

7.2.1 Social, environmental and economic impact

RISC-V operates as an ISA without licensing fees. This implies that companies or groups are not obligated to make payments for obtaining the license to produce or sell RISC-V processors; it is freely available for such purposes.

7.2.2 Cost analysis

Every single tool used in this project is free and open-source. If it is desired, he manufacturing of hardware will cost around $10000 which is quite costly.

7.2.3 Standards

The primary guide to adhere to in this project is the RISC-V ISA manual.

7.2.4 Health and safety concerns

This project arises no health and safety concerns.

7.3 Future Work and Recommendations

The future work on this project is decided as compiling all the codes of the CRYSTALS-Dilithium algorithm and connecting the KECCAK architecture with the Hornet core.

REFERENCES

[1] D. J. Bernstein and T. Lange, "Post-quantum cryptography," *Nature*, vol. 549, pp. 188-194, 2017.

[2] L. Ducas and D. Micciancio, "Lattice-Based Cryptography," in "Handbook of Elliptic and Hyperelliptic Curve Cryptography," CRC Press, 2018, pp. 1395-1446.

[3] I. T. L. Computer Security Division, "Post-quantum cryptography: CSRC," CSRC, https://csrc.nist.gov/projects/post-quantum-cryptography (accessed Jan. 7, 2024).

[4] C. Peikert, "A decade of lattice cryptography," *Foundations and trends® in theoretical computer science*, vol. 10, pp. 283-424, 2016.

[5] "Crystals ," Dilithium, https://pq-crystals.org/dilithium/index.shtml. (accessed Jan. 8, 2024).

[6] "V international – RISC-V: The open standard RISC instruction set architecture," RISC, https://riscv.org/ (accessed Jan. 8, 2024).

[7] Y. S. Tozlu and Y. Yılmaz, "Design and implementation of a 32-bit RISC-V core," senior design project, Electronics and Communication Eng. Dept., Istanbul Technical Univ., Istanbul, TURKEY 2021.

[8] Micciancio and O. Regev, "Lattice-based cryptography," in *Post-quantum cryptography*, ed: Springer, 2009, pp. 147-191.

[9] H. Nejatollahi, N. Dutt, S. Ray, F. Regazzoni, I. Banerjee, and R. Cammarota, "Post-quantum lattice-based cryptography implementations: A survey," ACM Computing Surveys (CSUR), vol. 51, pp. 1-41, 2019.

[10] Y.-L. Chuang, C.-I. Fan, and Y.-F. Tseng, "An efficient algorithm for the shortest vector problem," IEEE Access, vol. 6, pp. 61478-61487, 2018.

[11] D. Wong, "Lattices and Tikz posted August 2015," Lattices and Tikz, https://www.cryptologie.net/article/284/lattices-and-tikz/ (accessed Jan. 8, 2024).

[12] "Welcome to Virtualbox.org!," Oracle VM VirtualBox, https://www.virtualbox.org/ (accessed Jan. 8, 2024).

[13] V. Lyubashevsky, L. Ducas, E. Kiltz, T. Lepoint, P. Schwabe, G. Seiler, *et al.*, "Crystals-dilithium," *Algorithm Specifications and Supporting Documentation*, 2020.

[14] H. Mestiri and I. Barraj, "High-Speed Hardware Architecture Based on Error Detection for KECCAK," Micromachines, vol. 14, p. 1129, 2023.

[15] **Y. Wang, Y. Shi, C. Wang and Y. Ha**, "FPGA-based SHA-3 acceleration on a 32-bit processor via instruction set extension," *2015 IEEE International Conference on Electron Devices and Solid-State Circuits (EDSSC)*, Singapore, 2015, pp. 305-308, doi: 10.1109/EDSSC.2015.7285111.

[16] **P. Nannipieri, M. Bertolucci, L. Baldanzi, L. Crocetti, S. Di Matteo, F. Falaschi, et al.**, "SHA2 and SHA-3 accelerator design in a 7 nm technology within the European Processor Initiative," Microprocessors and Microsystems, vol. 87, p. 103444, 2021.

[17] **S. El Moumni, M. Fettach, and A. Tragha**, "High throughput implementation of SHA3 hash algorithm on field programmable gate array (FPGA)," Microelectronics journal, vol. 93, p. 104615, 2019.

[18] **M. Sundal and R. Chaves**, "Efficient FPGA implementation of the SHA-3 hash function," in 2017 IEEE Computer Society Annual Symposium on VLSI (ISVLSI), 2017, pp. 86-91.

APPENDICES

APPENDIX A: Makefile

APPENDIX A

```
SCC ?= /usr/bin/cc
CFLAGS += -Wall -Wextra -Wpedantic -Wmissing-prototypes -Wredundant-decls \
  -Wshadow -Wpointer-arith -mavx2 -mpopcnt -maes \
  -march=native -mtune=native -O3
NISTFLAGS += -Wno-unused-result -mavx2 -mpopcnt -maes \
  -march=native -mtune=native -O3
SOURCES = sign.c packing.c polyvec.c poly.c ntt.S invntt.S pointwise.S \
  shuffle.S consts.c rejsample.c rounding.c
HEADERS = align.h config.h params.h api.h sign.h packing.h polyvec.h poly.h ntt.h \
  consts.h shuffle.inc rejsample.h rounding.h symmetric.h randombytes.h
KECCAK_SOURCES = $(SOURCES) fips202.c fips202x4.c f1600x4.S symmetric-shake.c
KECCAK_HEADERS = $(HEADERS) fips202.h fips202x4.h
AES_SOURCES = $(SOURCES) fips202.c aes256ctr.c
AES_HEADERS = $(HEADERS) fips202.h aes256ctr.h

PROF = gprof

.PHONY: all shared clean

all: \
  test/test_dilithium2 \
  test/test_dilithium3 \
  test/test_dilithium5 \
  test/test_dilithium2aes \
  test/test_dilithium3aes \
  test/test_dilithium5aes \
  test/test_vectors2 \
  test/test_vectors3 \
  test/test_vectors5 \
  test/test_vectors2aes \
  test/test_vectors3aes \
  test/test_vectors5aes \
  test/deneme_test \
  speed

speed: \
  test/test_speed2 \
  test/test_speed3 \
  test/test_speed5 \
  test/test_speed2aes \
  test/test_speed3aes \
  test/test_speed5aes

shared: \
  libpqcrystals_dilithium2_avx2.so \
  libpqcrystals_dilithium3_avx2.so \
  libpqcrystals_dilithium5_avx2.so \
  libpqcrystals_dilithium2aes_avx2.so \
  libpqcrystals_dilithium3aes_avx2.so \
  libpqcrystals_dilithium5aes_avx2.so \
  libpqcrystals_fips202_avx2.so \
  libpqcrystals_fips202x4_avx2.so \
  libpqcrystals_aes256ctr_avx2.so
```

```
libpqcrystals_fips202_avx2.so: fips202.c fips202.h
	$(CC) -shared -fPIC $(CFLAGS) -o $@ $<

libpqcrystals_fips202x4_avx2.so: fips202x4.c fips202x4.h f1600x4.S
	$(CC) -shared -fPIC $(CFLAGS) -o $@ $< f1600x4.S

libpqcrystals_aes256ctr_avx2.so: aes256ctr.c aes256ctr.h
	$(CC) -shared -fPIC $(CFLAGS) -o $@ $<

libpqcrystals_dilithium2_avx2.so: $(SOURCES) $(HEADERS) symmetric-shake.c
	$(CC) -shared -fPIC $(CFLAGS) -DDILITHIUM_MODE=2 \
	  -o $@ $(SOURCES) symmetric-shake.c

libpqcrystals_dilithium3_avx2.so: $(SOURCES) $(HEADERS) symmetric-shake.c
	$(CC) -shared -fPIC $(CFLAGS) -DDILITHIUM_MODE=3 \
	  -o $@ $(SOURCES) symmetric-shake.c

libpqcrystals_dilithium5_avx2.so: $(SOURCES) $(HEADERS) symmetric-shake.c
	$(CC) -shared -fPIC $(CFLAGS) -DDILITHIUM_MODE=5 \
	  -o $@ $(SOURCES) symmetric-shake.c

libpqcrystals_dilithium2aes_avx2.so: $(SOURCES) $(HEADERS)
	$(CC) -shared -fPIC $(CFLAGS) -DDILITHIUM_MODE=2 -DDILITHIUM_USE_AES \
	   -o $@ $(SOURCES)

libpqcrystals_dilithium3aes_avx2.so: $(SOURCES) $(HEADERS)
	$(CC) -shared -fPIC $(CFLAGS) -DDILITHIUM_MODE=3 -DDILITHIUM_USE_AES \
	   -o $@ $(SOURCES)

libpqcrystals_dilithium5aes_avx2.so: $(SOURCES) $(HEADERS)
	$(CC) -shared -fPIC $(CFLAGS) -DDILITHIUM_MODE=5 -DDILITHIUM_USE_AES \
	   -o $@ $(SOURCES)

test/test_dilithium2: test/test_dilithium.c randombytes.c $(KECCAK_SOURCES) \
  $(KECCAK_HEADERS)
	$(CC) $(CFLAGS) -DDILITHIUM_MODE=2 \
	  -o $@ $< randombytes.c $(KECCAK_SOURCES)

test/test_dilithium3: test/test_dilithium.c randombytes.c $(KECCAK_SOURCES) \
  $(KECCAK_HEADERS)
	$(CC) $(CFLAGS) -DDILITHIUM_MODE=3 \
	  -o $@ $< randombytes.c $(KECCAK_SOURCES)

test/test_dilithium5: test/test_dilithium.c randombytes.c $(KECCAK_SOURCES) \
  $(KECCAK_HEADERS)
	$(CC) $(CFLAGS) -DDILITHIUM_MODE=5 \
	  -o $@ $< randombytes.c $(KECCAK_SOURCES)

test/test_dilithium2aes: test/test_dilithium.c randombytes.c $(AES_SOURCES) \
  $(AES_HEADERS)
	$(CC) $(CFLAGS) -DDILITHIUM_MODE=2 -DDILITHIUM_USE_AES \
	  -o $@ $< randombytes.c $(AES_SOURCES)

test/test_dilithium3aes: test/test_dilithium.c randombytes.c $(AES_SOURCES) \
```

```
test/test_dilithium3aes: test/test_dilithium.c randombytes.c $(AES_SOURCES) \
  $(AES_HEADERS)
	$(CC) $(CFLAGS) -DDILITHIUM_MODE=3 -DDILITHIUM_USE_AES \
	  -o $@ $< randombytes.c $(AES_SOURCES)

test/test_dilithium5aes: test/test_dilithium.c randombytes.c $(AES_SOURCES) \
  $(AES_HEADERS)
	$(CC) $(CFLAGS) -DDILITHIUM_MODE=5 -DDILITHIUM_USE_AES \
	  -o $@ $< randombytes.c $(AES_SOURCES)

test/test_vectors2: test/test_vectors.c $(KECCAK_SOURCES) $(KECCAK_HEADERS)
	$(CC) $(CFLAGS) -DDILITHIUM_MODE=2 \
	  -o $@ $< $(KECCAK_SOURCES)

test/test_vectors3: test/test_vectors.c $(KECCAK_SOURCES) $(KECCAK_HEADERS)
	$(CC) $(CFLAGS) -DDILITHIUM_MODE=3 \
	  -o $@ $< $(KECCAK_SOURCES)SS

test/test_vectors5: test/test_vectors.c $(KECCAK_SOURCES) $(KECCAK_HEADERS)
	$(CC) $(CFLAGS) -DDILITHIUM_MODE=5 \
	  -o $@ $< $(KECCAK_SOURCES)

test/test_vectors2aes: test/test_vectors.c $(AES_SOURCES) $(AES_HEADERS)
	$(CC) $(CFLAGS) -DDILITHIUM_MODE=2 -DDILITHIUM_USE_AES \
	  -o $@ $< $(AES_SOURCES)

test/test_vectors3aes: test/test_vectors.c $(AES_SOURCES) $(AES_HEADERS)
	$(CC) $(CFLAGS) -DDILITHIUM_MODE=3 -DDILITHIUM_USE_AES \
	  -o $@ $< $(AES_SOURCES)

test/test_vectors5aes: test/test_vectors.c $(AES_SOURCES) $(AES_HEADERS)
	$(CC) $(CFLAGS) -DDILITHIUM_MODE=5 -DDILITHIUM_USE_AES \
	  -o $@ $< $(AES_SOURCES)

test/test_speed2: test/test_speed.c test/speed_print.c test/speed_print.h \
  test/cpucycles.c test/cpucycles.h randombytes.c $(KECCAK_SOURCES) \
  $(KECCAK_HEADERS)
	$(CC) $(CFLAGS) -DDILITHIUM_MODE=2 \
	  -o $@ $< test/speed_print.c test/cpucycles.c randombytes.c \
	  $(KECCAK_SOURCES)

test/test_speed3: test/test_speed.c test/speed_print.c test/speed_print.h \
  test/cpucycles.c test/cpucycles.h randombytes.c $(KECCAK_SOURCES) \
  $(KECCAK_HEADERS)
	$(CC) $(CFLAGS) -DDILITHIUM_MODE=3 \
	  -o $@ $< test/speed_print.c test/cpucycles.c randombytes.c \
	  $(KECCAK_SOURCES)

test/test_speed5: test/test_speed.c test/speed_print.c test/speed_print.h \
  test/cpucycles.c test/cpucycles.h randombytes.c $(KECCAK_SOURCES) \
  $(KECCAK_HEADERS)
	$(CC) $(CFLAGS) -DDILITHIUM_MODE=5 \
	  -o $@ $< test/speed_print.c test/cpucycles.c randombytes.c \
	  $(KECCAK_SOURCES)


test/test_speed3aes: test/test_speed.c test/speed_print.c test/speed_print.h \
  test/cpucycles.c test/cpucycles.h randombytes.c $(AES_SOURCES) \
  $(AES_HEADERS)
	$(CC) $(CFLAGS) -DDILITHIUM_MODE=3 -DDILITHIUM_USE_AES \
	  -o $@ $< test/speed_print.c test/cpucycles.c randombytes.c \
	  $(AES_SOURCES)

test/test_speed5aes: test/test_speed.c test/speed_print.c test/speed_print.h \
  test/cpucycles.c test/cpucycles.h randombytes.c $(AES_SOURCES) \
  $(AES_HEADERS)
	$(CC) $(CFLAGS) -DDILITHIUM_MODE=5 -DDILITHIUM_USE_AES \
	  -o $@ $< test/speed_print.c test/cpucycles.c randombytes.c \
	  $(AES_SOURCES)

test/test_mul: test/test_mul.c randombytes.c $(KECCAK_SOURCES) \
  $(KECCAK_HEADERS)
	$(CC) $(CFLAGS) -UDBENCH -o $@ $< randombytes.c $(KECCAK_SOURCES)

PQCgenKAT_sign2: PQCgenKAT_sign.c rng.c rng.h $(KECCAK_SOURCES) \
  $(KECCAK_HEADERS)
	$(CC) $(NISTFLAGS) -DDILITHIUM_MODE=2 \
	  -o $@ $< rng.c $(KECCAK_SOURCES) $(LDFLAGS) -lcrypto

PQCgenKAT_sign3: PQCgenKAT_sign.c rng.c rng.h $(KECCAK_SOURCES) \
  $(KECCAK_HEADERS)
	$(CC) $(NISTFLAGS) -DDILITHIUM_MODE=3 \
	  -o $@ $< rng.c $(KECCAK_SOURCES) $(LDFLAGS) -lcrypto

PQCgenKAT_sign5: PQCgenKAT_sign.c rng.c rng.h $(KECCAK_SOURCES) \
  $(KECCAK_HEADERS)
	$(CC) $(NISTFLAGS) -DDILITHIUM_MODE=5 \
	  -o $@ $< rng.c $(KECCAK_SOURCES) $(LDFLAGS) -lcrypto

PQCgenKAT_sign2aes: PQCgenKAT_sign.c rng.c rng.h $(AES_SOURCES) $(AES_HEADERS)
	$(CC) $(NISTFLAGS) -DDILITHIUM_MODE=2 -DDILITHIUM_USE_AES \
	  -o $@ $< rng.c $(AES_SOURCES) $(LDFLAGS) -lcrypto

PQCgenKAT_sign3aes: PQCgenKAT_sign.c rng.c rng.h $(AES_SOURCES) $(AES_HEADERS)
	$(CC) $(NISTFLAGS) -DDILITHIUM_MODE=3 -DDILITHIUM_USE_AES \
	  -o $@ $< rng.c $(AES_SOURCES) $(LDFLAGS) -lcrypto

PQCgenKAT_sign5aes: PQCgenKAT_sign.c rng.c rng.h $(AES_SOURCES) $(AES_HEADERS)
	$(CC) $(NISTFLAGS) -DDILITHIUM_MODE=5 -DDILITHIUM_USE_AES \
	  -o $@ $< rng.c $(AES_SOURCES) $(LDFLAGS) -lcrypto


profile: test/test_vectors.c $(KECCAK_SOURCES) $(KECCAK_HEADERS)
	$(CC) $(CFLAGS) -pg -O0 -o profile_test_vectors $< $(KECCAK_SOURCES)
	./profile_test_vectors
	$(PROF) -p profile_test_vectors gmon.out

```

Figure A.1: Makefile

CURRICULUM VITAE

Name Surname	**: Ekin Türkü Erdoğan**
Place and Date of Birth	**: Izmir – 29.06.2000**
E-Mail	**: erdogane18@itu.edu.tr**

CURRICULUM VITAE

Name Surname **: Telat Işık**

Place and Date of Birth **: Istanbul – 26.11.2000**

E-Mail **: isikte18@itu.edu.tr**