

ISTANBUL TECHNICAL UNIVERSITY
ELECTRICAL-ELECTRONICS FACULTY

**Design and Implementation of a RISC-V Based Optimized Core for
High-Performance Cryptographic Algorithms**

SENIOR DESIGN PROJECT

Berk Arslanpençesi
Rabia Göksel Demiray

**ELECTRONICS AND COMMUNICATION ENGINEERING
DEPARTMENT**

JUNE 2025

ISTANBUL TECHNICAL UNIVERSITY
ELECTRICAL-ELECTRONICS FACULTY

PROJECT TITLE HERE
SECOND LINE IF NECESSARY
THIRD LINE IF NECESSARY, FIT TITLE IN THREE LINES

SENIOR DESIGN PROJECT

Berk ARSLANPENCESI
040220117
Rabia Goksel DEMIRAY
040220782

ELECTRONICS AND COMMUNICATION ENGINEERING
DEPARTMENT

Project Advisor: Prof. Dr. Siddika Berna Örs Yalcın

JUNE 2025

İSTANBUL TEKNİK ÜNİVERSİTESİ
ELEKTRİK-ELEKTRONİK FAKÜLTESİ

**Design and Implementation of a RISC-V Based Optimized Core for
High-Performance Cryptographic Algorithms**

LİSANS BİTİRME TASARIM PROJESİ

Berk Arslanpençesi
040220117

Rabia Göksel Demiray
040220782

Proje Danışmanı: Prof. Dr. Siddika Berna Örs Yalçın

ELEKTRONİK VE HABERLEŞME MÜHENDİSLİĞİ BÖLÜMÜ

HAZİRAN, 2025

We are submitting the Senior Design Project Report entitled as “Design and Implementation of a RISC-V Based Optimized Core for High-Performance Cryptographic Algorithms”. 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 .

Rabia Göksel DEMİRAY
040220782



Berk ARSLANPENÇESİ
040220117



FOREWORD

We would like to thank Prof. Dr. Sıddıka Berna ÖRS YALÇIN for her helpful guidance and support throughout this graduation project. We also thank our families for their constant encouragement and support.

This study was also supported by TÜBİTAK through the 2209-A Program. We would like to thank TÜBİTAK for their support.

June 2025

Berk ARSLANPENÇESİ
Rabia Göksel DEMİRAY

TABLE OF CONTENTS

	<u>Page</u>
FOREWORD	iv
TABLE OF CONTENTS	vii
ABBREVIATIONS	ix
SYMBOLS	x
LIST OF TABLES	xi
LIST OF FIGURES	xii
SUMMARY	xiii
ÖZET	xiv
1. INTRODUCTION	1
2. MATHEMATICAL BACKGROUND	3
2.1 Post Quantum Cryptography	3
2.2 Quantum Computers and Quantum Computing	4
2.3 Standardization of Post-Quantum Algorithms	5
2.4 Lattice-Based Cryptography	6
2.5 CRYSTALS-Kyber	7
2.5.1 Key Generation	8
2.5.2 Encryption	9
2.5.3 Decryption	10
3. SYSTEM REQUIREMENTS AND INSTALLATIONS	13
3.1 Processor Determination for the Project	13
3.1.1 Hornet	13
3.2 RISC-V GNU Toolchain	15
3.2.1 RISC-V GNU-Toolchain Installation	15
3.3 Downloading CRYSTALS-Kyber	17
4. LITERATURE REVIEW	19
4.1 Originality of Our Project	20
5. IMPLEMENTATION OF CRYSTALS-KYBER ON RISC-V	23
5.1 Running Applications on RISC-V Architecture	23
5.2 Execution of the CRYSTALS-Kyber Algorithm in Software	25
5.3 Optimizing CRYSTALS-Kyber For Hardware	26
5.4 Code Profiling For The CRYSTALS-Kyber Algorithm	29
5.5 Number Theoretic Transform	31
5.6 Butterfly	33
5.7 Modulo Multiplication	34
5.8 Montgomery Reduction Algorithm	35
6. INSTRUCTION SET EXTENSION OF RISC-V PROCESSOR	39
6.1 Replacing Montgomery with K2RED in Software and Testing K2RED on RISC-V	39
6.2 Instruction Set Extension for K2RED	40
6.2.1 Introducing a New Instruction to the RISC-V GNU Toolchain	41
6.2.2 Calling the Defined Instruction	42
6.2.3 Testing K2RED on Simulation	43

6.3 Modulo Multiplication Unit on Hardware	43
6.3.1 K2RED in RV32IM Instruction Set	44
6.3.2 Modmul Instruction Call	44
6.3.3 Testing Modmul Unit	45
6.4 Butterfly Unit	45
6.4.1 Planned Butterfly Module Design	46
6.4.2 The Requirement for the Modmul Instruction	47
6.4.3 The Test of Butterfly Instruction	48
7. ENERGY AND POWER ANALYSIS COMPARISON	49
8. SIDE-CHANNEL ANALYSIS	51
8.1 Simple Power Analysis	52
8.2 Simple Power Analysis Observation of "Heartbeat" on RISC-V	53
8.3 Assigning the Trigger Signal	54
8.4 Evaluation of First Results	54
8.5 UART Code for the Butterfly Module	55
9. REALISTIC CONSTRAINTS, CONCLUSIONS, AND RECOMMENDATIONS	59
9.1 Application Area of the Study	59
9.2 Realistic Design Constraints	59
9.2.1 Cost of the Project	60
9.2.2 Standards	61
9.2.3 Social, Environmental, and Economic Impact	61
9.2.4 Health and Safety Risks	61
9.3 Results	62
9.4 Suggestions for Future Work	62
REFERENCES	64
CURRICULUM VITAE	67

ABBREVIATIONS

DEC	: Decryption
DSP	: Digital Signal Processing
ECC	: Elliptic Curve Cryptography
ELF	: Executable and Linkable Format
ENC	: Encryption
FPGA	: Field Programmable Gate Array
GCC	: GNU Compiler Collection
GDB	: GNU Debugger
GNU	: GNU Not Unix
IND-CCA2	: Chosen-Ciphertext Attacks
INTT	: Inverse Number Theoretic Transform
ISA	: Instruction Set Architecture
ITU	: Istanbul Technical University
Kyber	: CRYSTALS-Kyber
LWE	: Learning With Errors
KEM	: Key Encapsulation Mechanism
KEYGEN	: Key Generation
MODMUL	: Modulo Multiplication Module
NIST	: National Institute of Standards and Technology
NTT	: Number Theoretic Transform
PQC	: Post-Quantum Cryptography
PRF	: Pseudorandom Function
RISC-V	: Reduced Instruction Set Computing - V
ROM	: Read Only Memory
RSA	: Rivest-Shamir-Adleman
SoC	: System-on-Chip
SVP	: Shortest Vector Problem
UART	: Universal Asynchronous Receiver/Transmitter
U.S.	: United States
XOF	: eXtensible Output Function

SYMBOLS

ω : Twiddle Vector

Q : Kyber-specific constant (kyber_q)

LIST OF TABLES

	<u>Page</u>
Table 6.1 : Comparison of Pure Software and Hardware/Software Implementation Execution Times	49
Table 6.2 : Hardware and Area Comparison Between the Basic Hornet Core and the Version with Integrated Butterfly Unit	50
Table 6.3 : Comparison of Results Between the Hornet Core with Butterfly Extension and the Original Hornet Core	50

LIST OF FIGURES

	<u>Page</u>
Figure 2.1: A 2-dimensional lattice and two different bases for it. [10]	6
Figure 2.2 : Learning with Errors [15]	8
Figure 2.3: Key Generation Function Referans Algorithm[16]	8
Figure 2.4: Encryption Function Reference Algorithm[16]	10
Figure 2.5: Decryption Function Reference Algorithm[16]	11
Figure 3.1 : Hornet RISC-V Core Pipelined Diagram	14
Figure 5.1: Makefile Corresponding File Configurations and Compiler Flags	23
Figure 5.2: Functional Structure of the Algorithm and Corresponding File Configuration Commands	24
Figure 5.3: Main function to test the algorithm to see if it works correctly	26
Figure 5.4: The Main Function to Test the ALgorithm on RISC-V	27
Figure 5.5: Addresses to Be Used for Executing the CRYSTALS-Kyber Algorithm on a RISC-V Processor	28
Figure 5.6: Timing Results of the CRYSTALS-Kyber Algorithm on a RISC-V Processor with the I Instruction set	29
Figure 5.7: Timing Results of the CRYSTALS-Kyber Algorithm on a RISC-V Processor with I and M Instruction sets	29
Figure 5.8: Makefile for Profiling	30
Figure 5.9: Results for Profiling of CRISTAL-Kyber	31
Figure 5.10: NTT function	32
Figure 5.11: INTT function	32
Figure 5.12: Polynomial Multiplication with NTT domain	32
Figure 5.13: Representation of Polynomial Multiplication with NTT and INTT Algorithm With 8 Bits [29]	33
Figure 5.14: Butterfly Representation	34
Figure 5.15: K2RED Algorithm in C Code	36
Figure 5.16 K2RED Algorithm [26]	37
Figure 6.1: Replacing Montgomery with K2RED in Software on the Hornet Processor Using the RV32I Instruction Set	40
Figure 6.2: Definition of MATCH and MASK Values	41
Figure 6.3: Declaration of the Function Using DECLARE_INSN	41
Figure 6.4: Definition of the K2RED Instruction in the riscv-opc File	42
Figure 6.5: Calling the Defined Instruction Using Inline Assembly	42
Figure 6.6: RTL Schematic of the K2RED Module	43
Figure 6.7: K2RED instruction test in RV32I set	43
Figure 6.8: K2RED Instruction Simulation in RV32IM	44
Figure 6.9: Inline Assembly Usage of Modmul	45
Figure 6.10: Modmul Instruction Simulation in RV32IM	45
Figure 6.11: NTT Algorithm Implementation in C	46

Figure 6.12: Schematic of Planned Design	47
Figure 6.13: INTT and Basemul Algorithms Implementation in C	48
Figure 6.14: Butterfly Instruction Simulation in RV32IM	48
Figure 8.1: Basic Power Analysis of Kyber	53
Figure 8.2: Observation of the Power Consumption Spike During Bit Transition	55
Figure 8.3: UART Code Designed for Butterfly Unit	56
Figure 8.4: MATLAB UART Code for Side Channel Analysis	57

DESIGN AND IMPLEMENTATION OF A RISC-V BASED OPTIMIZED CORE FOR HIGH-PERFORMANCE CRYPTOGRAPHIC ALGORITHMS

SUMMARY

Quantum computing technology possesses significantly greater processing power compared to classical computers. It can solve complex functions that are challenging for today's computers in a much shorter time. This capability threatens the security of existing cryptographic algorithms and necessitates the development of new cryptographic methods. As a result, a global transition from current algorithms to post-quantum cryptographic (PQC) algorithms has begun. This paper aims to provide hardware support for the CRYSTALS-Kyber algorithm, one of the finalists in the "Quantum-Resistant Cryptographic Algorithms" standardization competition organized by NIST (National Institute of Standards and Technology), using the RISC-V-based Hornet processor.

Shor's algorithm, developed by Peter Shor in 1994, is a quantum algorithm designed to run on quantum computers. It can factor large integers exponentially faster than classical algorithms, running in polynomial time. The security of asymmetric cryptographic systems relies on the difficulty of integer factorization for classical computers; therefore, Shor's algorithm poses a significant threat to these systems. The algorithm involves a classical part where a random number is chosen and certain arithmetic checks are performed, followed by a quantum part that uses the quantum Fourier transform to find the modular period, enabling factorization. While classical algorithms exhibit exponential time complexity, Shor's algorithm completes the task in polynomial time. This potential has accelerated research into post-quantum cryptography (PQC), anticipating that current asymmetric encryption methods will become insecure as quantum computers advance.

Among the finalists, the CRYSTALS-Kyber algorithm stands out as a leading post-quantum cryptographic algorithm expected to see widespread adoption as quantum computers become more prevalent. To provide hardware acceleration for this algorithm, the open-source and flexible RISC-V architecture offers an ideal platform for research and implementation of innovative solutions. This study utilizes the Hornet architecture, designed and developed by students at Istanbul Technical University. Hornet is a 32-bit RISC-V core with a compact design, making it suitable for hardware integration and further enhancements.

The work was conducted in two main stages. In the first stage, software-level optimizations were applied to improve the efficiency of the CRYSTALS-Kyber algorithm on hardware. The code structure was reorganized, unnecessary computations were eliminated, and the algorithm was aligned more closely with the hardware architecture. Profiling techniques were then used on the optimized code to identify the parts consuming the most time and energy. This analysis highlighted the system resources most heavily used, guiding the selection of hardware acceleration targets.

In the RISC-V architecture, the "execute" stage, where basic operations are performed, provides a flexible point for integrating custom hardware instructions. Accordingly, profiling results indicated that adding custom instructions at the execute stage was appropriate. In particular, the functions `montgomery_reduce`, `fqmul`, and `ntt` were found to have high energy and time consumption. Academic studies and experimental measurements confirmed that `montgomery_reduce` operates inefficiently in hardware. Therefore, it was replaced with the more efficient K2RED algorithm, which performs modular reduction with less hardware resource usage and lower energy consumption. This substitution significantly reduced both processing time and energy usage, resulting in a 52.25% performance improvement on the Hornet processor.

In the second stage, K2RED, the smallest among the critical functions performing modular reduction, was added to the hardware as a custom instruction, and the Kyber algorithm was re-executed. Compared to the simulation using the software-only K2RED, a 5% performance improvement was observed. Subsequently, `Modmul` and `Butterfly` operations were also integrated into hardware. These combined hardware enhancements led to a total speedup of 21.85% compared to software-only optimizations.

A key highlight of this research is that the acceleration was achieved by adding new instructions directly to the hardware. The area overhead of these added instructions is very efficient relative to the energy and time savings they provide.

In summary, this paper presents an effective method to enhance the performance of the CRYSTALS-Kyber post-quantum encryption algorithm on the RISC-V based Hornet core. Through detailed code profiling and the integration of custom hardware instructions, significant improvements were achieved in both execution time and energy consumption. Simulations demonstrate that the proposed hardware-accelerated optimizations considerably increase the efficiency of the Kyber algorithm. This study not only validates the feasibility of implementing PQC algorithms on RISC-V systems but also establishes a solid foundation for future research in the field.

YÜKSEK PERFORMANSLI KRİPTOGRAFİK ALGORİTMALAR İÇİN OPTİMİZE BİR RISC-V TABANLI BİR İŞLEMCİ TASARIMI VE GERÇEKLENMESİ

ÖZET

Kuantum bilgisayar teknolojisi, klasik bilgisayarlara kıyasla çok daha yüksek işlem gücüne sahiptir. Günümüzde kullanılan bilgisayarların çözmekte zorlandığı karmaşık fonksiyonlar, kuantum bilgisayarlar tarafından kısa sürede çözülebilmektedir. Bu durum, mevcut kriptografik algoritmaların güvenliğini tehlikeye atmakta ve kriptografi alanında yeni arayışları zorunlu kılmaktadır. Bu nedenle dünya genelinde, mevcut algoritmalarla kuantum sonrası kriptografik algoritmalar (Post-Quantum Cryptography - PQC) geçiş süreci başlamıştır. Bu makale, NIST (National Institute of Standards and Technology) tarafından düzenlenen “Quantum-Resistant Cryptographic Algorithms” standardizasyon yarışmasının finalistlerinden biri olan CRYSTALS-Kyber algoritmasını, RISC-V tabanlı Hornet işlemcisi üzerinde donanımsal olarak desteklemeyi amaçlamaktadır.

Shor’un algoritması, 1994 yılında Peter Shor tarafından geliştirilen ve kuantum bilgisayarlar üzerinde çalışmak üzere tasarlanmış bir algoritmadır. Bu algoritma, büyük sayıların asal çarpanlara ayrılmasını klasik algoritmalarla kıyasla çok daha hızlı, polinomsal zamanda gerçekleştirebilir. Asimetrik kriptografi sistemlerinin güvenliği bu ayrıştırma işleminin klasik bilgisayarlar için zor olmasına dayanır; dolayısıyla Shor’un algoritması bu sistemler için ciddi bir tehdit oluşturur. Algoritma, klasik bir bölümde rastgele bir sayı seçip bazı aritmetik kontroller yaptıktan sonra, kuantum bölümde kuantum Fourier dönüşümü yardımıyla modüler dönem bulma işlemini gerçekleştirir. Bu sayede, asal çarpanlara ulaşmak mümkün hale gelir. Klasik algoritmalar üstel zaman karmaşıklığına sahipken, Shor’un algoritması bu işlemi polinomsal sürede tamamlayabilir. Bu potansiyel nedeniyle, kuantum bilgisayarların yeterli ölçeğe ulaşması durumunda mevcut asimetrik şifreleme yöntemlerinin güvensiz hale geleceği öngörülmekte ve bu da post-kuantum kriptografi (PQC) çalışmalarını hızlandırmaktadır.

Finalistlerden biri olan CRYSTALS-Kyber algoritması, kuantum bilgisayarların yaygınlaşmasıyla birlikte yakın gelecekte kullanılacak kuantum sonrası kriptografik algoritmalar arasında öne çıkmaktadır. Bu algoritmayı donanımsal olarak desteklemek amacıyla, RISC-V mimarisi açık kaynaklı ve esnek yapısıyla araştırma yapmak ve yenilikçi çözümleri uygulamak için ideal bir platform sunmaktadır. Bu çalışmada, İstanbul Teknik Üniversitesi öğrencileri tarafından tasarlanmış ve geliştirme süreci devam eden Hornet mimarisi kullanılmıştır. Hornet, 32 bitlik bir RISC-V çekirdeğidir ve kompakt yapısı sayesinde donanımsal entegrasyon ve iyileştirmelere elverişlidir.

Çalışma iki ana aşamada yürütülmüştür. İlk aşamada, CRYSTALS-Kyber algoritmasının donanımda daha verimli çalışabilmesi için yazılım düzeyinde optimizasyonlar yapılmıştır. Bu optimizasyon sürecinde kodun yapısı yeniden düzenlenmiş, gereksiz hesaplamalar elenmiş ve algoritmanın donanım mimarisiyle daha uyumlu hale getirilmesi sağlanmıştır. Optimize edilmiş kod üzerinde profillemeye (profiling) yöntemi uygulanarak işlemcide en fazla zaman ve enerji tüketen bölümler tespit edilmiştir. Bu analiz, sistem kaynaklarının en çok hangi fonksiyonlar tarafından kullanıldığını ortaya koyarak, donanım hızlandırmalarının hangi noktalarda yapılması gerektiği konusunda yol gösterici olmuştur.

RISC-V mimarisinde temel işlemlerin gerçekleştirildiği "execute" (yürütme) aşaması, özelleştirilmiş donanım bloklarının entegre edilebileceği esnek bir yapıya sahiptir. Bu nedenle, yapılan profillemeye sonuçlarına göre yürütme aşamasına özel buyruk(instruction) eklenmesi uygun bulunmuştur. Profilleme analizinde özellikle montgomery_reduce, fmul ve ntt fonksiyonlarının sistem genelinde yüksek düzeyde enerji ve zaman tüketimine neden olduğu gözlemlenmiştir.

Özellikle montgomery_reduce fonksiyonunun donanımda verimsiz çalıştığı, yapılan akademik araştırmalar ve deneysel ölçümlerle doğrulanmıştır. Bu noktada, aynı işlevi daha verimli biçimde yerine getiren K2RED algoritması ile montgomery_reduce'un değiştirilmesine karar verilmiştir. K2RED, modüler indirgeme işlemlerini daha az donanım kaynağı kullanarak ve daha düşük enerji tüketimiyle gerçekleştirebilen bir yaklaşımdır. Bu değişiklik, hem işlem süresini hem de enerji tüketimini önemli ölçüde azaltmıştır. Yapılan tüm optimizasyonlar sonucunda, Hornet işlemcisi üzerinde %52,25 oranında performans artışı elde edilmiştir.

İkinci aşamada, daha önce belirtilen fonksiyonlar arasında en küçüğü olan ve mod işlemini gerçekleştiren K2RED fonksiyonu donanıma özel bir komut olarak eklenmiş ve Kyber algoritması tekrar çalıştırılmıştır. Yazılımda K2RED fonksiyonunun kullanıldığı simülasyon ile karşılaştırıldığında %5 oranında performans artışı gözlemlenmiştir. Ardından, Modmul ve Butterfly işlemleri de donanıma entegre edilmiştir. Tüm bu donanım destekli eklemeler sonucunda, yazılımda yapılan iyileştirmelere kıyasla toplamda %21,85 oranında bir hızlandırma sağlanmıştır.

Bu araştırmanın öne çıkan özelliği, yapılan hızlandırmaların donanıma yeni buyruklar eklenerek gerçekleştirilmiş olmasıdır. Eklenen buyrukların kapladığı alan, sağlanan enerji ve zaman tasarrufuna kıyasla oldukça verimlidir.

Bu makalede, RISC-V tabanlı Hornet çekirdeği üzerinde CRYSTALS-Kyber kuantum sonrası şifreleme algoritmasının performansını artırmak amacıyla etkili bir yöntem geliştirilmiştir. Yapılan detaylı kod profillemeye çalışmaları ve donanıma özel olarak tasarlanan yeni komutların entegrasyonu sayesinde, algoritmanın hem işlem süresinde hem de enerji tüketiminde kayda değer iyileştirmeler elde edilmiştir. Gerçekleştirilen simülasyonlar, önerilen donanım destekli optimizasyonların Kyber algoritmasının verimliliğini önemli ölçüde artırdığını ortaya koymaktadır. Bu çalışma, kuantum sonrası şifreleme algoritmalarının RISC-V sistemlerinde uygulanabilirliğini kanıtlamanın yanı sıra, alandaki gelecek araştırmalar için de sağlam bir temel oluşturmuştur.

1. INTRODUCTION

Post-quantum cryptographic algorithms have been developed to replace classical encryption systems that will be threatened by the emergence of quantum computers in the future. These algorithms are designed to resist problems that quantum computers are particularly efficient at solving. In this context, CRYSTALS-Kyber has emerged as a strong key exchange algorithm and has been selected for standardization by NIST due to its high level of security against both classical and quantum attacks. However, these algorithms have primarily been designed in software, and their hardware implementations have not been fully realized. As a result, CRYSTALS-Kyber currently lacks sufficient hardware support.

The advancement of quantum computers poses a serious threat to the security of traditional cryptographic systems. This has necessitated the development of new systems that are resilient to quantum attacks and can potentially replace classical algorithms. Post-quantum cryptography (PQC) has therefore emerged as a field aimed at addressing future security needs, with algorithms like CRYSTALS-Kyber gaining prominence. While Kyber offers strong resistance against both classical and quantum threats, it has primarily been developed in software, and significant gaps remain in terms of its hardware implementation. Addressing these gaps through hardware-supported solutions offers the potential to enhance both the efficiency and security of the algorithm.

In this project, the CRYSTALS-Kyber algorithm has been hardware-optimized on the RISC-V architecture and implemented using the modular and open-source Hornet core developed by ITU. The project pursues two main goals: First, to implement the Kyber algorithm at the hardware level and accelerate it through various hardware optimizations; and second, to identify the most time- and energy-consuming steps in the overall software and support these steps with custom hardware extensions to improve performance.

Accordingly, the Kyber algorithm was first examined in detail, and the functions used in the encryption and decryption stages were analyzed thoroughly. A comprehensive literature review was conducted to identify the most time- and power-intensive operations within these stages, and alternative algorithms capable of accelerating these operations were explored. The selected alternatives were integrated into the system both at the software and hardware levels, and simulations were carried out to test both the original and modified versions. The performance of the software and hardware was then compared in terms of execution time, and the observed improvements were analyzed.

Additionally, new custom instructions were added to the RISC-V ISA to accelerate specific steps of the algorithm at the hardware level. Based on the findings during the project, a roadmap for future improvements was created, laying the groundwork for further enhancements. Along with performance improvements, innovative steps were taken in terms of security, and new security checks were introduced into the project. In this context, the enhanced core was tested against side-channel attacks, and techniques such as power analysis and electromagnetic analysis were used to identify data leakage points.

In conclusion, this project enabled the optimization of the CRYSTALS-Kyber algorithm at both the hardware and software levels, contributing to the development of secure and high-performance cryptographic solutions for the quantum era. The efforts in post-quantum cryptography demonstrated here represent an important step toward creating more robust, faster, and energy-efficient security infrastructures for the future.

2. MATHEMATICAL BACKGROUND

2.1 Post Quantum Cryptography

The emergence of quantum computing poses a significant threat to traditional cryptographic algorithms, which rely on the computational complexity of problems such as modular arithmetic and discrete logarithms. In 1995, Peter Shor introduced Shor's algorithm[1], which demonstrated the potential to efficiently solve these problems using quantum computers. At the time, the absence of sufficiently advanced quantum hardware rendered classical algorithms like Rivest-Shamir-Adleman (RSA)[2] and Elliptic Curve Cryptography (ECC)[3] secure. However, the development of quantum computers has since raised concerns about their ability to break these algorithms. Although current quantum computers lack the processing power and memory required to compromise RSA and similar systems, they represent a future risk that necessitates proactive measures.

To address this challenge, Post-Quantum Cryptography (PQC)[4] has emerged as a critical field focused on developing cryptographic algorithms resistant to quantum computer attacks. Unlike traditional algorithms, which are vulnerable to quantum-based attacks due to their reliance on mathematical problems that quantum computers can solve efficiently, PQC algorithms are designed to withstand both classical and quantum attacks. These algorithms leverage mathematical problems believed to be resistant to quantum computation, including those from lattice-based cryptography[5], code-based cryptography, multivariate polynomial cryptography, and hash-based cryptography [6]. The CRYSTALS-Kyber algorithm[7], a lattice-based PQC algorithm, is a prominent example of such efforts.

In response to the growing need for quantum-resistant cryptography, the National Institute of Standards and Technology (NIST)[8] initiated a competition in 2012 to standardize PQC algorithms. This process, formalized in 2016, aims to identify and evaluate novel cryptographic algorithms capable of securing digital communications

in the presence of large-scale quantum computers. The NIST PQC standardization process involves multiple rounds of submissions, reviews, and public scrutiny to ensure robust and secure algorithms. By 2022, the competition had entered its final stage, and algorithms like CRYSTALS-Kyber were selected as finalists moving toward becoming standards. The main goal is to develop post-quantum algorithms that can eventually replace current cryptographic methods and keep digital communication secure in the future.

2.2 Quantum Computers and Quantum Computing

In the past decade, significant progress has been made in the experimental development of quantum computers. These machines use the full complexity of many-particle quantum wavefunctions to perform computations that classical computers cannot efficiently solve.

Unlike classical computers, quantum computers are not simply faster or smaller; they are fundamentally different. They rely on coherent control of quantum states such as superposition and entanglement, enabling entirely new types of computation.[9]

One of the main motivations behind building quantum computers is Shor's algorithm, which can factor large numbers exponentially faster than the best-known classical algorithms. This poses a serious threat to current encryption systems like RSA, which rely on the difficulty of factoring. Shor's work demonstrated a clear example where a quantum computer can outperform any classical machine on a real-world problem.

Beyond cryptography, quantum computing holds great promise in fields such as quantum chemistry, materials science, and optimization. For example, quantum algorithms could allow scientists to simulate complex molecular interactions with high precision, which would accelerate drug discovery and the development of new materials. These are tasks that are currently intractable even for the most powerful supercomputers.

Moreover, as quantum hardware continues to mature, we may see the rise of hybrid quantum-classical systems, where quantum processors are used alongside classical

ones to tackle specialized problems. The development of quantum technologies also extends beyond computing, enabling secure quantum communication and ultra-precise sensing and measurement techniques through quantum metrology.

Although there are still many technical challenges to overcome, such as error correction, qubit scalability, and system stability, the pace of progress suggests that quantum computing is steadily moving from theory to practical reality. As research continues, quantum technologies are likely to redefine the boundaries of what is computationally possible, offering transformative capabilities across science, engineering, and industry.

2.3 Standardization of Post-Quantum Algorithms

The National Institute of Standards and Technology (NIST) is a U.S. federal agency responsible for developing measurement standards and promoting technological innovation. In the field of cybersecurity, NIST plays a key role by establishing cryptographic standards used worldwide.

Recognizing the potential threat posed by quantum computers, NIST launched a Post-Quantum Cryptography Standardization project. The aim is to identify and standardize public-key cryptographic algorithms that remain secure even against quantum-capable adversaries.

Quantum computers, if built on a large scale, would be able to solve certain mathematical problems exponentially faster than classical computers. This includes breaking widely used public-key cryptosystems such as RSA and ECC, which are foundational to secure digital communication today. Such a development would put the confidentiality, authenticity, and integrity of data at serious risk.

Post-quantum cryptography, also referred to as quantum-resistant cryptography, seeks to develop new algorithms that can withstand both quantum and classical attacks. These algorithms must also be practical, capable of integrating into existing systems and communication protocols without requiring major infrastructure changes.

Given that cryptographic infrastructure takes many years to design, test, and deploy globally, the transition to quantum-safe cryptography must begin now to ensure long-term data security.

2.4 Lattice-Based Cryptography

A lattice is a collection of discrete points arranged in a periodic structure within an n -dimensional Euclidean space. Any point in a lattice can be represented as an integer linear combination of a set of basis vectors.

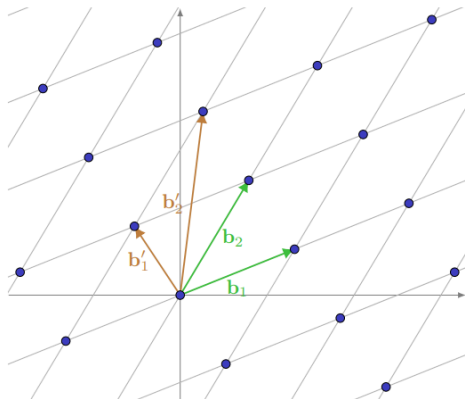


Figure 2.1: A 2-dimensional lattice and two different bases for it. [10]

For example, Figure 2.1 shows 2 different bases that can be chosen in 2-dimension. Some of them are considered a good basis and some of them are bad. The geometric relationship between these basis vectors, particularly the angles between them, plays a crucial role in determining whether the basis is considered good or bad

A good basis consists of vectors that are close to orthogonal, meaning that each vector clearly spans a distinct dimension of the space. This leads to more efficient computations and representations. In contrast, bad bases are formed by vectors that are nearly linearly dependent. For example, the angle between them is very small, making them almost parallel. Such configurations lead to highly redundant representations and computational difficulties.[5]

The distinction between good and bad bases becomes especially important in the context of hard computational problems on lattices, such as the Shortest Vector Problem (SVP)[11]. When using a good basis, identifying the shortest vector in the

lattice is relatively straightforward. However, with a bad basis, the same task becomes significantly more complex, as the redundancy in vector directions makes it difficult to distinguish short lattice vectors.

While solving SVP in two dimensions is relatively simple, the problem becomes exponentially harder as the dimension increases. In high-dimensional lattices, such as those in 300 dimensions, solving SVP is computationally infeasible even for modern classical or quantum computers, with no known efficient algorithms for exact solutions in general cases.

2.5 CRYSTALS-Kyber

CRYSTALS-Kyber is a post-quantum, asymmetric cryptographic algorithm designed to be secure against both classical and quantum adversaries. It functions as a Key Encapsulation Mechanism (KEM)[12] and has been selected by the NIST as part of its Post-Quantum Cryptography standardization efforts.

In asymmetric cryptography, a sender encrypts a message using the recipient's public key. Only the recipient, possessing the corresponding private key, can decrypt the message. Kyber adheres to this paradigm, offering security against chosen-ciphertext attacks (IND-CCA2)[13] by leveraging the computational hardness of the Module Learning With Errors (LWE)[14] problem. The LWE problem involves solving systems of linear equations that have been perturbed by small random errors. Mathematically, given a matrix A and a secret vector s , one computes $B = A \cdot s + e$, where e is an error vector with small, randomly chosen entries. While determining s from A and b is straightforward when $e = 0$, the presence of the error term e renders the problem computationally intractable. This difficulty persists even for quantum computers, making LWE a robust foundation for cryptographic schemes. Kyber employs a structured variant known as Module-LWE, operating over polynomial rings to enhance efficiency. Specifically, computations are performed in the ring $R_q = \mathbb{Z}_q[x]/(x^{n+1})$, where n is a power of two and q is a prime modulus. This structure allows for efficient polynomial arithmetic, which is advantageous for both software and hardware implementations.

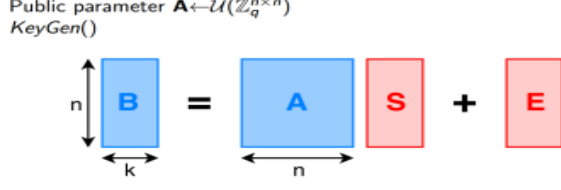


Figure 2.2: Learning with Errors [15]

2.5.1 Key Generation

In asymmetric cryptographic algorithms, two parties wishing to exchange data securely require a key pair consisting of a public key and a private key. The private key must remain confidential, while the public key can be freely distributed. Encryption is performed using the public key, and only the corresponding private key can decrypt the resulting ciphertext. In schemes based on the Learning With Errors (LWE) problem, the public key is mathematically derived from the private key by introducing carefully structured noise. Due to the computational hardness of the LWE problem, recovering the private key from the public key is considered infeasible, even with substantial computational resources.

Algorithm 4 KYBER.CPAPKE.KeyGen(): key generation

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

```

1:  $d \leftarrow \mathcal{B}^{32}$ 
2:  $(\rho, \sigma) := G(d)$ 
3:  $N := 0$ 
4: for  $i$  from 0 to  $k - 1$  do                                ▷ Generate matrix  $\hat{\mathbf{A}} \in R_q^{k \times k}$  in NTT domain
5:   for  $j$  from 0 to  $k - 1$  do
6:      $\hat{\mathbf{A}}[i][j] := \text{Parse}(\text{XOF}(\rho, j, i))$ 
7:   end for
8: end for
9: for  $i$  from 0 to  $k - 1$  do                                ▷ Sample  $\mathbf{s} \in R_q^k$  from  $B_{\eta_1}$ 
10:   $\mathbf{s}[i] := \text{CBD}_{\eta_1}(\text{PRF}(\sigma, N))$ 
11:   $N := N + 1$ 
12: end for
13: for  $i$  from 0 to  $k - 1$  do                                ▷ Sample  $\mathbf{e} \in R_q^k$  from  $B_{\eta_1}$ 
14:   $\mathbf{e}[i] := \text{CBD}_{\eta_1}(\text{PRF}(\sigma, N))$ 
15:   $N := N + 1$ 
16: end for
17:  $\hat{\mathbf{s}} := \text{NTT}(\mathbf{s})$ 
18:  $\hat{\mathbf{e}} := \text{NTT}(\mathbf{e})$ 
19:  $\hat{\mathbf{t}} := \hat{\mathbf{A}} \circ \hat{\mathbf{s}} + \hat{\mathbf{e}}$ 
20:  $pk := (\text{Encode}_{12}(\hat{\mathbf{t}} \bmod^+ q) \parallel \rho)$                                 ▷  $pk := \mathbf{A}\mathbf{s} + \mathbf{e}$ 
21:  $sk := \text{Encode}_{12}(\hat{\mathbf{s}} \bmod^+ q)$                                 ▷  $sk := \mathbf{s}$ 
22: return  $(pk, sk)$ 

```

Figure 2.3: Key Generation Function Referans Algorithm [16]

The algorithm of the key generation function is illustrated in Figure 2.5.2. The matrix A is generated using random values through an eXtendable Output Function (XOF), which is a type of cryptographic hash function. The secret key and error matrices are also generated using another cryptographic hash function, the Pseudorandom Function (PRF). These matrices are represented as polynomials, and they undergo polynomial multiplication, which is a fundamental operation in Kyber. To perform this multiplication more efficiently, the polynomials in the integer domain are transformed into the NTT[17] (Number Theoretic Transform) domain using the NTT function. In this domain, the complexity of multiplication is significantly reduced. After multiplying and adding the error term in the NTT domain, the result is transformed back to the integer domain, yielding the public and secret keys.

2.5.2 Encryption

The message intended for transmission is encrypted by the sender using the recipient's public key. In this process, the public key is used to generate matrices A and t , and a matrix r is generated using random tokens. Together with error vectors e_1 and e_2 , the term $r \cdot [A \parallel t]$ is computed. The message is then added to this result while all values remain in the NTT domain, allowing for efficient polynomial operations due to reduced computational complexity. The matrix A , r , e_1 , e_2 is calculated with secure hash functions.

The resulting ciphertext consists of two components, which are converted back to the integer domain using the inverse NTT (INTT) function. These two components form the matrices u and v , which are then transmitted to the recipient.

Due to the computational hardness of lattice-based problems, particularly those based on the Learning With Errors (LWE) assumption, it is computationally infeasible to extract any useful information from u and v without knowledge of the corresponding secret key. Thus, only the intended recipient, who holds the private key, can decrypt the ciphertext and recover the original message.

Algorithm 5 KYBER.CPAPKE.Enc(pk, m, r): encryption

Input: Public key $pk \in \mathcal{B}^{12 \cdot k \cdot n/8 + 32}$

Input: Message $m \in \mathcal{B}^{32}$

Input: Random coins $r \in \mathcal{B}^{32}$

Output: Ciphertext $c \in \mathcal{B}^{d_u \cdot k \cdot n/8 + d_v \cdot n/8}$

```

1:  $N := 0$ 
2:  $\hat{\mathbf{t}} := \text{Decode}_{12}(pk)$ 
3:  $\rho := pk + 12 \cdot k \cdot n/8$ 
4: for  $i$  from 0 to  $k - 1$  do                                 $\triangleright$  Generate matrix  $\hat{\mathbf{A}} \in R_q^{k \times k}$  in NTT domain
5:   for  $j$  from 0 to  $k - 1$  do
6:      $\hat{\mathbf{A}}^T[i][j] := \text{Parse}(\text{XOF}(\rho, i, j))$ 
7:   end for
8: end for
9: for  $i$  from 0 to  $k - 1$  do                                 $\triangleright$  Sample  $\mathbf{r} \in R_q^k$  from  $B_{\eta_1}$ 
10:   $\mathbf{r}[i] := \text{CBD}_{\eta_1}(\text{PRF}(r, N))$ 
11:   $N := N + 1$ 
12: end for
13: for  $i$  from 0 to  $k - 1$  do                                 $\triangleright$  Sample  $\mathbf{e}_1 \in R_q^k$  from  $B_{\eta_2}$ 
14:   $\mathbf{e}_1[i] := \text{CBD}_{\eta_2}(\text{PRF}(r, N))$ 
15:   $N := N + 1$ 
16: end for
17:  $e_2 := \text{CBD}_{\eta_2}(\text{PRF}(r, N))$                                  $\triangleright$  Sample  $e_2 \in R_q$  from  $B_{\eta_2}$ 
18:  $\hat{\mathbf{r}} := \text{NTT}(\mathbf{r})$ 
19:  $\mathbf{u} := \text{NTT}^{-1}(\hat{\mathbf{A}}^T \circ \hat{\mathbf{r}}) + \mathbf{e}_1$                                  $\triangleright \mathbf{u} := \mathbf{A}^T \mathbf{r} + \mathbf{e}_1$ 
20:  $v := \text{NTT}^{-1}(\hat{\mathbf{t}}^T \circ \hat{\mathbf{r}}) + e_2 + \text{Decompress}_q(\text{Decode}_1(m), 1)$      $\triangleright v := \mathbf{t}^T \mathbf{r} + e_2 + \text{Decompress}_q(m, 1)$ 
21:  $c_1 := \text{Encode}_{d_u}(\text{Compress}_q(\mathbf{u}, d_u))$ 
22:  $c_2 := \text{Encode}_{d_v}(\text{Compress}_q(v, d_v))$ 
23: return  $c = (c_1 \| c_2)$                                  $\triangleright c := (\text{Compress}_q(\mathbf{u}, d_u), \text{Compress}_q(v, d_v))$ 

```

Figure 2.4: Encryption Function Reference Algorithm[16]

2.5.3 Decryption

When the encrypted message reaches the recipient, it must be decrypted using the decryption function. The recipient uses the received matrices \mathbf{u} and \mathbf{v} , along with their own secret key, to recover the original message. For this purpose, the matrix \mathbf{u} is first multiplied by the secret key in the NTT domain. The result is then subtracted from the matrix \mathbf{v} , yielding a value that contains both the original message and the associated error terms.

Due to the properties of the error distribution, these error terms can be effectively eliminated using modular reduction techniques, allowing the correct message to be recovered.

Algorithm 6 KYBER.CPAPKE.Dec(sk, c): decryption

Input: Secret key $sk \in \mathcal{B}^{12 \cdot k \cdot n/8}$

Input: Ciphertext $c \in \mathcal{B}^{d_u \cdot k \cdot n/8 + d_v \cdot n/8}$

Output: Message $m \in \mathcal{B}^{32}$

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

Figure 2.5: Decryption Function Reference Algorithm[16]

3. MATHEMATICAL BACKGROUND

3.1 Processor Determination for the Project

RISC-V ISA[18], with its open-source nature and easily comprehensible architecture, provides an ideal environment for research and development. RISC-V is defined as an open-source reduced instruction set computing (RISC) architecture. In areas such as cryptography, which demand high computational intensity, RISC-V's open and flexible design offers unique advantages over proprietary architectures. Throughout this project, the improvements were implemented on the RISC-V architecture.

For these reasons, the decision was made to adopt the Hornet processor, originally developed at Istanbul Technical University (ITU). Hornet is a RISC-V-based processor characterized by its straightforward setup, modular architecture, and ease of modification. These features, combined with its adaptability to custom changes, were pivotal in our selection process.

3.1.1 Hornet

Hornet[19] is an open-source 32-bit RISC-V core developed by students at Istanbul Technical University (ITU). We selected RISC-V for this project due to its open-source nature, flexibility, and scalability. Hornet, which is available on GitHub, features a well-documented architecture supporting the RV32IM instruction sets (Figure 3.1). The core also includes support for peripherals, software, and system-on-chip (SoC) examples.

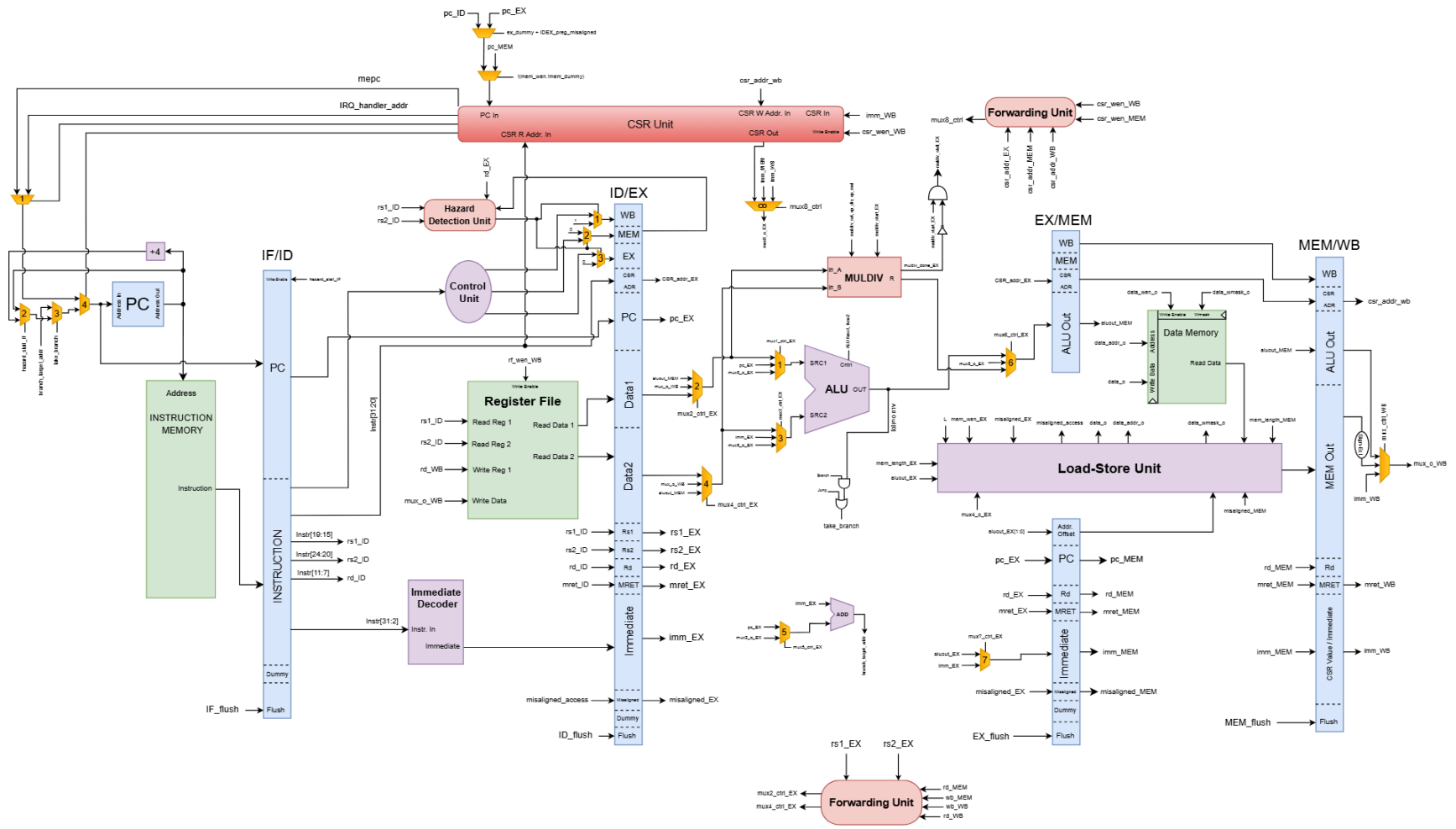


Figure 3.1: Hornet RISC-V Core Pipelined Diagram

The RV32I [20] base instruction set provides essential 32-bit RISC-V operations, including arithmetic (add/subtract), memory access (load/store), branching, and bit manipulation. The RV32IM extension augments this by adding hardware-accelerated multiplication and division, eliminating the need for software emulation and significantly boosting performance in compute-intensive applications like embedded systems or digital signal processing.

Hornet employs a pipelined architecture, fetching and executing one instruction per clock cycle. Additionally, it integrates an internal debugger that monitors a designated address value (0 or 1), allowing developers to verify whether algorithms running on the processor have executed successfully or terminated with errors.

3.2 RISC-V GNU Toolchain

To generate executable instruction memory files for the processor, the algorithms must be compiled into hexadecimal format using the RISC-V GNU Toolchain[21], a specialized compiler suite designed for RISC-V architectures. The RISC-V GNU Toolchain is an open-source collection of programming tools (including GCC, binutils, and GDB) that supports cross-compilation for RISC-V targets. Notably, it includes assemblers, linkers, and debuggers tailored for RISC-V's instruction sets (RV32I, RV64IM).

Since the RISC-V GNU Toolchain is natively compatible only with Linux systems, we selected the Ubuntu distribution[22] for this project. All algorithms were compiled through the Ubuntu terminal environment, leveraging the toolchain's Linux-optimized build infrastructure. This setup ensured seamless compatibility with Hornet's RV32IM instruction set and debugging requirements.

3.2.1 RISC-V GNU-Toolchain Installation

The installation and configuration of software applications on Linux systems are performed through terminal commands. These specialized Linux commands handle essential operations such as downloading, installing, and executing programs. The RISC-V GNU Toolchain, being open-source, is distributed via GitHub. To install it,

users must first navigate to the desired directory in the Ubuntu terminal using “cd” command, then execute the following installation command:

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

Prior to software installation, certain prerequisites must be satisfied to ensure proper configuration of the Ubuntu environment and successful toolchain setup. Several standard packages need to be installed as dependencies. The following command is required to install these essential packages:

- `sudo apt-get install autoconf automake autotools-dev curl python3 python3-pip python3-tomli libmpc-dev libmpfr-dev libgmp-dev gawk build-essential bison flex texinfo gperf libtool patchutils bc zlib1g-dev libexpat-dev ninja-build git cmake libglib2.0-dev libslirp-dev`

To build the Linux cross-compiler, go to the specific folder with the “cd” command:

- `cd riscv-gnu-toolchain`

and do the configurations with the following command:

- `./configure --prefix=/opt/riscv --with-arch=risc32im --with-abi=ilp32-`
- `sudo make`

This RISC-V configuration supports the use of both I and M instruction sets on 32-bit processors. The installation process may require over two hours to complete, during which maintaining a stable internet connection and ensuring adequate battery power is strongly recommended.

Upon successful installation, the toolchain's installation directory must be added to the system PATH variable. This configuration can be achieved by modifying the appropriate environment file. The following command should be executed to access and edit this configuration file:

- `sudo nano /etc/environment`

This will open a text file in the Terminal. Now, simply add the following text to the end of the file:

- `:/opt/riscv/bin`

Save the file using CTRL+O followed by CTRL+X to exit. At this stage, the RISC-V GNU Toolchain becomes fully operational, allowing for both modifications to the toolchain itself and successful compilation of algorithms for the RISC-V architecture.

3.3 Downloading CRYSTALS-Kyber

Throughout this project, we utilized the finalist version of CRYSTALS-Kyber from NIST's official website[23], which represents the most optimized implementation. While a GitHub version exists, it contains experimental functions and demonstrates suboptimal performance. The NIST-certified version not only delivers superior efficiency but also features a more comprehensible code structure that facilitates necessary modifications.

4. LITERATURE REVIEW

In a previous study conducted at ITU whose name is Implementation of CRYSTALS-Kyber Post Quantum Algorithm On RISC-V Processor[24], the CRYSTALS-Kyber algorithm was implemented on an FPGA-based RISC-V processor, where the KeccakF1600_StatePermute function was accelerated using a dedicated hardware unit. In this design, the Keccak hardware was directly connected to memory and the Ibex core, without any software-level optimizations. In the literature, similar works, such as those by Mesera et al.[25], have integrated custom hardware for KeccakF1600, Montgomery Reduce into RISC-V-based PULPino or PULPissimo microcontrollers through a bus interface and instruction set extensions. These efforts achieved acceleration factors ranging from 1.84 to 9.6. In contrast to these studies, our work focuses on accelerating the NTT and Butterfly operation, which is one of the most time-consuming parts of the Kyber algorithm, rather than the Keccak function.

Another relevant study titled "Hardware Design of K2RED Modular Multiplication Algorithm Used in NTT for Post Quantum Cryptography and Homomorphic Encryption"[26] investigated various modular multiplication and reduction algorithms in the context of NTT acceleration for PQC and homomorphic encryption schemes. The study demonstrated that the K2RED algorithm outperforms the traditional Montgomery algorithm in terms of clock frequency, resource utilization, and overall efficiency when implemented on an FPGA using DSP blocks. By evaluating designs with different butterfly unit configurations, it was shown that K2RED achieves better performance, especially in area and speed trade-offs. Motivated by these findings, our design also utilizes the K2RED modular reduction algorithm instead of Montgomery reduction to accelerate the NTT operation within CRYSTALS-Kyber, aiming for higher efficiency and reduced resource usage.

In her Ph.D. thesis "Hardware Design of Elliptic Curve Cryptosystems and Side-Channel Attacks"[27], S. B. Ors provides a comprehensive analysis of how side-channel attacks can be performed on cryptographic hardware and discusses effective countermeasures. This work served as a foundational reference in understanding the methodology of side-channel analysis. In our study, we benefited from this thesis to better understand how to conduct side-channel evaluations and to take necessary precautions during the hardware implementation of post-quantum cryptographic algorithms.

In the study titled "Instruction Set Extension of RISC-V Core with Plantard Modular Reduction for NTT Used in Post-Quantum Cryptography"[28], A. Üstün demonstrates how to integrate custom modular reduction operations into a RISC-V core using instruction set extensions. The work provides valuable insights into modifying the RISC-V toolchain and implementing hardware-software co-design techniques for accelerating NTT operations. This study was an important reference for understanding how to design and integrate custom instructions into a RISC-V-based architecture for post-quantum cryptographic applications.

4.1 Originality of Our Project

This study offers several key contributions that distinguish it from previous work and add value to the field of hardware-based post-quantum cryptography. Unlike earlier approaches that mainly focused on accelerating hashing functions like Keccak, our work targets the NTT and its Butterfly operations, which are the most computationally intensive parts of the CRYSTALS-Kyber algorithm. By optimizing where the algorithm spends most of its time and energy, we achieved more meaningful and holistic performance improvements.

A major innovation in our design is the integration of the K2RED modular reduction algorithm directly into the processor as a custom instruction. Rather than treating K2RED as a standalone FPGA module, we embedded it within the Hornet RISC-V pipeline, allowing low-latency, energy-efficient execution. This design not only reduces processing time but also minimizes hardware resource usage, offering a practical solution for resource-constrained systems.

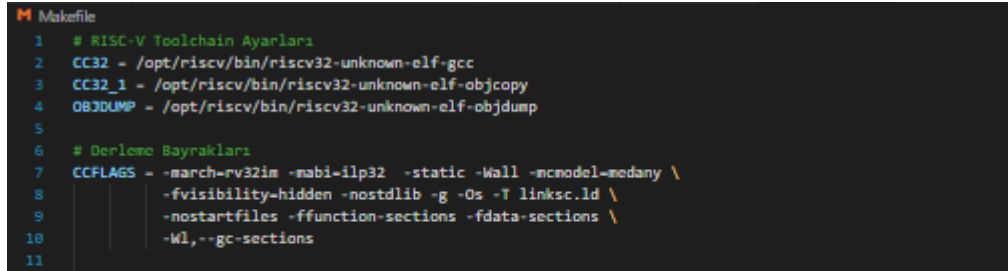
Additionally, our methodology follows a tightly coupled hardware-software co-design approach, avoiding complex external interfaces and reducing area overhead. This makes our system more scalable and easier to integrate into embedded or IoT platforms. Furthermore, by considering side-channel resistance during the design, we align with security best practices essential for cryptographic hardware.

Overall, our work contributes to the literature by providing a targeted, efficient, and secure method to accelerate core PQC operations, demonstrating that meaningful speedups can be achieved with minimal area and energy trade-offs, advancing the readiness of PQC for real-world hardware applications.

5. IMPLEMENTATION OF CRYSTALS-KYBER ON RISC-V

5.1 Running Applications on RISC-V Architecture

The selected architecture features a unified memory system with dedicated partitions: one segment for instruction memory, another for data memory, a reserved section for interrupt handling, and a specific region allocated for debugger operations. The translation of algorithms into executable RISC-V instructions is accomplished through a Makefile structure. This Makefile utilizes the RISC-V GNU Toolchain to convert the software through multiple representations: first to ELF format, then to assembly, binary, disassembly, and finally to hexadecimal format. Figure 5.1 enumerates the specific compiler flags and toolchain configurations employed in this process.



```
1 # RISC-V Toolchain Ayarları
2 CC32 = /opt/riscv/bin/riscv32-unknown-elf-gcc
3 CC32_1 = /opt/riscv/bin/riscv32-unknown-elf-objcopy
4 OBJDUMP = /opt/riscv/bin/riscv32-unknown-elf-objdump
5
6 # Derleme Bayrakları
7 CCFLAGS = -march=rv32im -mabi=ilp32 -static -Wall -mcnodel=medany \
8           -fvisibility=hidden -nostdlib -g -Oz -T linksc.ld \
9           -nostartfiles -ffunction-sections -fdata-sections \
10          -Wl,--gc-sections
11
```

Figure 5.1: Makefile Corresponding File Configurations and Compiler Flags

Figure 5.2 details both the functional components of the algorithm and the specific commands needed to convert these components into their corresponding file configurations.

```

12 # Dosyalar
13 SRC = top_1.c
14 CRT0 = crt0.s
15 TARGET = full
16
17 SOURCES = kem.c indcpa.c polyvec.c poly.c ntt.c cbd.c reduce.c verify.c test_speed.c
18 SOURCESKECCAK = $(SOURCES) fips202.c symmetric-shake.c
19 HEADERS = params.h kem.h indcpa.h polyvec.h poly.h ntt.h cbd.h reduce.h verify.h symmetric.h api.h
20 HEADERSKECCAK = $(HEADERS) fips202.h
21
22 SON = $(SOURCESKECCAK) $(HEADERSKECCAK)
23
24 # Hedefler
25 .PHONY: all clean
26
27 all: $(TARGET).elf $(TARGET).dis $(TARGET).bin $(TARGET).data
28
29 # ELF dosyasını oluşturma
30 $(TARGET).elf: $(SON) $(CRT0)
31 | $(CC32) $(SON) $(CRT0) $(CCFLAGS) -o $@
32
33 # Assembly dosyası oluşturma (opsiyonel)
34 $(TARGET).s: $(SON)
35 | $(CC32) -S -o $@ $(SON)
36
37 # Binary dosyasını oluşturma
38 $(TARGET).bin: $(TARGET).elf
39 | $(CC32_1) -O binary -j .init -j .text -j .rodata $< $@
40
41 # Disassembly dosyası oluşturma
42 $(TARGET).dis: $(TARGET).elf
43 | $(OBJDUMP) -SD $< > $@
44
45 # Binary dosyasını .data dosyasına dönüştürme
46 $(TARGET).data: $(TARGET).bin
47 | ./rom_generator $(TARGET).bin
48 | cp $(TARGET).data ../mem/memory_mulkk.data

```

Figure 5.2: Functional Structure of the Algorithm and Corresponding File Configuration Commands

A critical intermediate step in the transformation process is the generation of the Executable and Linkable Format (ELF) file. The ELF file encapsulates essential components such as compiled code, data segments, symbols, address information, and debugging data. These elements enable loaders to place the code at the designated memory address. The starting point for these addresses is defined in the crt.s file. This file specifies the `_start` label, which serves as the entry point for a RISC-V program. It performs initial system configurations and facilitates the transition to the main function, representing the first step required for the program to begin execution. Additionally, tools such as `riscv32-unknown-elf-objcopy` extract data from the ELF file to convert it into usable formats, such as the .mem file, which contains hexadecimal instructions.

The binary representations of the functions are stored in a .bin file. However, at this stage, the format does not yet adhere to the ROM boundaries, which are configured in the final step when generating the hexadecimal structure.

The assembly file (.s) and the disassembly file (.dis) provide visibility into the algorithm's progression at the assembly level. These files are instrumental in identifying and debugging errors in case of any issues during execution. They serve as valuable tools for analyzing the program's behavior and are particularly useful for future improvements in the development process.

In the final stage, the rom_generator program processes the 32-bit instructions from the .bin file and converts them into a hexadecimal format, which is then stored in a text file (.data). This step ensures that the instructions are in a format suitable for execution on the processor.

5.2 Execution of the CRYSTALS-Kyber Algorithm in Software

To understand the background and functionality of the Kyber algorithm, several functions were thoroughly examined. Two separate folders were created to organize the development process. The first folder was dedicated to ensuring that software modifications could be implemented without affecting the output. To achieve this, random functions responsible for generating coins were removed to ensure transparency in comparisons. Instead, outputs from single executions of these random functions were hardcoded as fixed values. This approach allowed for easier comparison of new results with previous ones when modifications were made to the algorithm. Specifically, it facilitated the analysis of outputs from key generation, encryption, and decryption processes, enabling a more streamlined evaluation of the algorithm's performance.

To ensure the interoperability of the examined functions, a test_speed.c file was developed. This file sequentially executes the key generation, encryption, and decryption codes. As illustrated in Figure 3.5.1, the code defines key and key2 as plaintext inputs. During the key generation phase, a public key is produced, which is then used in the encryption phase to generate a ciphertext. The resulting ciphertext is subsequently decrypted using the secret key generated during the key generation

phase. The resulting plaintexts are compared at the end of the main function to verify their equivalence, confirming the correctness of the process and allowing further development to proceed.

```

12  int main()
13  {
14      unsigned char *a;
15      unsigned char *b;
16      unsigned int i;
17      unsigned char pk[CRYPTO_PUBLICKEYBYTES] = {0};
18      unsigned char sk[CRYPTO_SECRETKEYBYTES] = {0};
19      unsigned char ct[CRYPTO_CIPHTEXTBYTES] = {0};
20      unsigned char key[CRYPTO_BYTES] = {0};
21      unsigned char key2[CRYPTO_BYTES] = {0}; // For storing decrypted key
22
23
24      for(i=0;i<NTESTS;i++) {
25          crypto_kem_keypair(pk, sk); // Generate public and secret key
26      }
27      for(i=0;i<NTESTS;i++) {
28          crypto_kem_enc(ct, key, pk); // Encrypt the key with public key
29      }
30      for(i=0;i<NTESTS;i++) {
31          crypto_kem_dec(key2, ct, sk); // Decrypt the ciphertext to get the key
32      }
33      a = key;
34      b = key2;
35
36      for (i = 0; i < CRYPTO_BYTES; i++) {
37          if (a[i] != b[i]) {
38              printf("a");
39              break;
40          }
41          else
42              printf("%d,", a[i]);
43      }
44      return 0;

```

Figure 5.3: Main function to test the algorithm to see if it works correctly

5.3 Optimizing CRYSTALS-Kyber For Hardware

To enable the CRYSTALS-Kyber algorithm to operate efficiently on hardware and to simplify it to its most basic form, several modifications to its functions were necessary. The first modification involved removing the printf function, as it lacks relevance in the RISC-V architecture and does not contribute to the algorithm's functionality in a hardware context. The second change entailed eliminating unnecessary header (.h) files to streamline the Kyber algorithm to its simplest

version. The third modification focused on removing the random functions embedded within the algorithm. Instead of relying on these functions, they were executed once to obtain the desired outputs, which were then hardcoded as fixed arrays and variables. This replacement of random coins with constant values was critical not only for simplification but also to ensure consistent algorithm execution speed, preventing variations due to random number generation.

As a final step, to facilitate testing the algorithm on an FPGA, the `test_speed.c` program incorporated the `volatile` keyword. In the C programming language, the `volatile` keyword informs the compiler that a variable's value may change due to external factors, preventing unwanted optimizations. In the context of the RISC-V processor running on the FPGA, direct access to specific hardware addresses is required. The `volatile` keyword ensures that the contents of these addresses are read from or written to memory each time, bypassing compiler optimizations that could otherwise interfere with accurate hardware interactions.

```
14 int main()
15 {
16     volatile unsigned int *LED_ADDR = (volatile unsigned int *) 0x0003FFD8;
17     volatile unsigned int *LED_ADDR_1 = (volatile unsigned int *) 0x0003FFD4;
18     unsigned int j;
19     uint8_t pk[CRYPTO_PUBLICKEYBYTES] = {0};
20     uint8_t sk[CRYPTO_SECRETKEYBYTES] = {0};
21     uint8_t ct[CRYPTO_CIPHTEXTBYTES];
22     uint8_t key[CRYPTO_BYTES];
23     uint8_t key2[CRYPTO_BYTES];
24
25     *LED_ADDR_1 = 242425;
26
27     *LED_ADDR_1 = 2;
28     crypto_kem_keypair(pk, sk); // Generate public and secret key
29
30     *LED_ADDR_1 = 3;
31     crypto_kem_enc(ct, key, pk); // Encrypt the key with public key
32
33     *LED_ADDR_1 = 4;
34     crypto_kem_dec(key2, ct, sk); // Decrypt the ciphertext to get the key
35     *LED_ADDR_1 = 5;
36
37     for (j = 0; j < CRYPTO_BYTES; j++) {
38         if (key[j] != key2[j]) {
39             *LED_ADDR_1 = 6;
40         }
41     }
42
43     if (*LED_ADDR_1 == 6) {
44         *LED_ADDR = 0;
45     } else {
46         *LED_ADDR = 1;
47     }
```

Figure 5.4: The main function to test the algorithm on RISC-V

The address 0x0003FFD8 is designated as the debug address for the Hornet processor, corresponding to memory address 65526 in the processor's memory space. When this address is assigned a value of 0, the processor halts and signals a failure. Conversely, when it is assigned a value of 1, the processor also halts but indicates a successful operation. Another critical memory address, 0x0003FFD4, which maps to address 65525 in the processor's memory, is used to monitor function transitions within the algorithm. This address updates its value after the completion of each key generation, encryption, and decryption function. By observing these changes during simulation, it is possible to measure the execution time of critical functions, providing insights into the algorithm's performance.

As depicted in the accompanying figure, the memory layout is divided into four distinct regions. The first region specifies the size of the instruction memory, the second defines the data memory range, the third is reserved for interrupts, and the fourth represents the address used for debugging purposes. Due to the Hornet system's division of memory addresses by a factor of four, the address 0x0003FFD8 corresponds to the calculated memory address 65526 (i.e., $0x0003FFD8 / 4 = 65526$).

```

assign slave_adr_begin[0] = 32'h0000_0000;
assign slave_adr_end[0] = 32'h0000_3C40;

assign slave_adr_begin[1] = 32'h0000_0000;
assign slave_adr_end[1] = 32'h0003_FFDC;

assign slave_adr_begin[2] = 32'h0003_FFDD;
assign slave_adr_end[2] = 32'h0003_FFEC;

assign slave_adr_begin[3] = 32'h0003_FFD8;
assign slave_adr_end[3] = 32'h0003_FFD8;

```

Figure 5.5: Addresses to Be Used for Executing the CRYSTALS-Kyber Algorithm on a RISC-V Processor

Following the generation of hexadecimal instructions and the invocation of the .mem file containing these instructions within the testbench, the simulation of the code was performed. A critical aspect of this phase is configuring the memory size based on the requirements of the instructions. To interpret the simulation results effectively, the registers, wires, and memory addresses to be observed must be identified through the design objects and included in the simulation setup. Analysis is conducted by monitoring changes in the previously defined volatile memory addresses. The timing results obtained from the behavioral simulation are recorded and will serve as a baseline for comparisons in subsequent stages. The results from the initial simulation run are considered the reference point for evaluating future iterations and improvements.

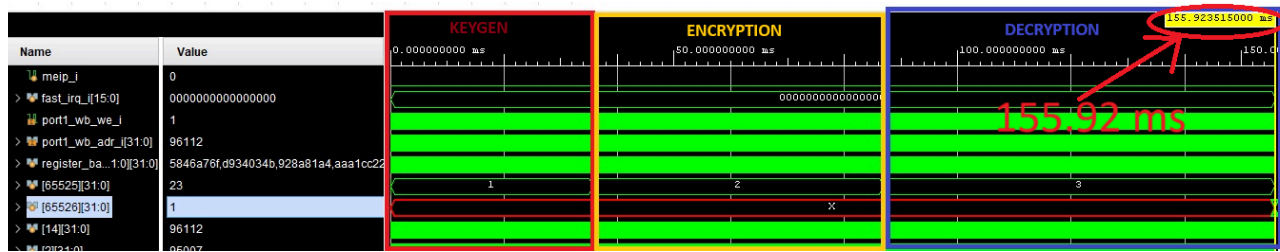


Figure 5.6: Timing Results of the CRYSTALS-Kyber Algorithm on a RISC-V Processor with the I Instruction set

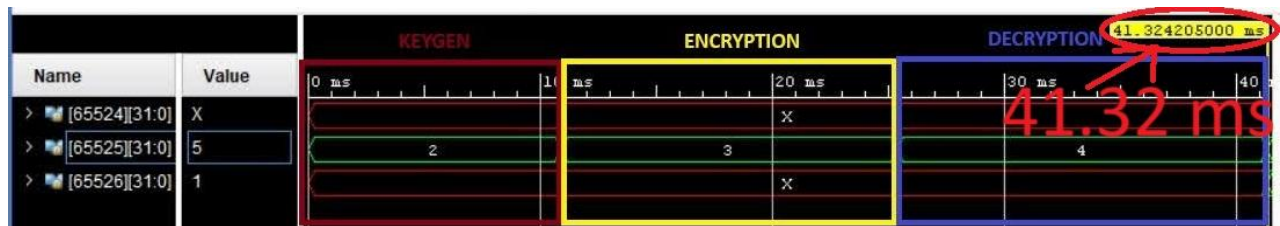


Figure 5.7: Timing Results of the CRYSTALS-Kyber Algorithm on a RISC-V Processor with I and M Instruction sets

5.4 Code Profiling for the CRYSTALS-Kyber Algorithm

After verifying that the Kyber algorithm can be successfully executed on the processor, the next step involves identifying the most frequently used and time-consuming functions within the algorithm. The primary objective of this project is to enhance the performance of post-quantum cryptographic algorithms, such as

Kyber, on open-source processors, demonstrating that optimizations are feasible and outlining a clear methodology for achieving them. To this end, software-based code profiling of the Kyber algorithm is essential. Profiling is a technique used to analyze a program's execution, determining the duration of each function's operation and identifying which processes consume the most resources. The goal of profiling is to detect bottlenecks, slow segments, or sections of the code that unnecessarily consume resources, thereby enabling performance optimization. Through profiling, developers can pinpoint critical code sections that significantly impact execution time or resource usage, allowing targeted improvements to enhance overall system efficiency.

```

1 CC=/usr/bin/gcc
2 CFLAGS += -Wall -Wextra -Werror -Wpedantic -Wredundant-decls -Wshadow -Wpointer-arith -O0 -fomit-frame-pointer
3 NISTFLAGS += -Wno-unused-result -O3 -fomit-frame-pointer
4
5 CC_P = gcc
6 CFLAGS_p += -Wall -g -pg
7 PROF = gprof
8
9 SOURCES = cbd.c fips202.c indcpa.c test_speed.c kem.c ntt.c poly.c polyvec.c reduce.c verify.c symmetric-shake.c
10 HEADERS = aes256ctr.h api.h sha2.h kem.h cbd.h fips202.h indcpa.h ntt.h params.h poly.h polyvec.h reduce.h verify.h symmetric.h
11
12 EXECUTABLE = PQCGenKAT_kem
13
14 .PHONY: all clean profile
15
16 SON = $(SOURCES) $(HEADERS)
17
18 all: $(EXECUTABLE)
19
20 $(EXECUTABLE): $(SON)
21     $(CC) $(CFLAGS) $(SOURCES) -o $(EXECUTABLE)
22
23 profile: $(SOURCES)
24     $(CC_P) $(CFLAGS_p) -O0 $(SOURCES) -o $(EXECUTABLE) -pg
25     ./$(EXECUTABLE)
26     $(PROF) -p $(EXECUTABLE) gmon.out > profiling_report.txt
27
28 clean:
29     -rm -f $(EXECUTABLE) gmon.out profiling_report.txt

```

Figure 5.8: Makefile for Profiling

The gprof tool, supported by the GCC compiler, was utilized for code profiling of the Kyber algorithm. A dedicated step was incorporated into the Makefile to enable profiling, with the -O0 flag specified to disable compiler optimizations, ensuring accurate function-level timing analysis. The profiling data was collected using gprof and exported to a text file named profiling_report.txt. Analysis of this file revealed that the NTT function, along with its associated sub-functions, constitutes the most frequently used and time-intensive group of functions within the algorithm.

```

1 Flat profile:
2
3 Each sample counts as 0.01 seconds.
4
5 % cumulative self self total
6 time seconds seconds calls us/call us/call name
7 20.91 1.22 1.22 880000 1.38 1.38 KeccakF1600_StatePermute
8 13.77 2.02 0.80 290560000 0.00 0.00 pqcrystals_kyber512_ref_montgomery_reduce
9 11.19 2.67 0.65 285440000 0.00 0.01 fgmul
10 7.57 3.10 0.44 100000 4.40 8.91 pqcrystals_kyber512_ref_ntt
11 7.57 3.54 0.44 70000 6.29 14.78 pqcrystals_kyber512_ref_invntt
12 6.97 3.95 0.41 134400000 0.00 0.00 pqcrystals_kyber512_ref_barrett_reduce
13 4.82 4.23 0.28 23040000 0.01 0.04 pqcrystals_kyber512_ref_basemul
14 3.27 4.42 0.19 9760000 0.02 0.02 load64
15 2.58 4.57 0.15 120000 1.25 1.25 rej_uniform
16 2.50 4.71 0.14 12760000 0.01 0.01 store64
17 2.41 4.86 0.14 360000 0.39 1.68 keccak_absorb
18 1.89 4.96 0.11 60000 1.83 1.83 pqcrystals_kyber512_ref_poly_frombytes
19 1.55 5.05 0.09 190000 0.47 0.47 pqcrystals_kyber512_ref_poly_add
20 1.55 5.14 0.09 80000 1.12 1.25 cbd3
21 1.46 5.23 0.09 28160000 0.00 0.00 pqcrystals_kyber512_ref_csubq
22 1.38 5.31 0.08 280000 0.29 1.06 pqcrystals_kyber512_ref_poly_reduce
23 1.20 5.38 0.07 180000 0.39 5.16 pqcrystals_kyber512_ref_poly_basemul_montgomery

```

Figure 5.9: Results for Profiling of CRISTAL-Kyber

The NTT function encompasses the fgmul function, which in turn incorporates the Montgomery reduction function. Profiling results clearly indicate that these functions are the most time-consuming components of the Kyber algorithm during its execution. Consequently, throughout the project, it was decided to focus on optimizing, modifying, and integrating these functions into the hardware implementation to improve overall performance.

5.5 Number Theoretic Transform

The Number Theoretic Transform (NTT) is a widely utilized tool in lattice-based cryptography to accelerate polynomial multiplication. The NTT operation transforms a polynomial from its coefficient representation to a point-value representation by substituting specific values into the polynomial. This transformation enables efficient computation of matrix and vector multiplication operations. Essentially, the NTT is a finite-field adaptation of the Fast Fourier Transform (FFT), achieving a quasilinear time complexity of $O(n \log n)$ for polynomial multiplication, a significant improvement over the naive schoolbook method, which has a time complexity of $O(n^2)$. This performance enhancement substantially accelerates the Kyber algorithm, making it a critical component of its implementation.

In the Kyber algorithm, polynomials are first transformed into the NTT domain. Polynomial multiplication is then performed in this domain, after which the result is converted back to the polynomial domain through an operation known as the Inverse NTT (INTT). The INTT process is similar to the NTT but executes comparable operations in reverse order. During polynomial multiplication, the increase in bit size is mitigated using modular reduction functions to keep the values within the desired range. The NTT employs an n -th primitive root of unity, denoted as ω , at each stage, while the INTT uses the inverse, ω^{-1} , along with an additional multiplication by n^{-1} to complete the transformation. In the Kyber implementation, the parameters $N = 256$ and $Q = 3329$ are chosen to define the polynomial ring and modulus, respectively.

$$\hat{a}_k = \sum_{i=0}^{n-1} a_i \omega_n^{ik} \bmod q, \quad k = 0, 1, \dots, n-1.$$

Figure 5.10: NTT function

$$a_i = n^{-1} \sum_{k=0}^{n-1} \hat{a}_k \omega_n^{-ik} \bmod q, \quad i = 0, 1, \dots, n-1. \quad (2)$$

Figure 5.11: INTT function

$$c = INTT_{GS}^{\psi^{-1}} (NTT_{CT}^{\psi}(a) \circ NTT_{CT}^{\psi}(b))$$

Figure 5.12: Polynomial Multiplication with NTT domain

Two primary algorithms, known as butterfly functions, are employed to compute the results of polynomials in the NTT domain. These are the Cooley-Tukey algorithm and the Gentleman-Sande algorithm[17]. The NTT algorithm utilizes precomputed coefficients, denoted as ω^{\square} , which serve as inputs. Additionally, the twiddle vector, denoted as ω^{\square} , is a constant that is also precalculated. These values are precomputed and embedded within the algorithm to avoid the computationally expensive modulo operations, which are time-consuming at both the software and hardware levels.

Precomputing these coefficients enhances efficiency, and their values are publicly accessible for use in implementations.

The polynomial multiplication process in the NTT algorithm can be expressed as follows:

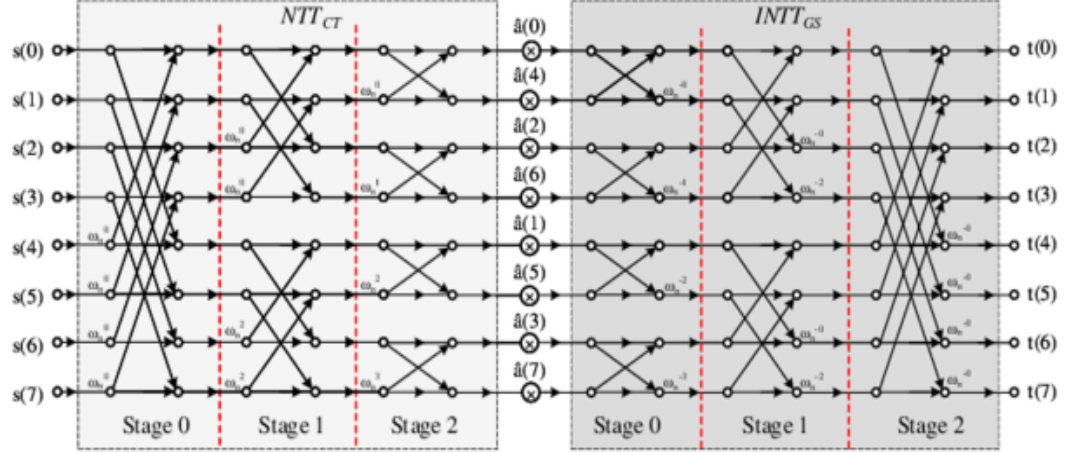


Figure 5.13: Representation of Polynomial Multiplication With NTT and INTT Algorithm With 8 Bits [29]

In the context of the NTT, the number of butterfly operation stages is determined by the logarithm base 2 of the input size. For example, for an 8-bit input, where $\log_2(8)$ equals 3, three stages of butterfly operations are required to complete the NTT process. In the Kyber algorithm, which processes two 256-bit polynomials as input, the NTT computation involves $\log_2(256) = 8$ stages. This indicates that the NTT operation for these polynomials is completed after eight stages of butterfly computations, enabling efficient polynomial multiplication in the NTT domain.

5.6 Butterfly

The butterfly operations, which are the cornerstone of both the NTT and Inverse NTT (INTT) processes, are divided into two distinct algorithms: the Cooley-Tukey and Gentleman-Sande algorithms[17]. The Cooley-Tukey algorithm forms the primary building block of the NTT, while the Gentleman-Sande algorithm is utilized in the INTT process. These algorithms involve modular multiplication, addition, and subtraction operations, where modularity refers to the inclusion of reduction

operations to keep values within a specified range. Profiling results indicate that the Montgomery reduction function, one of the most frequently used functions in the Kyber algorithm, is integral to these operations.

As illustrated in Figure 5.14, the constant ω represents a precomputed value used within the algorithm. The inputs U and V , shown in the figure, are external values to be processed (e.g., for encryption). The butterfly operation begins by multiplying the input V by the constant ω , and the result is then added to U to produce an output E . Similarly, V is subtracted from U to yield another output E . To ensure the results remain within a 12-bit range, a reduction operation is performed using the Kyber-specific constant $Q = 3329$. The combined process of multiplication and reduction is encapsulated in the Kyber algorithm's `fqmul` function, which efficiently handles these modular arithmetic operations.

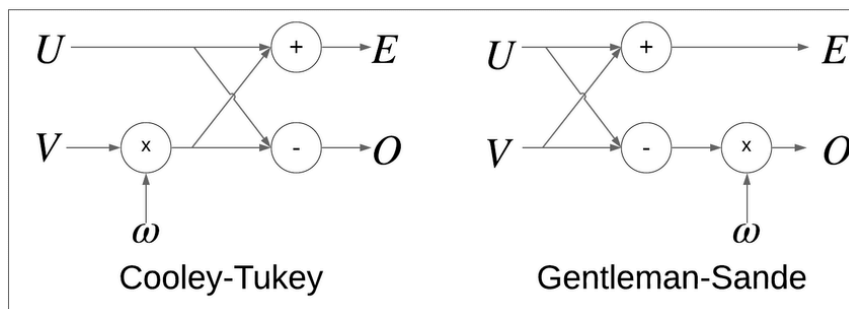


Figure 5.14: Butterfly Representation

5.7 Modulo Multiplication

The `fqmul` function that is used on Kyber is a modular multiplication algorithm that encompasses both multiplication and reduction operations. In the Kyber algorithm, the result of a 32-bit multiplication is reduced to fit within the bit constraints of the modulus. The original NIST version of the Kyber algorithm employs the Montgomery reduction algorithm for this purpose. The `fqmul` function is invoked repeatedly within loops in the NTT, INTT, and `basemul` functions, making it a critical component of the algorithm's execution. Profiling results reveal that `fqmul` is called 290,530,000 times when the Kyber algorithm is executed 10,000 times, equating to approximately 29,053 calls per single execution.

Given the significant computational load of the `fqmul` function, the project prioritized optimizing the NTT and its associated sub-algorithms, with an initial focus on the Montgomery reduction algorithm. Accelerating such a frequently invoked function is expected to substantially enhance the overall performance of the Kyber algorithm. This optimization effort is not only relevant to Kyber but also holds broader significance for nearly all PQC algorithms, particularly lattice-based homomorphic encryption schemes, where reduction operations play a pivotal role. Thus, improving the efficiency of the reduction algorithm represents a critical advancement in the field of PQC.

5.8 Montgomery Reduction Algorithm

In the Butterfly function, a modular reduction operation is required after each multiplication. This operation is performed using the Montgomery reduce function, which implements the Montgomery reduction algorithm [REF]. However, since modular reduction is highly time- and resource-consuming in hardware, it constitutes the most time-consuming step within the `fqmul` structure. Therefore, we first analyzed the Montgomery algorithm to determine the steps to follow in our work.

The Montgomery algorithm is a method developed to perform modular operations more efficiently. It utilizes two constant values: `qinv` and `kyber_q`. Here, `qinv` is the modular inverse of `kyber_q` modulo 2^{16} , `kyber_q` is the modulus used in the Kyber algorithm and is fixed at 3329.

The Montgomery algorithm applies multiplication, subtraction, and bit-shifting operations to compute the result of a modulo q reduction for a given input a . Since multiplication with large numbers leads to significant energy consumption and processing time in hardware, efficient implementation of modular operations is critical for the performance of `fqmul`. Optimizing the Montgomery algorithm helps mitigate these performance losses and improves overall efficiency.

```

/*****
* Name:      montgomery_reduce
*
* Description: Montgomery reduction; given a 32-bit integer a, computes
*              16-bit integer congruent to a * R^-1 mod q,
*              where R=2^16
*
* Arguments:  - int32_t a: input integer to be reduced;
*              has to be in {-q2^15,...,q2^15-1}
*
* Returns:    integer in {-q+1,...,q-1} congruent to a * R^-1 modulo q.
*****/
int16_t montgomery_reduce(int32_t a)
{
    int32_t t;
    int16_t u;

    u = a*QINV;
    t = (int32_t)u*KYBER_Q;
    t = a - t;
    t >>= 16;
    return t;
}

```

Figure 5.15: K2RED Algorithm in C Code

The K2RED[26] algorithm performs modular reduction using only assignment and subtraction operations, making it highly suitable for hardware implementation. Its simple arithmetic structure eliminates the need for resource-intensive operations such as multiplication or division with large numbers. By relying solely on bit-shifting and subtraction for modular reduction, K2RED enables implementations on programmable hardware like FPGAs with fewer logic blocks, lower latency, and reduced power consumption.

In NTT-based post-quantum cryptographic algorithms that use fixed moduli, K2RED offers significant advantages compared to the Montgomery method. The Montgomery approach requires multiple complex operations such as multiplication, addition, and division for modular multiplication, which leads to increased hardware resource usage and latency. In contrast, K2RED's streamlined design delivers more efficient hardware performance.

This makes K2RED especially advantageous in applications like IoT devices and embedded systems, where energy efficiency and area savings are crucial. Its

scalability and hardware compatibility position it as a preferred choice in such resource-constrained environments.

In this context, K2RED is considered a more suitable alternative to the Montgomery algorithm for designing cryptographic processors that require high speed and low power consumption.

Algorithm 12 K²RED Modular Multiplication Algorithm

Require: A, B, P, n with $P < 2^n$, $P = k2^m + 1$, k is odd, $k < 2^m$, $0 \leq A, B < 2^n$, $t < 2n$

Ensure: $C' = k^2(AB) \bmod P$

- 1: $R = AB$
 - 2: $R_l \leftarrow (r_m, r_{m-1}, \dots, r_1, r_0)_2$
 - 3: $R_h \leftarrow (r_{2n-1}, \dots, r_{m+1})_2$
 - 4: $C \leftarrow kR_l - R_h$
 - 5: $C_l \leftarrow (c_m, c_{m-1}, \dots, c_1, c_0)_2$
 - 6: $C_h \leftarrow (c_t, \dots, c_{m+1})_2$
 - 7: $C' \leftarrow kC_l - C_h$
-

Figure 5.16 K2RED Algorithm [26]

6. INSTRUCTION SET EXTENSION OF RISC-V PROCESSOR

RISC-V processors support different instruction set variants, allowing for flexible and optimized solutions tailored to various computational needs. For instance, the 32I instruction set provides basic arithmetic operations such as addition, subtraction, and shifting. In contrast, the 32IM instruction set extends these capabilities by including more complex operations like multiplication and division. This diversity enables processors to be designed for higher efficiency and performance depending on the specific application. The flexible nature of the RISC-V architecture offers a significant advantage over other processor architectures. Leveraging this advantage, we can implement custom instruction set extensions tailored specifically for our cryptographic algorithm, Kyber. By directly supporting the most frequently used and power-consuming operations of Kyber at the hardware level, we can improve processor performance while reducing power consumption. To achieve this, we first need to design and integrate the necessary hardware extension, then introduce the new instruction to the compiler we use, and finally employ this instruction effectively within our algorithm. This approach ensures an integrated optimization across both hardware and software layers.

6.1 Replacing Montgomery with K2RED in Software and Testing K2RED on RISC-V

In order to concretely evaluate the advantages provided by the K2RED algorithm, particularly in modular reduction operations, compared to the Montgomery algorithm, the Montgomery reduction method used within the Kyber post-quantum cryptographic algorithm was initially replaced with K2RED at the software level. Following this modification, both the software and hardware performance of the algorithm were analyzed. If the change had not resulted in a significant improvement in execution time, the original Montgomery algorithm would have remained the candidate for hardware implementation.

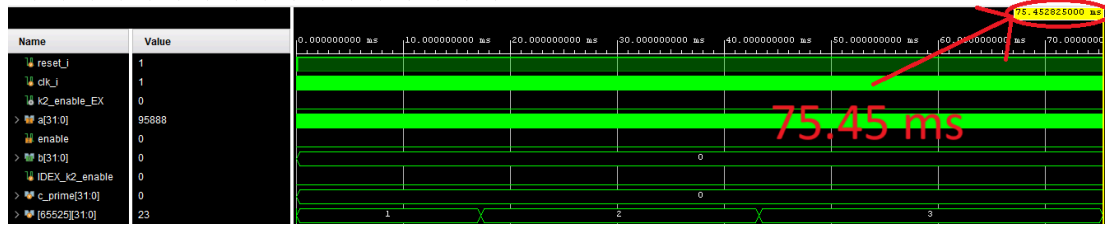


Figure 6.1: Replacing Montgomery with K2RED in Software on the Hornet Processor Using the RV32I Instruction Set

The experimental results presented in Figure 6.1 clearly demonstrate the impact of replacing the Montgomery reduction function with the K2RED algorithm in the Kyber post-quantum cryptographic scheme at the software level. As a result of this modification, the total execution time of the Kyber algorithm was measured as 75.45 ms, compared to 155.9 ms when using the original implementation based on Montgomery reduction. In other words, this software-only substitution led to a performance improvement of approximately 51.7%, without any hardware-level optimization. This significant gain was achieved solely through the simplification of the modular reduction method. The K2RED algorithm performs modular reduction using only low-cost operations such as bit shifting and subtraction, which substantially reduces computational overhead. In contrast, the Montgomery method involves multiple multiplications, additions, and divisions, leading to longer execution times in software. These findings highlight that the K2RED algorithm offers a more efficient approach to modular arithmetic, and should be considered a favorable alternative, particularly in applications where execution time is critical. Moreover, the performance improvement observed at the software level strongly suggests that even greater acceleration can be expected when the algorithm is implemented in hardware.

6.2 Instruction Set Extension for K2RED

After demonstrating that the K2RED algorithm outperforms the Montgomery method at the software level, the next step was to implement K2RED as a custom instruction set extension for hardware acceleration. However, before integrating any custom instruction into the hardware, it is essential to ensure that the compiler recognizes and supports this new instruction. Therefore, introducing the proposed instruction to

the compiler and properly integrating it into the compilation process represents a critical stage in the overall design flow.

6.2.1 Introducing a New Instruction to the RISC-V GNU Toolchain

To incorporate a custom instruction into the instruction set architecture, it is first necessary to define the opcode, func3, and, if applicable, func7 fields for the new instruction. These values must be carefully chosen to ensure they do not conflict with existing instructions in the RISC-V instruction set. Once it has been confirmed that there is no overlap, the RISC-V GNU toolchain can be updated accordingly. The initial step involves modifying the “riscv-gnu-toolchain/binutils/include/riscv-opc.h” file, which contains fixed instruction encodings for the RISC-V architecture. In this file, the MATCH_ and MASK_ macros for the new instruction must be defined. The MATCH macro specifies the fixed bit pattern of the instruction, including the opcode, func3, and optionally func7 fields. The MASK macro indicates which bits in the instruction encoding are relevant for matching by setting them to 1; this ensures that a bitwise AND operation between the instruction and the MASK will yield the MATCH value. Importantly, fields such as rs1, rs2, rd, and imm are considered variables must be set to 0 in both MATCH and MASK definitions. After these values are specified, the instruction is declared using the DECLARE_INSN macro in the same file. This macro follows the format DECLARE_INSN(instruction_name, match_value, mask_value), and it allows the assembler to recognize and handle the new instruction. These steps are fundamental for successfully integrating a custom instruction into the RISC-V GNU toolchain.

```
#define MATCH_K2RED 0x77  
#define MASK_K2RED 0x707F
```

Figure 6.2: Definition of MATCH and MASK Values

```
#ifdef DECLARE_INSN  
DECLARE_INSN(k2red, MATCH_K2RED, MASK_K2RED)
```

Figure 6.3: Declaration of the Function Using DECLARE_INSN

Next, the new instruction needs to be added to the “riscv-gnu-toolchain/binutils/opcodes/riscv-opc” file to define how it works in the processor. The K2RED instruction uses the I-type format, which includes one source register (rs1), one immediate value (imm), and one destination register (rd). The immediate value is used to avoid having to provide a constant number every time the function is called. In this format, the fields d (destination), s (source), and j (immediate) are used. As shown in Figure 6.4, the K2RED instruction is defined in the riscv-opc file. This definition explains how the processor will handle the instruction and is set up according to the I-type instruction format.

```
{ "k2red", 0, INSN_CLASS_I, "d,s,j", MATCH_K2RED, MASK_K2RED, match_opcode, 0 },
```

Figure 6.4: Definition of the K2RED Instruction in the riscv-opc File

6.2.2 Calling the Defined Instruction

The defined instruction should be integrated using inline assembly when called. This approach allows assembly code to be embedded directly within the C programming language, enabling efficient utilization of custom hardware instructions. As shown in Figure 6.5, the defined instruction is successfully invoked within the C code through inline assembly. This technique not only facilitates seamless incorporation of custom instructions into the programming workflow but also maximizes the performance benefits provided by the underlying hardware.

```
int16_t k2_red_Reduce(int32_t a)
{
    int16_t C_prime;
    asm volatile("k2red    %[z], %[x], %[y]\n\t"
        : [z] "=r" (C_prime)
        : [x] "r" (a), [y] "i" (200)
        ) ;
}
```

Figure 6.5: Calling the Defined Instruction Using Inline Assembly

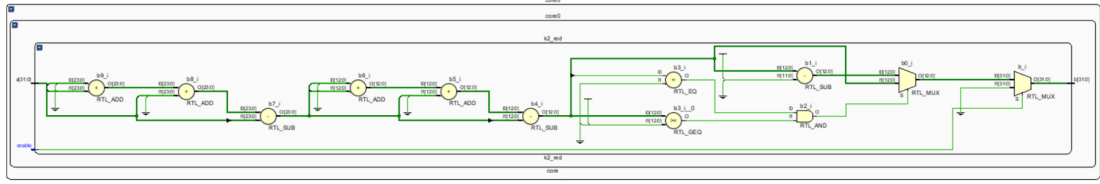


Figure 6.6: RTL Schematic of the K2RED Module

6.2.3 Testing K2RED on Simulation

The K2RED instruction was integrated into the ALU to perform modular reduction of the input by 3329 and produce the output within a single cycle. All necessary adjustments for the newly designed ALU were implemented in the related modules. Subsequently, the Kyber algorithm, incorporating the inline K2RED instruction, was re-executed, and the performance and accuracy results were thoroughly analyzed.

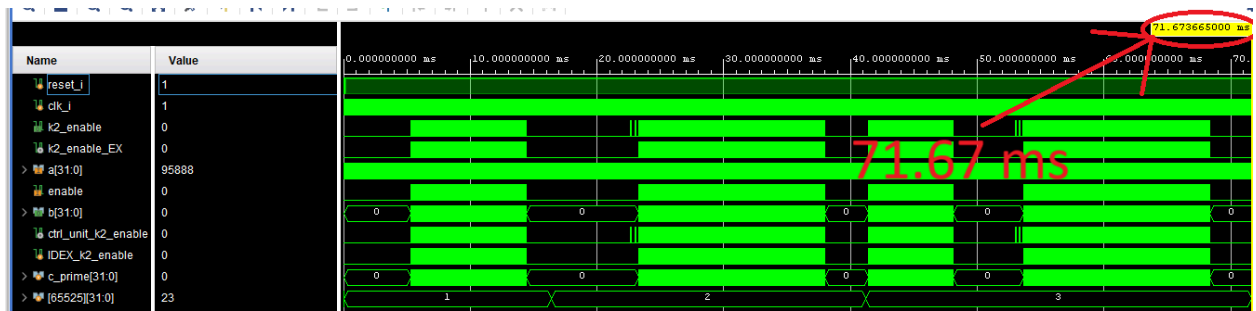


Figure 6.7: K2RED instruction test in RV32I set

As shown in Figure 6.7, the hardware implementation of the K2RED algorithm reduced the total execution time of the Kyber algorithm to 71.67 ms. Furthermore, simulation results indicate that the K2RED operations are heavily utilized during the NTT, inverse NTT, and base multiplication stages. In these stages, modular reduction is repeatedly performed, playing a critical role in the majority of the algorithm's execution.

6.3 Modulo Multiplication Unit on Hardware

The K2RED instruction performs only the modular reduction step and is not sufficient on its own to implement a full modular multiplication operation. In contrast, the fgmul function used in the Kyber algorithm performs a multiplication of

two input values followed by a modular reduction with respect to 3329. To achieve this complete functionality in hardware, a multiplication stage was added before the existing K2RED module. Since the input values in Kyber are already reduced modulo 3329, the multiplication can be performed within 12 bits. This allows efficient implementation of the multiplication stage using DSP blocks, resulting in a complete and optimized modular multiplication unit.

6.3.1 K2RED in RV32IM Instruction Set

The modmul instruction performs a multiplication operation. In the RV32I instruction set, multiplication is quite time-consuming. For a fair comparison, it was anticipated that the modmul algorithm would achieve higher efficiency when executed on the RV32IM instruction set. Therefore, for comparison purposes, the K2RED algorithm was also run on the RV32IM instruction set, completing in 37.6 ms.

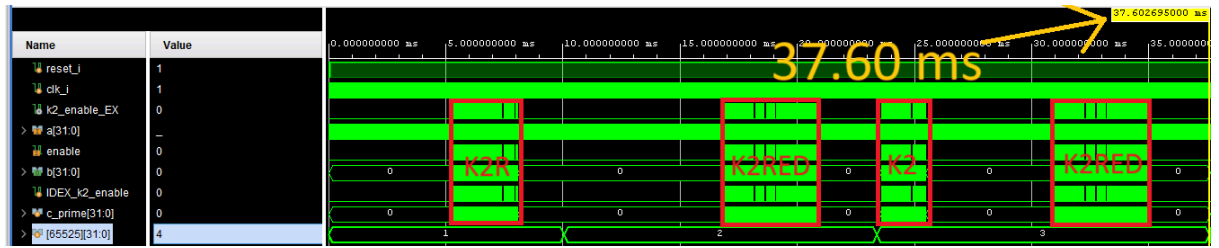


Figure 6.8: K2RED Instruction Simulation in RV32IM

6.3.2 Modmul Instruction Call

Since the K2RED instruction operates as a sub-operation within the modmul instruction, it was deemed appropriate to replace the previously introduced K2RED command with the modmul instruction. Unlike the K2RED module, the modmul instruction requires two inputs instead of one. Consequently, it was redefined within the instruction set as an R-type instruction instead of an I-type. While the funct3 and opcode values remained unchanged, the funct7 field was assigned the value 000_0000. This modification enables the entire fgmul operation to be invoked with a single inline assembly instruction. As a result of redefining the instruction as R-type, the fgmul process can be executed via one inline assembly command. As illustrated

in Figure 6.9, the modmul instruction is called using inline assembly, performing the fqmul operation with a single command. This approach is expected to yield more efficient performance on hardware.

```
static int16_t fqmul(int16_t a, int16_t b) {
    int16_t C_prime;
    asm volatile
    (
        "modmul    %[z], %[x], %[y]\n\t"
        : [z] "=r" (C_prime)
        : [x] "r" (a), [y] "r" (b)
    ) ;
    return C_prime;
}
```

Figure 6.9: Inline Assembly Usage of Modmul

6.3.3 Testing Modmul Unit

As shown in Figure 6.10, the Modmul instruction was executed on the RV32IM set, completing the entire algorithm in 32.73 ms. Compared to K2RED, which took 37.6 ms, this reduction corresponds to a 13% speedup.

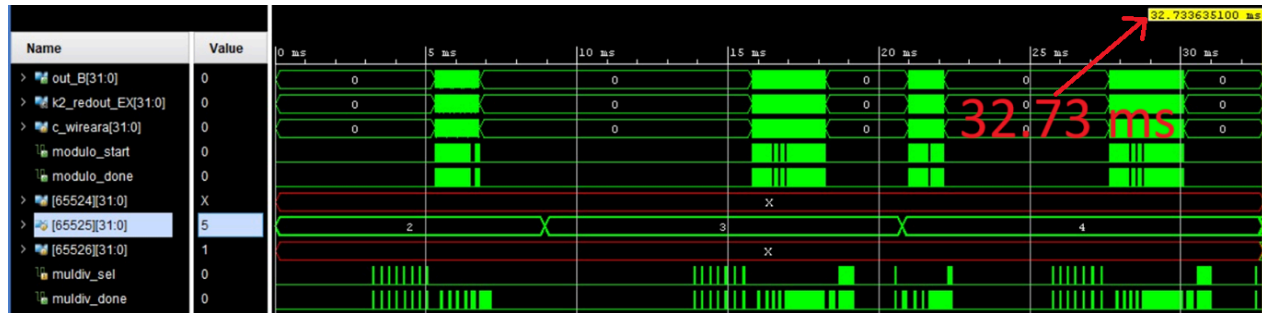


Figure 6.10: Modmul Instruction Simulation in RV32IM

6.4 Butterfly Unit

The NTT algorithm is implemented through the repeated invocation of butterfly structures. As shown in Figure 6.11, the innermost for-loop represents a single butterfly operation, enabling the sequential execution of multiple butterfly

computations. Each butterfly structure has three inputs: $r[j]$, $r[j+len]$, and a zeta value. The zeta values form a fixed sequence that is updated at regular intervals. Different zeta sequences are used in the NTT and inverse NTT (INTT) processes.

Since the number of zeta values is too large to fit entirely into registers, accessing memory is required each time these values are updated, which increases processing time. The repeated butterfly operations in the NTT structure further amplify this delay. Therefore, storing frequently used zeta values in a dedicated memory and retrieving them from a secondary memory when needed aims to accelerate the Kyber algorithm.

```

/*****
* Name:      ntt
*
* Description: Inplace number-theoretic transform (NTT) in Rq
*              input is in standard order, output is in bitreversed order
*
* Arguments:  - int16_t r[256]: pointer to input/output vector of elements
*                  of Zq
*****/
void ntt(int16_t r[256]) {
    unsigned int len, start, j, k;
    int16_t t, zeta;

    k = 1;
    for(len = 128; len >= 2; len >>= 1) {
        for(start = 0; start < 256; start = j + len) {
            zeta = zetas[k++];
            for(j = start; j < start + len; ++j) {
                t = fqmul(zeta, r[j + len]);
                r[j + len] = r[j] - t;
                r[j] = r[j] + t;
            }
        }
    }
}

```

Figure 6.11: NTT Algorithm Implementation in C

6.4.1 Planned Butterfly Module Design

A dedicated memory unit will be designed to store the Zeta values. This memory will update its own counter structure during the counting and resetting of the Zeta values. To support this, two inline assembly functions are planned to perform the count and

reset operations. As shown in Figure 6.12, the Zeta values are first determined, then multiplied, followed by modular reduction. The previously designed modmul instruction is used for the multiplication and modular reduction operations. The resulting value will then be either added to or subtracted from the input value $r[j]$. These operations produce two different outputs. For the RISC-V architecture, two separate instructions are required for these two different output operations; these instructions are defined as `butter_ave` and `butter_b`, respectively. The designed unit includes four signals: `zeta_cnt`, `zeta_res`, `butter_ave`, and `butter_b`. To reduce input dependencies, the `zeta_cnt` and `zeta_res` signals are defined as U-type instructions. Since the `butter_ave` instruction uses two inputs, it is defined as an R-type instruction, while the `butter_b` instruction is designated as an I-type instruction.

Figure 6.12: Schematic of Planned Design

As shown in Figure 6.13, the `fqmul` function is used not only in the NTT process but also in the INTT and BaseMul functions. Therefore, even if the butterfly instruction is added to the hardware, the need for the `modmul` instruction remains. For this reason, operations will be performed with both the butterfly and `modmul` instructions implemented in hardware.

```

void invntt(int16_t r[256]) {
    unsigned int start, len, j, k;
    unsigned int i;
    int16_t t, zeta;

    k = 0;
    for(len = 2; len <= 128; len <<= 1) {
        for(start = 0; start < 256; start = j + len) {
            zeta = zetas_inv[k++];
            for(j = start; j < start + len; ++j) {
                t = r[j];
                r[j] = barrett_reduce(t + r[j + len]);
                r[j + len] = t - r[j + len];
                r[j + len] = fqmul(zeta, r[j + len]);
            }
        }
    }

    for(j = 0; j < 256; ++j)
        r[j] = fqmul(r[j], zetas_inv[127]);
}

void basemul(int16_t r[2],
             const int16_t a[2],
             const int16_t b[2],
             int16_t zeta)
{
    r[0] = fqmul(a[1], b[1]);
    r[0] = fqmul(r[0], zeta);
    r[0] += fqmul(a[0], b[0]);

    r[1] = fqmul(a[0], b[1]);
    r[1] += fqmul(a[1], b[0]);
}

```

Figure 6.13: INTT and Basemul Algorithms Implementation in C

6.4.3 The Test of Butterfly Instruction

As a result of the combined operation of the Butterfly and Modmul instructions, the Kyber algorithm was executed in 32.29 ms. In comparison, the original algorithm took 41.32 ms to run, meaning our enhancement achieved a 21.85% speedup.

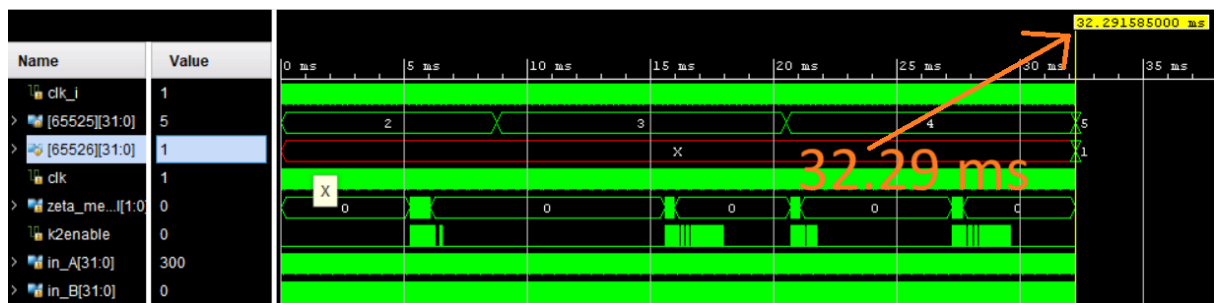


Figure 6.14: Butterfly Instruction Simulation in RV32IM

7. ENERGY AND POWER ANALYSIS COMPARISON

We evaluated the performance of the Kyber algorithm implemented purely in software on our existing processor, without any additional hardware modifications. This evaluation measured and compared key metrics including execution time (clock cycles), hardware usage, resource utilization, and total energy consumption.

The tables below present the detailed results of this comparison. Each table includes the measured values for each parameter along with an analysis of the findings. These analyses demonstrate the effectiveness of our work.

Table 6.1 : Comparison of Pure Software and Hardware/Software Implementation Execution Times

Pure Software (Clock Cycle)	Hardware/Software Co-Design (Clock Cycle)	Acceleration Rate
4132000	3229000	%21.85

Table 6.2 : Hardware and Area Comparison Between the Basic Hornet Core and the Version with Integrated Butterfly Unit

	First Version	Resource Utilization (%)	New Version	Resource Utilization (%)
LUT	6527	%10.29	6846	%10.8
FF	2488	%1.96	2606	%2.06
BRAM	64	%47.41	64	%47.41
IO	20	%9.52	20	%9.52
DSP	0	%0	2	%0.83

Table 6.3: Comparison of Results Between the Hornet Core with Butterfly Extension and the Original Hornet Core

	First Version	New Version
Average Power	0.223 W	0.167 W
Period	15.32 ns	18.6ns
Frequency	65.28 MHz	53.76 MHz
Number of Clock Cycles	4132000	3229000
Energy	14.116 mJ	10.02 mJ

As shown in the tables, the enhancements we implemented led to clear improvements, resulting in reductions in both total execution time and overall energy consumption. These gains highlight the effectiveness of the proposed modifications and demonstrate the impact of our design choices on system performance and efficiency.

8. SIDE-CHANNEL ANALYSIS

Cryptographic hardware is traditionally designed to speed up frequently used operations and securely store secret keys. However, such hardware can be vulnerable to side-channel attacks, which exploit the analysis of physical characteristics. These attacks extract secret information by taking advantage of application-specific physical phenomena, including power consumption, electromagnetic emissions, and timing variations. In many cases, side-channel attacks prove to be more effective than conventional cryptanalysis techniques [27].

The hardware unit we designed aims to accelerate the Kyber algorithm. Kyber employs the NTT algorithm in both encryption and decryption stages. The NTT algorithm consists of butterfly modules that perform critical operations such as polynomial multiplication. However, this acceleration process also requires testing the system's resistance against side-channel attacks. Therefore, we decided to conduct side-channel analysis to evaluate the security of our design.

The primary goal of side-channel analysis is to obtain secret information such as keys, messages, or clues related to them. In these types of attacks, the functions executed during the use of the message or key are especially critical for analysis. These functions can cause sensitive information to be leaked through physical side effects observable on the processor, such as power consumption, timing variations, or electromagnetic emissions.

When examining the operational steps in the Kyber algorithm, the parts that directly interact with the key and message stand out. In particular, the NTT operation is prominent in terms of the risk of leaking secret information. As explained in previous sections, the NTT process performs polynomial multiplication and may contain sensitive information that attackers seek to obtain in its inputs. For example, one of the inputs to the inverse NTT function used during decryption is the secret key itself. The other input is the value u , which is derived from the ciphertext and transformed

into the NTT domain. By multiplying these two inputs and passing the result through the inverse NTT, the transmitted message is recovered by subtracting it from the value v , which is also derived from the ciphertext. As can be seen, the inputs to the NTT function are directly related to the secret key and ciphertext. Therefore, by carefully analyzing the physical leakages generated during these operations, it is possible to extract critical information through side-channel attacks. In this context, analyzing the resistance of the butterfly module we designed against side-channel attacks has been deemed meaningful and necessary for the overall system security. As a method, the entire Kyber algorithm will first be run on the RISC-V platform to perform Simple Power Analysis (SPA) in an attempt to extract key-related information. This analysis will serve as an important step to evaluate the system's resistance to physical leakages and to identify potential security vulnerabilities.

8.1 Simple Power Analysis

The Kyber algorithm was executed on the Hornet processor implemented on the Nexys A7-100T development board, and side-channel analysis was performed. The software was updated to continuously encrypt previously entered data, after which the analysis was carried out. However, the obtained traces were quite noisy. The main reason for this is the five-stage pipeline architecture of the RISC-V processor.

Therefore, in the initial stage, the effect of a high computational load operation on the side-channel trace of the RISC-V processor was intended to be observed. For this purpose, modifications were needed on both the software and hardware sides.

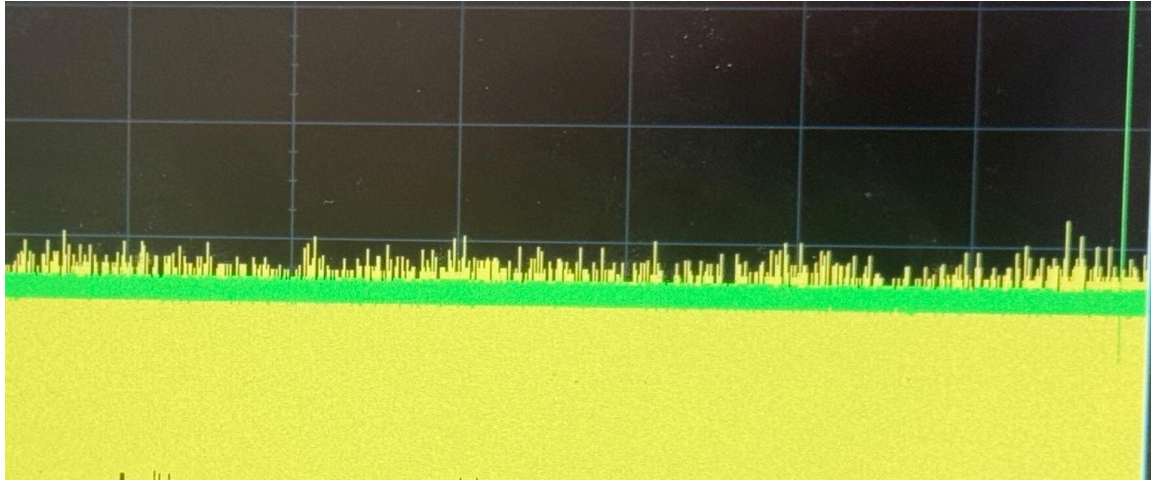


Figure 8.1: Basic Power Analysis of Kyber

8.2 Simple Power Analysis Observation of "Heartbeat" on RISC-V

Before performing analyses aimed at revealing secret information, it is necessary to observe the effects of the operations we carried out on the FPGA on power consumption. For this purpose, a secondary memory module was added to the processor's memory access stage. This module duplicates write operations to a specific address in the main memory, thereby creating an observable physical response.

The main goal here is to amplify low-level power consumption signals that cannot be directly observed on the oscilloscope, making them measurable. This allows us to verify whether the operations performed on the FPGA have a tangible physical counterpart.

While designing this module, the attribute (`* dont_touch = "true" *`) was used. This directive prevents Vivado from optimizing, removing, or merging the specified signals or cells during synthesis and implementation stages. The `dont_touch` command ensures that Vivado does not modify the targeted structure, preserving the signal or module exactly as defined in the design. This is particularly useful for signals that Vivado might consider "unused" but are crucial for side-channel analysis or external connections, guaranteeing their persistence in the design.

8.3 Assigning the Trigger Signal

A trigger mechanism was established to reduce the processing time and to measure only the signal region of interest. Although there are various methods to create a trigger, triggering was performed via UART signals to avoid interfering with the system hardware. Thanks to the developed structure, the first trigger signal is generated after the board processes the last receiver bit via UART. Subsequently, a second trigger signal is produced just before the first transmitter bit is sent. This allows the oscilloscope to ignore periods when the board is idle and focus directly on the signal region of interest for analysis.

8.4 Evaluation of First Results

First, the value '0' was written to the address specified via UART, followed by a short delay, after which the value '-1' (all bits set to 1) was written to the same address. During this operation, a high energy consumption was expected due to the transition of all bits from 0 to 1. This should be observed as a spike on the oscilloscope. As shown in Figure 8.2, the theoretically expected instantaneous energy increase was successfully observed.

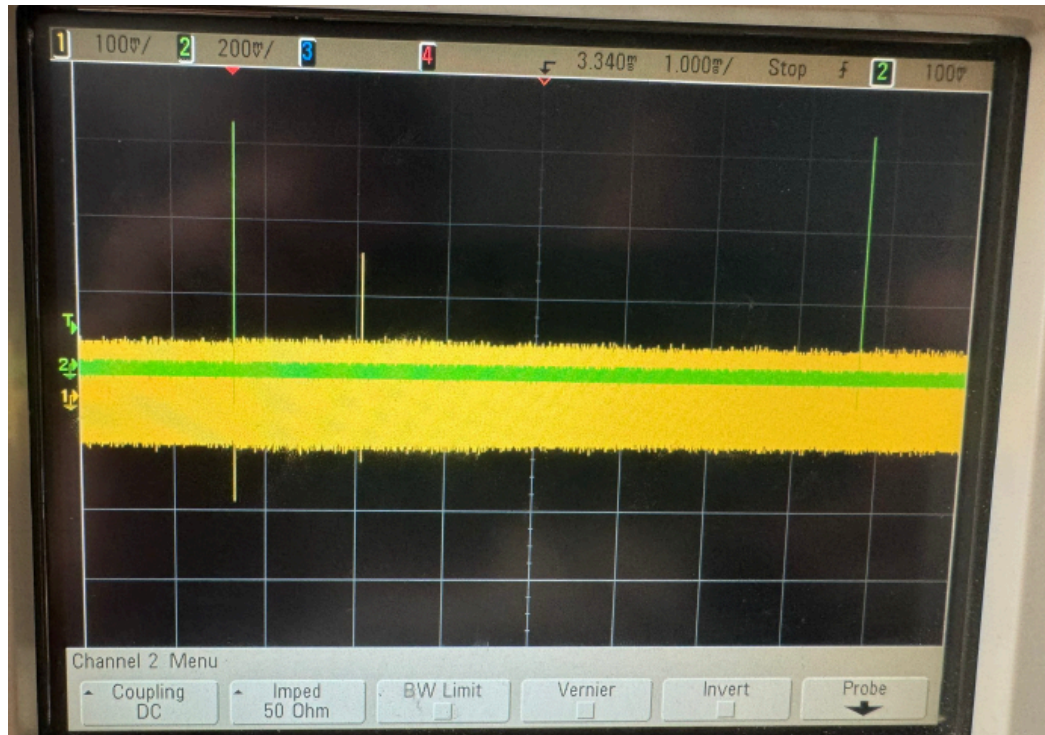


Figure 8.2: Observation of the Power Consumption Spike During Bit Transition

8.5 UART Code for the Butterfly Module

As a result of our side-channel analyses, it has been demonstrated that the operations performed on the hardware have an electromagnetic counterpart. The next step will be to perform differential power analysis by applying different input values to our custom-designed module. For this purpose, a C code has been written as an initial step.

```

volatile int count;
volatile uint8_t input_array[8];

uart uart0;
int main()
{
    SET_MTVEC_VECTOR_MODE();
    count = 0;
    uart_init(&uart0, 0x00008010);

    while(1)
    {
        uart_transmit_string(&uart0, "Send 16 bytes of input block!\n", 30);
        ENABLE_GLOBAL_IRQ();
        ENABLE_FAST_IRQ(0);
        while(count != 8);
        butterfly(input_array);
        uart_transmit_string(&uart0, (const char*)input_array, 8);
        count = 0;
    }

    void mti_handler() {}
    void exc_handler() {}
    void mei_handler() {}
    void msi_handler() {}
    void fast_irq0_handler()
    {
        char *rx_ptr = (char*)(uart0.base_addr) + UART_RX_ADDR_OFFSET;
        char rx_byte = *rx_ptr;
        input_array[count] = rx_byte;
        count++;
        if(count == 8)
            DISABLE_GLOBAL_IRQ();
    }
}

```

Figure 8.3: UART Code Designed for Butterfly Unit

Subsequently, the software we wrote in C was embedded into our RISC-V hardware using the previously described method. Afterwards, as shown in Figure 8.4, a MATLAB code was developed to enable communication between our board and the computer.

```

for row = 1:100
    writeline(device, ':SINGLE');
    pause(1); % Wait for acquisition

    % UART communication
    num1 = int32(data(row, 1));
    num2 = int32(data(row, 2));
    bytes1 = flip(typecast(num1, 'uint8'));
    bytes2 = flip(typecast(num2, 'uint8'));
    bytesToSend = [bytes1, bytes2];
    disp(['Row ', num2str(row), ' UART bytes sent: ', sprintf('%d ', bytesToSend)]);

    flush(s, "input");
    for i = 1:8
        write(s, bytesToSend(i), 'uint8');
        pause(0.001);
    end

    pause(0.5); % Wait for response
    if s.NumBytesAvailable >= 8
        receivedData = read(s, 8, 'uint8');
        disp(['Row ', num2str(row), ' UART bytes received: ', sprintf('%d ', receivedData)]);
        receivedNum1 = typecast(uint8(flip(receivedData(1:4))), 'int32');
        receivedNum2 = typecast(uint8(flip(receivedData(5:8))), 'int32');
        expectedNum1 = int32(data(row, 3));
        expectedNum2 = int32(data(row, 4));
        if receivedNum1 == expectedNum1 && receivedNum2 == expectedNum2
            disp(['Row ', num2str(row), ': Verification successful!']);
        else
            disp(['Row ', num2str(row), ': Verification failed!']);
        end
    else
        disp(['Row ', num2str(row), ': Insufficient UART data: ', num2str(s.NumBytesAvailable)]);
    end

    % Read waveform (channel 1 - olcum)
    writeline(device, [':WAVEform:SOURCE ', olcum_channel]);
    writeline(device, ':WAVEform:POINTS 5000'); % Re-ensure points setting
    ready = 0;
    while ready ~= 1
        writeline(device, '*OPC?');
        ready = double(readline(device));
    end
    fprintf(device, ':WAVEform:DATA?');
    trv

```

Figure 8.4: MATLAB UART Code for Side Channel Analysis

9. REALISTIC CONSTRAINTS, CONCLUSIONS, AND RECOMMENDATIONS

9.1 Application Area of the Study

This study focuses on improving the performance and efficiency of post-quantum cryptographic algorithms, especially CRYSTALS-Kyber, by using hardware-based acceleration on a RISC-V core. As quantum computers continue to develop, current cryptographic algorithms such as RSA and ECC are no longer considered secure. At the same time, side-channel attacks like power analysis and electromagnetic leakage also create serious security risks.

Post-quantum algorithms are designed to resist quantum attacks, but they usually require more processing time and power due to operations like modular multiplication and NTT. These performance needs can be too heavy for traditional CPUs, especially in low-power systems.

In this work, the RISC-V-based Hornet processor was used to add custom hardware instructions for critical Kyber functions like modular reduction. These changes helped reduce both power usage and execution time. As a result, the system becomes more suitable for real-world applications such as embedded systems, secure communication devices, and IoT platforms.

The methods developed in this project can be used in designing cryptographic processors that support quantum-safe algorithms with better efficiency. This makes the study valuable for both academic research and practical use in industries that require secure, low-power cryptographic solutions for the future.

9.2 Realistic Design Constraints

This project was developed using open-source hardware and software tools, offering flexibility and allowing architectural modifications without licensing limitations.

However, several realistic design constraints had to be considered to ensure a functional and efficient implementation.

One major constraint is the critical path delay. Custom hardware components integrated into the processor must not exceed the processor's original timing path. Otherwise, the clock frequency would need to be reduced, negatively impacting performance. To avoid this, all modules, especially K2RED and Butterfly units, were optimized to fit within timing limits.

Another key constraint is FPGA resource usage. Logic elements, memory blocks, and DSP slices are limited, so efficient area utilization was essential. All custom units were carefully designed to remain within the available FPGA capacity while delivering performance improvements.

Power efficiency also plays a role, especially in embedded or low-power environments. Although performance was the primary focus, unnecessary switching and logic complexity were minimized to avoid excessive energy consumption.

Finally, extending the instruction set required corresponding updates in the software toolchain and careful synchronization between hardware and software components. Ensuring compatibility and system stability was crucial throughout the design process.

In summary, while open-source development allowed great flexibility, constraints such as timing, area, power, and integration complexity guided the overall design strategy.

9.2.1 Cost of the Project

The designed processor was implemented on the Nexys A7-100T development board available in our university's laboratory. The oscilloscope and probes needed for side-channel analysis were also provided by the university lab. For software development, the Xilinx Vivado student license was utilized.

9.2.2 Standards

For hardware design, the IEEE standards for Verilog and the RISC-V Instruction Set Manual are used. Likewise, the C programming language implementation adhered to the C99 standard. A

9.2.3 Social, Environmental, and Economic Impact

The successful completion of our project has delivered impactful results with significant social, environmental, and economic implications. Innovations developed in the field of post-quantum cryptography have proven to be highly applicable on open-source processors designed for cryptographic purposes, achieving notable improvements in both processing speed and energy efficiency.

Cryptography plays a critical role globally, ranging from secure communication and financial transactions to data privacy and beyond. With the rapid advancement of quantum computing technologies, the demand for quantum-resistant cryptographic algorithms has become increasingly vital to protect global information exchange.

Through the implementation of these algorithms on a RISC-V-based architecture optimized for low power consumption and high efficiency, our project successfully enhanced both the performance and security of post-quantum cryptographic systems. These achievements are expected to accelerate the adoption of quantum-resistant security standards, establish more secure communication infrastructures, and increase public trust in digital platforms.

In conclusion, our project has made a meaningful contribution to the field of cybersecurity, establishing a strong foundation for secure global communication and data protection. It represents a significant step forward in building a future-proof digital infrastructure that can withstand the challenges posed by quantum computing.

9.2.4 Health and Safety Risks

Since the environments used in the project are software-based, no health or safety risks have been identified.

9.3 Results

When the results of the project are examined, a 21.85% reduction in the number of clock cycles has been achieved. The measurements clearly indicate that the overall energy consumption has also decreased. These outcomes demonstrate that the primary objective of the project, which is accelerating the execution time of the Kyber algorithm by implementing part of the NTT (Number Theoretic Transform) operation in hardware, has been successfully accomplished. By offloading the most time-consuming portion of the NTT to custom hardware, specifically the Butterfly unit, significant improvements were achieved not only in execution speed but also in energy efficiency. The reduced computation time directly contributed to lower energy usage. This confirms that hardware and software co-design is an effective strategy for optimizing post-quantum cryptographic algorithms on RISC-V platforms.

9.4 Suggestions for Future Work

We have implemented a hardware module based on the Cooley-Tukey butterfly structure, which is commonly used in the NTT. This structure performs multiplication before the addition and subtraction steps. However, the Inverse NTT (INTT) typically utilizes the Gentleman-Sande butterfly structure, where the operation order is reversed: addition and subtraction are performed first, followed by multiplication.

To ensure compatibility with both NTT and INTT, there are two possible approaches. The first option is to modify and generalize the existing instruction to support both butterfly structures by introducing a configurable operation sequence. The second option is to design a separate instruction specifically optimized for the Gentleman-Sande structure used in INTT.

REFERENCES

- [1] **Shor, P. W.** (1994). Algorithms for quantum computation: Discrete logarithms and factoring. *Proceedings 35th Annual Symposium on Foundations of Computer Science*, Santa Fe, NM, USA, 124-134.
- [2] **R. L. Rivest, A. Shamir, and L. Adleman.** (1978). A method for obtaining digital signatures and public-key cryptosystems. *Commun. ACM* 21, 2 (Feb. 1978), 120–126.
- [3] **Miller, V. S.** (1985). Use of elliptic curves in cryptography. In *Advances in Cryptology - CRYPTO '85, Santa Barbara, California, USA, August 18-22, 1985, Proceedings* (Vol. 218, pp. 417-426). Springer.
- [4] **Url-1** <<https://www.nist.gov/cybersecurity/what-post-quantum-cryptography>>
- [5] **Chris Peikert** (2016), "A Decade of Lattice Cryptography", *Foundations and Trends® in Theoretical Computer Science*: Vol. 10: No. 4, pp 283-424.
- [6] **NIST.** (2016). Report on Post-Quantum Cryptography. NISTIR 8105. National Institute of Standards and Technology.
- [7] **Avanzi, R., Bos, J., Ducas, L., Kiltz, E., Lepoint, T., Lyubashevsky, V., Schanck, J. M., Schwabe, P., Seiler, G., & Stehlé, D.** (2021). CRYSTALS-Kyber: Algorithm specifications and supporting documentation (version 3.01), 31 January. <https://pq-crystals.org/kyber/data/kyber-specification-round3.pdf>
- [8] **Url-2** <<https://csrc.nist.gov/pqc-standardization>>
- [9] **Valiev, K. A.** (2005). Quantum computers and quantum computations. *Physics-Uspekhi*, 48(1), 1.
- [10] **Shah, P., Prajapati, P., Patel, R., Patel, D.** (2025). Post Quantum Cryptography: A Gentle Introduction of Lattice-Based Cryptography (Kyber, NTRUCrypto). In: Fong, S., Dey, N., Joshi, A. (eds) *ICT Analysis and Applications. ICT4SD 2024. Lecture Notes in Networks and Systems*, vol 1161. Springer, Singapore.
- [11] **Micciancio, D., & Goldwasser, S.** (2002). Complexity of lattice problems: A cryptographic perspective. Springer.
- [12] **Hofheinz, D., Hövelmanns, K., & Kiltz, E.** (2017). A modular analysis of the Fujisaki–Okamoto transformation. In *Theory of Cryptography Conference (TCC)* (pp. 341–371). Springer.
- [13] **Katz, J., & Lindell, Y.** (2020). *Introduction to Modern Cryptography* (3rd ed.). Chapman and Hall/CRC.

- [14] **Regev, O.** (2009). On lattices, learning with errors, random linear codes, and cryptography. *Journal of the ACM*, 56(6), 1–40.
- [15] **Galloway, P.** (2017). *Learning with errors: Foundations and applications* [Lecture notes]. University of California, Santa Barbara, CS293G.
- [16] **Avanzi, R., Bos, J., Ducas, L., Kiltz, E., Lepoint, T., Lyubashevsky, V., Schanck, J. M., Schwabe, P., Seiler, G., & Stehlé, D.** (2021). *CRYSTALS-Kyber: Algorithm specifications and supporting documentation* (version 3.01), 31 January.
- [17] **Cooley, J.W., Tukey, J.W.** (1965) An algorithm for the machine calculation of complex fourier series. *Math. Comput.* 19(90), 297–301
- [18] **Url-3** <<https://riscv.org/about/>>
- [19] **Url-4** <<https://github.com/yavuz650/RISC-V/tree/main>>
- [20] **Waterman, A., Lee, Y., Patterson, D., & Asanović, K.** (2016, May 31). *The RISC-V instruction set manual, Volume I: User-Level ISA* (Version 2.1). CS Division, EECS Department, University of California, Berkeley.
- [21] **Url-5** <<https://github.com/riscv-collab/riscv-gnu-toolchain>>
- [22] **Url-6** <<https://ubuntu.com/desktop/wsl>>
- [23] **Url-7** <<https://csrc.nist.gov/projects/post-quantum-cryptography/selected-algorithms>>
- [24] **Celik, A., Yilmaz, F., Korkmaz, M. A., & Ors, B.** (2023). Implementation of CRYSTALS-Kyber post-quantum algorithm using RISC-V processor. In *2023 30th IEEE International Conference on Electronics, Circuits and Systems (ICECS)* (pp. 1–4). IEEE.
- [25] **Masera, G., & Martina, M.** (2023). *Integration and optimization of a RISC-V based Keccak accelerator* (Master's thesis, Politecnico di Torino). Corso di Laurea Magistrale in Ingegneria Elettronica. <https://webthesis.biblio.polito.it/26725/>
- [26] **Can, F.** (2024). Hardware design of K2RED modular multiplication algorithm used in Number Theoretic Transform for post-quantum cryptography and homomorphic encryption (Master's thesis, Istanbul Technical University). https://web.itu.edu.tr/~orssi/thesis/2024/FurkanCan_tez.pdf
- [27] **S. B. Ors,** (2005) “Hardware Design of Elliptic Curve Cryptosystems and Side-Channel Attacks”, Ph. D. thesis, Katholieke Universiteit Leuven.
- [28] **Üstün, A.** (2024). Instruction set extension of RISC-V core with Plantard modular reduction for number theoretic transform used in post-quantum cryptography (Master's thesis, Istanbul Technical University) https://web.itu.edu.tr/~orssi/thesis/2024/AliUstun_tez.pdf

- [29] **Bisheh Niasar, M., Azarderakhsh, R., & Mozaffari Kermani, M.** (2021, May). High-speed NTT-based polynomial multiplication accelerator for CRYSTALS-Kyber post-quantum cryptography.

CURRICULUM VITAE



Name Surname :Berk ARSLANPENCESI
Place and Date of Birth :İstanbul, Türkiye – 2001
E-Mail : arslanpencesi22@itu.edu.tr

EDUCATION:

Istanbul Technical University / Electronics and Communication Engineering
(2020-2025)

EXPERIENCE:

- ASELSAN / Scholar Student / 2025

CONFERENCE:

- EPoSS Annual Forum 2025

CURRICULUM VITAE



Name Surname : Rabia Göksel DEMIRAY

Place and Date of Birth : Istanbul / 16.07.2002

E-Mail : rabiagoksel0@gmail.com

EDUCATION:

Istanbul Technical University / Electronics and Communication Engineering
(2020-2025)

EXPERIENCE:

- Tübitak Bilgem TÜTEL / Digital Design Engineer / Part-Time 2024
- Tübitak Bilgem TÜTEL / Digital Design Engineer / Intern 2024

CONFERENCE:

- EPoSS Annual Forum 2025