

ISTANBUL TECHNICAL UNIVERSITY ★ GRADUATE SCHOOL

**INSTRUCTION SET EXTENSION OF RISC-V CORE WITH PLANTARD
MODULAR REDUCTION FOR NUMBER THEORETIC TRANSFORM
USED IN POST-QUANTUM CRYPTOGRAPHY**

M.Sc. THESIS

Ali ÜSTÜN

Department of Electronics and Communication

Electronics Engineering Programme

FEBRUARY 2024

ISTANBUL TECHNICAL UNIVERSITY ★ GRADUATE SCHOOL

**INSTRUCTION SET EXTENSION OF RISC-V CORE WITH PLANTARD
MODULAR REDUCTION FOR NUMBER THEORETIC TRANSFORM
USED IN POST-QUANTUM CRYPTOGRAPHY**

M.Sc. THESIS

**Ali ÜSTÜN
(504201204)**

Department of Electronics and Communication

Electronics Engineering Programme

Thesis Advisor: Prof. Dr. S. Berna ÖRS YALÇIN

FEBRUARY 2024

İSTANBUL TEKNİK ÜNİVERSİTESİ ★ LİSANSÜSTÜ EĞİTİM ENSTİTÜSÜ

**RISC-V ÇEKİRDEĞİNİN TALİMAT SETİNİN POST-KUANTUM
KRİPTOGRAFİDE KULLANILAN SAYISAL TEORİK DÖNÜŞÜM İÇİN
PLANTARD MODÜLER İNDİRGEME METODU İLE GENİŞLETİLMESİ**

YÜKSEK LİSANS TEZİ

**Ali ÜSTÜN
(504201204)**

Elektronik Mühendisliği Anabilim Dalı

Elektronik Mühendisliği Programı

Tez Danışmanı: Prof. Dr. S. Berna ÖRS YALÇIN

ŞUBAT 2024

Ali ÜSTÜN, a M.Sc. student of ITU Graduate School student ID 504201204 successfully defended the thesis entitled “INSTRUCTION SET EXTENSION OF RISC-V CORE WITH PLANTARD MODULAR REDUCTION FOR NUMBER THEORETIC TRANSFORM USED IN POST-QUANTUM CRYPTOGRAPHY”, which he/she prepared after fulfilling the requirements specified in the associated legislations, before the jury whose signatures are below.

Thesis Advisor : **Prof. Dr. S. Berna ÖRS YALÇIN**
Istanbul Technical University

Jury Members : **Prof. Dr. S. Berna Örs YALÇIN**
Istanbul Technical University

Assoc. Dr. Şerif BAHTİYAR
Istanbul Technical University

Assoc. Dr. Özgü CAN
Ege University

Date of Submission : **05 January 2024**
Date of Defense : **22 February 2024**

FOREWORD

First of all, I would like to thank my project advisor, Prof. Dr. Sıddıka Berna Örs Yalçın for guiding me through her advises and knowledge as well as sharing lots of her experiences with me.

Secondly, I would like to thank many others who have shared their precious time and insights.

Also, I would like to thank all my friends for their support. Their company has always kept me entertained and motivated in times of struggle.

Finally, I want to express my endless gratitude and appreciation for my family, who have supported our decisions and guided us with their experiences.

February 2024

Ali ÜSTÜN

TABLE OF CONTENTS

	<u>Page</u>
FOREWORD	vii
TABLE OF CONTENTS	x
ABBREVIATIONS	xi
LIST OF TABLES	xiii
LIST OF FIGURES	xvi
SUMMARY	xvii
ÖZET	xix
1. INTRODUCTION	1
1.1 Purpose of Thesis	1
2. MATHEMATICAL BACKGROUND	5
2.1 CRYSTALS-Kyber Post-Quantum Cryptography Algorithm	5
2.2 CRYSTALS-Dilithium Post-Quantum Cryptography Algorithm	6
2.3 Polynomial Multiplication	7
2.3.1 Schoolbook Polynomial Multiplication (SPM)	8
2.3.2 Number Theoretic Transform (NTT)	8
2.3.2.1 Comparison of NTT and SPM	10
2.4 Modular Reduction	11
2.4.1 Barret Modular Reduction	11
2.4.2 K ² Red Modular Reduction [1]	12
2.4.3 Plantard Modular Reduction	13
3. HARDWARE IMPLEMENTATION OF MODULAR ARITHMETIC ALGORITHMS	15
3.1 Barret Modular Reduction Algorithm	15
3.2 K ² red Modular Reduction	16
3.3 Plantard Modular Reduction	17
3.4 Plantard Modular Multiplication	18
4. RISC-V	21
4.1 RISC-V ISA	22
4.2 RISC-V GNU Toolchain Collection	25
4.2.1 Building the RISC-V GCC Toolchain	25
4.3 RISC-V Core Implementation	26
4.3.1 Microcontroller Structure based Ibex RISC-V in Core	26
4.4 Customization of RISC-V Core and GCC	27
4.4.1 RISC-V GNU Toolchain Modification	28
4.4.2 Adding New Instruction to RISC-V Core	29
4.4.2.1 Single Cycle Instruction Addition	30
4.4.2.2 Multi Cycle Instruction Addition	31
4.4.2.3 Software Application of Custom Instructions	31
5. RESULTS	35
5.1 Barrett Modular Reduction	35
5.2 Plantard Modular Reduction	35
5.3 Plantard Modular Multiplication Area-Performance Benchmarks	35
5.4 Plantard Modular Reduction in Ibex RISC-V core	36
5.4.1 Critical Path Analysis	39
5.4.2 Power and Energy Analysis	42
6. CONCLUSIONS	47
REFERENCES	49
APPENDIX A : Multi-Cycle Instruction Addition to RISC-V Core EX block ..	53
APPENDIX B : NTT functions that are used for comparison	55
APPENDIX C : Multi cycle Plantard Modular multiplication assembly example	57

APPENDIX D : Makefile example	59
-------------------------------------	----

ABBREVIATIONS

NTT	: Number Theoretical Transform
PQC	: Post-Quantum Cryptography
LBC	: Lattice-based Cryptography
SPM	: Schoolbook Polynomial Multiplication
LWE	: Learning with Errors
KEM	: Key Encapsulation Mechanism
HE	: Homomorphic Encryption
NIST	: National Institute of Standards and Technology
ENISA	: European Union Agency for Cybersecurity
FPGA	: Field Programmable Gate Array
FFT	: Fast Fourier Transform
BR	: Barret Reduction
SoC	: System On a Chip
SPI	: Serial Peripheral Interface
JTAG	: Joint Test Action Group
I2S	: Inter-IC Sound
DMA	: Direct Memory Access
RISC	: Reduced Instruction Set Computer
ISA	: Instruction Set Architecture
UART	: Universal Asynchronous Receiver Transmitter
GF	: Galois Field
GCC	: GNU Compiler Collection
CSR	: Control and Status Register
CPA	: Critical Path Analysis
SAIF	: Switching Activity Interchange Format

LIST OF TABLES

	<u>Page</u>
Table 2.1 : Security Levels of CRYSTALS-Kyber Algorithm with Respect to AES	6
Table 4.1 : Used Environment Settings.....	25
Table 5.1 : Implementation Utilization Results of the Modular Reduction Algorithms	35
Table 5.2 : Implementation Performance Results of the Modular Reduction Algorithm Designs	36
Table 5.3 : Utilization and Performance of Plantard modular multiplication algorithm for CRYSTALS-Kyber and CRYSTALS-Dilithium PQC Schemes	36
Table 5.4 : Performance Analysis of RISC-V Core with Combinations of Designed Instruction Extensions	38
Table 5.5 : Performance Analysis of RISC-V Core with Combinations of Designed Instruction Extensions	45

LIST OF FIGURES

	<u>Page</u>
Figure 2.1 : CRYSTALS-Kyber PQC Algorithm Diagram [2]	6
Figure 2.2 : An 8-point NTT-based polynomial multiplication [3]	9
Figure 3.1 : Barrett Modular Reduction Block Diagram	16
Figure 3.2 : RTL Schematic Diagram of the Design for Barrett Modular Reduction Algorithm for Kyber.....	16
Figure 3.3 : K ² red Modular Reduction Block Diagram	17
Figure 3.4 : Plantard Modular Reduction Block Diagram	17
Figure 3.5 : Plantard Modular Reduction CRYSTALS-Kyber RTL Diagram	18
Figure 3.6 : Comparisons of K ² RED and Plantard modular multiplication for CRYSTALS-Kyber [4]	19
Figure 3.7 : Comparisons of K ² RED and Plantard modular multiplication for CRYSTALS-Dilithium [4]	19
Figure 4.1 : RISC-V Instruction Sets and Extension [5].	22
Figure 4.2 : RISC-V Base Instruction Formats [5].	23
Figure 4.3 : Ibex RISC-V Core [6]	27
Figure 4.4 : RISC-V GNU Toolchain "riscv-opc.c" Modification for Custom Instruction.....	28
Figure 4.5 : Declaration of Custom Instruction in the RISC-V GNU Toolchain "riscv-opc.h"	28
Figure 4.6 : RISC-V GNU Toolchain "riscv-opc.h" Modification for Custom Instruction.....	29
Figure 4.7 : Single Cycle Plantard Modular Reduction Added to RISC-V ALU Block	30
Figure 4.8 : Average cycles with and without "plntrd_sc" modular reduction instruction.....	31
Figure 4.9 : Algorithm to benchmark modular multiplication	32
Figure 4.10 : Ibex RISC-V Core Modification in ID stage	32
Figure 4.11 : Software comparison of modular multiplication with using "plantard_mc" instruction	32
Figure 4.12 : Accessing MSBs of Result $n \times 2n \bmod 2^{2n}$ multiplication	33
Figure 5.1 : Modular multiplication with "pltrd_sc added"	37
Figure 5.2 : Modular multiplication with software modulo	37
Figure 5.3 : NTT performance comparison with "pltrd_sc" instruction and SW modulo	37
Figure 5.4 : Comparisons of RISC-V core areas with combinations of different Plantard modular reduction algorithms	39
Figure 5.4 : Comparisons of RISC-V core areas with combinations of different Plantard modular reduction algorithms (cont.)	40

Figure 5.5 : Comparisons of Critical Path in RISC-V core areas with combinations of different Plantard modular reduction algorithms (cont.) **41**

Figure 5.6 : Differences of Power Analysis with and without SAIF File..... **43**

Figure 5.7 : Differences of Power Analysis with and without SAIF File..... **44**

Figure D.1 : NTT software makefile example. **59**

INSTRUCTION SET EXTENSION OF RISC-V CORE WITH PLANTARD MODULAR REDUCTION FOR NUMBER THEORETIC TRANSFORM USED IN POST-QUANTUM CRYPTOGRAPHY

SUMMARY

The advent of quantum computers has underscored the importance of security. Traditional cryptography algorithms, such as RSA, are becoming increasingly vulnerable to quantum attacks. As a result, there is a growing need for cryptography algorithms that can withstand the threat of quantum computers. In response to this need, the National Institute of Standards and Technology (NIST) has initiated a project to standardize robust post-quantum cryptography algorithms. Among the selected algorithms, this study focuses on the CRYSTALS-Kyber and CRYSTALS-Dilithium algorithm. Encryption algorithms and post-quantum cryptography algorithms are often slow due to the complex mathematical operations they involve. With the rise of digitalization, cybersecurity has become even more critical. Therefore, even small devices with Reduced Instruction Set Computer (RISC) architectures may need to use complex cryptography algorithms. This project aims to extend the RISC-V instruction set architecture to enhance the performance of cryptography algorithms, focusing on the foundations of these cryptography schemes.

RISC-V is an open-source instruction set architecture that allows for easy addition of custom extensions, making it an ideal architecture for this project. Among various RISC-V cores, Ibex was chosen for its modular and understandable structure. Ibex implements the standard multiplication extension of RISC-V. One minor drawback of Ibex is that it does not come with a bus interface or any peripherals. To easily add and use peripherals, a modified project of Ibex with a Wishbone interface is used. This project includes a Wishbone bus interface module, allowing any Wishbone compatible peripheral module to be easily connected to the core. An example project top module for Field-Programmable Gate Array (FPGA) in the repository is used as a base to prepare a new one for the Nexys 4 DDR FPGA board. For debugging purposes and to familiarize with editing the project, several peripherals are added. These peripherals include General Purpose Input/Output (GPIO), timer, and Universal Asynchronous Receiver/Transmitter (UART) modules. GPIO and timer modules are written from scratch, while the UART module is sourced from an open-source repository. A C library is written for each peripheral to facilitate their use on the software side.

At this stage, if you cannot install any operating system the project needs to be synthesized and implemented again after each software modification. To avoid this delay, a bootloader program is written. This bootloader receives the application image file via UART, copies it to the Random Access Memory (RAM), and runs the program from RAM. This configuration significantly speeds up development.

Once the development environment is set up, the CRYSTALS-Kyber library is downloaded from the official website and tested on the device. Using static counters

and the GNU Debugger (GDB), the application is profiled to analyze which functions are used the most. Based on these analyses, a custom instruction for modular multiplication is added.

To add a custom extension, the Arithmetic Logic Unit (ALU) and the instruction decoder must be edited. Additionally, the compiler source code must be edited and rebuilt to inform the compiler about the custom instructions. Later, the custom instructions are used with inline assembly code inside the C code. The addition of the plantard modular reduction instruction to the Ibex RISC-V core reduced the modular multiplication function from 73 cycles to 37 cycles.

RISC-V ÇEKİRDEĞİNİN TALİMAT SETİNİN POST-KUANTUM KRİPTOGRAFİDE KULLANILAN SAYISAL TEORİK DÖNÜŞÜM İÇİN PLANTARD MODÜLER İNDİRGEME METODU İLE GENİŞLETİLMESİ

ÖZET

Kuantum bilgisayarlarının ortaya çıkışı, güvenliğin önemini vurgulamıştır. Geleneksel RSA gibi kriptografi algoritmaları, kuantum saldırılarına karşı giderek daha savunmasız hale gelmektedir. Bu nedenle, kuantum bilgisayarların tehdidine karşı koyabilen kriptografi algoritmalarına artan bir ihtiyaç vardır. Bu ihtiyaca yanıt olarak, Ulusal Standartlar ve Teknoloji Enstitüsü (NIST), sağlam post-kuantum kriptografi algoritmalarını standardize etmek için bir proje başlatmıştır. Seçilen algoritmalar arasında bu çalışma, CRYSTALS-Kyber ve CRYSTALS-Dilithium algoritmasına odaklanmaktadır. Şifreleme algoritmaları ve post-kuantum kriptografi algoritmaları karmaşık matematiksel işlemleri içerdiğinden dolayı genellikle yavaştır. Dijitalleşmenin artmasıyla, siber güvenlik daha da önemli hale gelmiştir. Bu nedenle, hatta Reduced Instruction Set Computer (RISC) mimarilerine sahip küçük cihazların bile karmaşık kriptografi algoritmalarını kullanması gerekebilir. Bu proje, RISC-V talimat seti mimarisini genişleterek kriptografi algoritmalarının performansını artırmayı amaçlamaktadır ve bu kriptografi şemalarının temellerine odaklanmaktadır.

RISC-V, özel uzantıların kolayca eklenmesine izin veren açık kaynaklı bir talimat seti mimarisidir, bu da onu bu proje için ideal bir mimari yapar. Çeşitli RISC-V çekirdekleri arasında, modüler ve anlaşılır yapısı nedeniyle Ibex tercih edilmiştir. Ibex, RISC-V'nin standart çarpma uzantısını uygular. Ibex'in küçük bir dezavantajı, bir veriyolu arayüzü veya herhangi bir çevre birimi ile gelmemesidir. Çevre birimlerini kolayca eklemek ve kullanmak için, Wishbone arayüzüne sahip bir Ibex projesi kullanılmıştır. Bu proje, Wishbone veriyolu arayüzü modülü içerir ve çekirdeğe kolayca bağlanabilir herhangi bir Wishbone uyumlu çevre birimi modülünü kullanmayı sağlar. Depoda bulunan FPGA için bir örnek proje modülü, Nexys 4 DDR FPGA kartı için yeni bir tane hazırlamak üzere bir temel olarak kullanılmıştır. Hata ayıklama amaçları ve projeyi düzenlemeye alışmak için birkaç çevre birimi eklenmiştir. Bu çevre birimleri Genel Amaçlı Giriş/Çıkış (GPIO), zamanlayıcı ve Evrensel Asenkron Alıcı/Verici (UART) modüllerini içerir. GPIO ve zamanlayıcı modülleri sıfırdan yazılmıştır, UART modülü ise açık kaynaklı bir depodan alınmıştır. Her çevre birimi için bir C kütüphanesi yazılmıştır, böylece yazılım tarafında kolayca kullanılabilirler.

Bu aşamada, eğer bir işletim sistemi yükleyemiyorsanız, proje her yazılım değişikliğinden sonra tekrar sentezlenmeli ve uygulanmalıdır. Bu gecikmeyi önlemek için bir önyükleme programı yazılmıştır. Bu önyükleme programı, uygulama görüntü dosyasını UART üzerinden alır, onu Rastgele Erişimli Bellek (RAM)'e kopyalar ve programı RAM'den çalıştırır. Bu yapılandırma, geliştirme hızını önemli ölçüde arttırmıştır.

Geliştirme ortamı kurulduktan sonra, CRYSTALS-Kyber kütüphanesi resmi web sitesinden indirilir ve cihazda test edilir. Statik sayaçlar ve GNU Debugger (GDB) kullanılarak, hangi fonksiyonların en çok kullanıldığını analiz etmek için uygulama profillenir. Bu analizlere dayanarak, modüler çarpma için özel bir talimat eklenir.

Özel bir uzantı eklemek için, Aritmetik Mantık Birimi (ALU) ve talimat çözücü düzenlenmelidir. Ayrıca, derleyici kaynak kodu düzenlenmeli ve yeniden oluşturulmalıdır, böylece derleyiciye özel talimatlar hakkında bilgi verilir. Daha sonra, özel talimatlar C kodu içindeki satır içi montaj kodu ile kullanılır. Ibex RISC-V çekirdeğine eklenen plantard modüler indirgeme talimatı, modüler çarpma fonksiyonunu 73 saatlik döngüden 37 saatlik döngüye düşürdü.

1. INTRODUCTION

This graduation project focuses on implementing an instruction set extension for RISC-V cores, specifically for modular arithmetic. The creation of this extension required modifications to the Arithmetic Logic Unit (ALU). Modular arithmetic plays a crucial role in cryptography algorithms, particularly in the multiplication of polynomials, which are vital for safeguarding personal and private data. The modular arithmetic discussed here is applicable to small primes, a topic addressed in both Post-Quantum Cryptography and Homomorphic Encryption schemes.

In the context of Post-Quantum Cryptography (PQC), Shor's and Grover's theoretical proofs suggest that traditional cryptography algorithms, such as the Rivest, Shamir, Adleman algorithm (RSA), could be easily compromised and become susceptible to attacks by quantum computers. [7,8].

The research is focused on developing an encryption algorithm robust enough to withstand attacks from both post-quantum and classical computing methods. This is why the National Institute of Standards and Technology (NIST) has initiated a project aimed at standardizing the encryption algorithm for post-quantum computational systems [9].

In this project, a post-quantum cryptography algorithm is analyzed and has been implemented on a RISC-V core. In order to implement RISC-V Core and Instruction Set Extension, Xilinx Vivado tools and Nexys 4 DDR Field-Programmable Gate Arrays (FPGA) are used. Also, experiments that are done on RISC-V compiled with GNU Toolchain (GCC).

1.1 Purpose of Thesis

The thought process in this thesis primarily involves the implementation of cryptographic algorithms and then identifying where these algorithms spend the most time and how that part can be optimized. It was noticed that in lattice-based

cryptographic algorithms, the application spends a lot of time in numerical theoretical transformation. Therefore, the question of how numerical theoretical transformation can be improved has arisen. Due to the use of modular arithmetic in numerical theoretical transformation, a literature review on modular arithmetic, which is the smallest building block in the algorithm, was conducted. Subsequently, the Plantard modular reduction method was chosen because it does not have any hardware implementation and performs well for small prime numbers. This method is expected to be efficient for post-quantum cryptographic algorithms, as smaller prime numbers are used compared to modern algorithms. The thesis also includes the addition of this method to the RISC-V core.

The optimization of modular arithmetic and its integration into the RISC-V instruction set is a crucial step forward in the realm of PQC algorithms. Modular arithmetic forms the backbone of many cryptographic algorithms, including those resistant to quantum attacks. By optimizing these operations, we can significantly enhance the efficiency and speed of PQC algorithms, making them more practical for real-world applications. Furthermore, incorporating these optimized operations into the RISC-V instruction set allows for more seamless and effective hardware implementations. This not only broadens the applicability of PQC algorithms but also paves the way for more secure, robust, and efficient cryptographic systems in the post-quantum era. Therefore, the importance of this endeavor cannot be overstated, as it stands at the forefront of our fight for privacy and security in the quantum computing age.

This research is centered on enhancing the performance of the Post-Quantum Cryptography (PQC) algorithms, specifically CRYSTALS-Kyber and CRYSTALS-Dilithium. It introduces additional instructions related to modular arithmetic, which can be applied to various algorithms. The goal is to improve the latency and instruction memory efficiency of the software versions of these algorithms. Profiling methods are used to pinpoint the operations that require the most clock cycles during algorithm execution. Based on this analysis, new custom instructions are developed to enhance performance. These instructions, tailored for specific cryptographic operations within the algorithms, expand the RISC-V Instruction Set Architecture (ISA).

Despite the absence of hardware implementations of Plantard modular multiplication or reduction until 2024, software implementations are available and referenced in [10]–[12]. The studies suggest that the use of Plantard arithmetic enhances both performance and code size. Moreover, in [12], an optimization is proposed that leverages the benefits of signed numbers, as Plantard originally uses unsigned values.

This study implements Plantard arithmetic on hardware and incorporates it into the RISC-V core for comparison with software counterparts.

2. MATHEMATICAL BACKGROUND

In the forthcoming section, we will delve into the practical applications of modular arithmetic, a fundamental mathematical concept, within the context of two significant cryptographic systems: CRYSTALS-Kyber and CRYSTALS-Dilithium. Through this detailed exploration, our aim is to provide a comprehensive understanding of the use cases of modular arithmetic in these cryptographic systems, shedding light on its importance in the field of cryptography.

2.1 CRYSTALS-Kyber Post-Quantum Cryptography Algorithm

CRYSTALS-Kyber [13] is a post-quantum cryptographic (PQC) algorithm that is a finalist in the NIST PQC competition. It is an IND-CCA2-secure key encapsulation mechanism (KEM) whose security is based on the hardness of solving the learning-with-errors (LWE) problem over module lattices.

The design of CRYSTALS-Kyber has its roots in the seminal LWE-based encryption scheme of Regev [14]. Since Regev’s original work, the practical efficiency of LWE encryption schemes has been improved by observing that the secret in LWE can come from the same distribution as the noise and also noticing that “LWE-like” schemes can be built by using a square (rather than a rectangular) matrix as the public key.

Another improvement was applying an idea originally used in the NTRU cryptosystem [15] to define the Ring-LWE and Module-LWE problems that used polynomial rings rather than integers. The CCA-secure KEM CRYSTALS-Kyber is built on top of a CPA-secure cryptosystem that is based on the hardness of Module-LWE.

CRYSTALS-Kyber comes in three security levels: CRYSTALS-Kyber-512, CRYSTALS-Kyber-768, and CRYSTALS-Kyber-1024, aiming at security roughly equivalent to AES-128, AES-192, and AES-256 respectively. The size vs. security tradeoffs are shown in the following table with RSA as a pre-quantum comparison:

Table 2.1 : Security Levels of CRYSTALS-Kyber Algorithm with Respect to AES

Version	Security Level	Private Key Size	Public Key Size	Ciphertext Size
Kyber512	AES128	1632	800	768
Kyber768	AES192	2400	1184	1088
Kyber1024	AES256	3168	1568	1568
RSA3072	AES128	384	384	384
RSA15360	AES256	1920	1920	1920

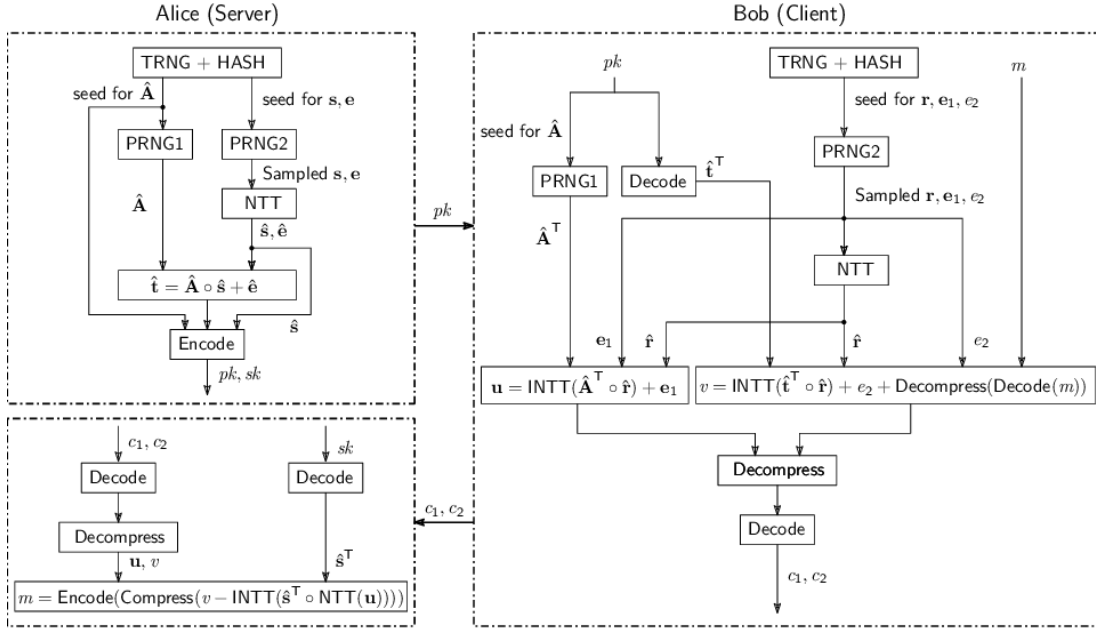


Figure 2.1 : CRYSTALS-Kyber PQC Algorithm Diagram [2]

2.2 CRYSTALS-Dilithium Post-Quantum Cryptography Algorithm

CRYSTALS-Dilithium [16] is a digital signature scheme offering robust security against chosen message attacks, grounded in the difficulty of lattice problems over module lattices. It stands as one of the algorithms proposed for the NIST post-quantum cryptography initiative. Dilithium's design draws inspiration from Lyubashevsky's "Fiat-Shamir with Aborts" technique, leveraging rejection sampling to enhance the compactness and security of lattice-based Fiat-Shamir schemes [17,18].

Dilithium surpasses the efficiency of prior schemes relying solely on the uniform distribution, such as the one by Bai and Galbraith. It achieves this by employing an

innovative approach that reduces the size of the public key by over half. As far as our understanding goes, Dilithium boasts the smallest combined size of public key and signature among lattice-based signature schemes utilizing only uniform sampling.

2.3 Polynomial Multiplication

Polynomial multiplication is a mathematical operation where two polynomials are multiplied together. The process involves multiplying each term of the first polynomial by each term of the second polynomial and then combining like terms.

In the context of cryptography, polynomial multiplication plays a crucial role, particularly in lattice-based cryptography (LBC). LBC has emerged as a viable substitute to classical cryptographic schemes, with 5 out of 7 finalist schemes in the 3rd round of the NIST post-quantum cryptography (PQC) standardization process being lattice-based [19].

The most critical and computationally intensive operation of Ring-LWE based cryptosystems, a type of LBC, is polynomial multiplication over rings. This operation is a bottleneck in every LBC construction. Efficient polynomial multiplication can lead to high-speed cryptographic systems, making them more practical for real-world applications [19,20].

Polynomial multiplication is a crucial operation employed in various cryptographic algorithms, especially in the context of safeguarding personal and private data. This operation becomes particularly significant in the use of small primes, a characteristic evident in both the CRYSTALS-Kyber and CRYSTALS-Dilithium post-quantum cryptography algorithms. In these algorithms, the employment of small primes in polynomial multiplication is deliberate, aligning with the broader goals of Post-Quantum Cryptography and Homomorphic Encryption schemes. The use of small primes is a strategic choice, contributing to the robustness of these cryptographic schemes against potential threats from both quantum and classical computing methods. As a result, polynomial multiplication with small primes stands out as a fundamental and carefully considered operation in modern cryptography, playing a pivotal role in enhancing the security of cryptographic systems.

2.3.1 Schoolbook Polynomial Multiplication (SPM)

Schoolbook Polynomial Multiplication (SPM) is a fundamental mathematical operation where two polynomials, represented as $a(x)$ and $b(x)$, are multiplied together. The process involves multiplying each term of the first polynomial by each term of the second polynomial and then combining like terms. The multiplication operation is expressed as follows:

$$a(x) \times b(x) = \sum_{i=0}^{n-1} \sum_{j=0}^{n-1} a_i b_j x^{i+j} \mod (x^n + 1) \quad (2.1)$$

In this equation, a_i and b_j represent the coefficients of the respective polynomials, and n is the degree of the polynomials. The modulus $(x^n + 1)$ ensures that the result remains within the polynomial ring.

2.3.2 Number Theoretic Transform (NTT)

The Number Theoretic Transform (NTT) is a generalization of the Fast Fourier Transform (FFT) that operates over a quotient ring instead of the complex numbers. It replaces the complex n th root of unity in the FFT with an n th primitive root of unity.

NTT is obtained by replacing $e^{(-2\pi i k/N)}$ with an n th primitive root of unity. This effectively means doing a transform over the quotient ring \mathbb{Z}/\mathbb{Z}_q instead of the complex numbers \mathbb{C} . The NTT has become increasingly important in developing PQC and HE. Its ability to efficiently calculate polynomial multiplication using the convolution theorem with a quasi-linear complexity $O(n \log n)$ when implemented with Fast Fourier Transform-style algorithms has made it a key component in modern cryptography.

The FFT-style NTT algorithm or fast-NTT is particularly useful in lattice-based cryptography, which relies on the hardness of certain mathematical problems to ensure security. Its importance in these fields continues to grow as quantum computing technology advances and traditional encryption methods become vulnerable [21].

Most computationally intensive operation is polynomial multiplication. This operation can be performed using either the schoolbook polynomial multiplication algorithm or the Number Theoretic Transform (NTT). The latter is particularly efficient for

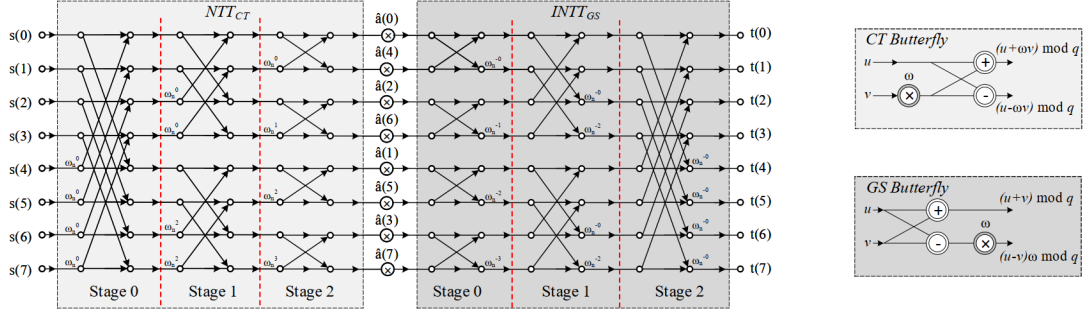


Figure 2.2 : An 8-point NTT-based polynomial multiplication [3]

computing polynomial multiplication over a polynomial ring $\mathbb{Z}_q[X]/\langle X^n + 1 \rangle$. The NTT is a generalization of the Fast Fourier Transform (FFT), but it is defined in a finite field [22]. Suppose we have a polynomial f of degree n , where

$$f = \sum_{i=0}^{n-1} f_i X^i \quad (2.2)$$

and $f_i \in \mathbb{Z}_q$. Let ω_n be an n th primitive root of unity such that

$$\omega_n^n = 1 \mod q \quad (2.3)$$

The forward NTT is defined as

$$\hat{f} = NTT(f) \quad (2.4)$$

, where

$$\hat{f}_i = \sum_{j=0}^{n-1} f_j \omega_n^{ij} \mod q \quad (2.5)$$

The inverse NTT is given by

$$f = INTT(\hat{f}) \quad (2.6)$$

, where

$$f_i = n^{-1} \sum_{j=0}^{n-1} \hat{f}_j \omega_n^{-ij} \mod q \quad (2.7)$$

Using the NTT, we can perform polynomial multiplication between f and g as follows:

$$f \cdot g = INTT(NTT(f)) \quad (2.8)$$

This formula represents the convolution of the two polynomials.

Algorithm 1 Number Theoretic Transform (NTT)

Require: Polynomial $A(x) \in \mathbb{Z}_q[x]$, $A(x) = a[k-1]x^{k-1} + a[k-2]x^{k-2} + \dots + a[0]$

Require: Primitive k -th root of unity $\omega \in \mathbb{Z}_q$

Require: $q \equiv 1 \pmod{2n}$,

Ensure: $NTT(A(x))$

```
1:  $A \leftarrow \text{br}_7(A)$ 
2: for  $s = 1$  to  $\log_2(k)$  do
3:    $m \leftarrow 2^s$ 
4:    $\omega_m \leftarrow \omega^{\frac{k}{m}}$ 
5:   for  $k = 0$  to  $n - 1$  step  $m$  do
6:      $\omega_k \leftarrow 1$ 
7:     for  $j = 0$  to  $\frac{m}{2} - 1$  do
8:        $t \leftarrow \omega_k \cdot A[k + j + \frac{m}{2}] \pmod{p}$ 
9:        $u \leftarrow A[k + j]$ 
10:       $A[k + j] \leftarrow u + t \pmod{q}$ 
11:       $A[k + j + \frac{m}{2}] \leftarrow u - t \pmod{q}$ 
12:       $\omega_k \leftarrow \omega_k \cdot \omega_m \pmod{q}$ 
13:     end for
14:   end for
15: end for
```

To carry out point-wise multiplication in CRYSTALS-Kyber, we need to compute 128 polynomial multiplications of degree-2. This can be represented as

$$(\hat{a}_{j,2i} + \hat{a}_{j,2i+1}X) \cdot (\hat{s}_{2i} + \hat{s}_{2i+1}X) = (\hat{a}_{j,2i}\hat{s}_{2i} + \hat{a}_{j,2i+1}\hat{s}_{2i+1}\omega_n^{2br_7(i)+1}) + (\hat{a}_{j,2i}\hat{s}_{2i+1} + \hat{a}_{j,2i+1}\hat{s}_{2i})X \quad (2.9)$$

, where br_7 represents bit reversal function. This bit reversal function rearranges the order of coefficients in the polynomial.

2.3.2.1 Comparison of NTT and SPM

The Number Theoretic Transform (NTT) is a powerful tool in modern cryptography that provides an efficient method for polynomial multiplication. In contrast, the schoolbook method for polynomial multiplication, which involves multiplying each coefficient of the first polynomial with each coefficient of the second polynomial, has a time complexity of $O(n^2)$. This can be computationally expensive, especially for

large polynomials. The NTT, however, can reduce this computational complexity to roughly $O(n \log n)$. This is achieved by transforming the polynomials into a different domain (the “frequency domain”) where the multiplication of polynomials becomes a simple point-wise multiplication. After the multiplication, an inverse transform is applied to get the result back in the original domain. Furthermore, the NTT is particularly well-suited for hardware implementations, which can further speed up the computation. For instance, the use of the NTT in the CRYSTALS-Kyber algorithm, a finalist in the NIST post-quantum cryptography standardization process, has led to significant improvements in efficiency.

In conclusion, the NTT is faster than the schoolbook method for polynomial multiplication due to its lower computational complexity and its amenability to efficient hardware implementations [3].

2.4 Modular Reduction

Modular reduction is a fundamental operation in modular arithmetic, which is a branch of arithmetic mathematics associated with the “mod” operation. Given two integers a and b , and a positive integer m modulus, modular reduction of a times b with the modulo m , is defined as follows

$$(a \cdot b) \mod m = (a \mod m \cdot b \mod m) \mod m \quad (2.10)$$

2.4.1 Barret Modular Reduction

The result of this operation, as described in Equation 2.10, represents the remainder of the multiplication of a and b divided by m . Modular multiplication is used extensively in computer science and cryptography. For example, Barrett modular multiplication is a method introduced by P.D. Barrett [23] in 1986. It is a reduction algorithm designed to optimize modular multiplication operations, replacing divisions with multiplications. This method is particularly beneficial in scenarios where the modulus is constant.

Algorithm 2 Barrett Modular Reduction Algorithm

Require: A, P, R, n with $P < 2^n, 0 \leq A \leq (P-1)^2, R = \lfloor \frac{2^{2n}}{P} \rfloor$

Ensure: C with $0 \leq C < P, C = A \bmod P$

- 1: $C \leftarrow A - (\lfloor (\lfloor \frac{A}{2^{n-1}} \rfloor) \cdot R \rfloor / 2^{n+1})$
- 2: $C \leftarrow C \cdot P$
- 3: **if** $C \geq 2 \cdot P$ **then**
- 4: $C \leftarrow C - 2 \cdot P$
- 5: **end if**
- 6: **if** $C \geq P$ **then**
- 7: $C \leftarrow C - P$
- 8: **end if**

* Notation $\lfloor \frac{x}{y} \rfloor$ is integer part of division

Instead of performing the division operation, which can be computationally expensive, Barrett reduction approximates the division with a multiplication and a right shift operation as seen in Algorithm 2. This approximation is then used to calculate the modular multiplication.

The Barrett modular multiplication method is widely used in cryptography, especially in public-key cryptosystems. It provides a more efficient way to perform modular multiplication, which is a fundamental operation in many cryptographic algorithms.

2.4.2 K²Red Modular Reduction [1]

K²RED, a unique modular reduction algorithm, is designed to work specifically with a subset of prime numbers known as Proth numbers [24]. This algorithm was introduced by Niasar et al. in [24]. A Proth number has a specific form, $q = k \cdot 2^m + 1$, where $2^m > k$ and k is an odd number.

The K²RED algorithm, shown in Algorithm 3, is a variant of the Montgomery Algorithm presented in [24]. It is fundamentally based on the KRED algorithm proposed in [24], and applies the KRED algorithm twice. The implementation of K²RED involves an adder and a shifter.

In the context of the CRYSTALS-Kyber algorithm, the parameters are set as $P = 3329$, $m = 8$, and $k = 13$. For the CRYSTALS-Dilithium algorithm [16], the parameters are

$P = 8380417$, $m = 13$, and $k = 1023$. In the case of HE, the parameters can vary in bit length. An example selection is $P = 2146959361$, $m = 19$, and $k = 4095$.

For some primes in HE, steps 7 and 8 of the algorithm are necessary. This is because the C_h value can be very large when the value of m is small, resulting in the output value exceeding the prime value when subtracted from C_l .

Algorithm 3 K²RED Modular Reduction Algorithm

Require: A, P, n with $P < 2^n$, $P = k \cdot 2^m + 1$, k is odd, $k < 2^m$, $0 \leq A < 2^{2n}$

Ensure: $C' = k^2 \cdot A \mod P$

Step 1:

- 1: $A_l \leftarrow (a_m, a_{m-1}, \dots, a_1, a_0)_2$
- 2: $A_h \leftarrow (a_{2n-1}, \dots, a_{m+1})_2$
- 3: $C \leftarrow k \cdot A_l - A_h$

Step 2:

- 4: $C_l \leftarrow (c_m, c_{m-1}, \dots, c_1, c_0)_2$
 - 5: $C_h \leftarrow (c_t, \dots, c_{m+1})_2$
 - 6: $C' \leftarrow k \cdot C_l - C_h$
 - 7: **if** $C' > P$ **then**
 - 8: $C' \leftarrow C' - P$
 - 9: **end if**
-

The bit length, denoted as t , differs from the bit length obtained in the previous operation. While these values are fixed in the CRYSTALS-Kyber and CRYSTALS-Dilithium algorithms, this bit length can change in HE depending on the prime values used.

2.4.3 Plantard Modular Reduction

Thomas Plantard proposed Plantard reduction algorithm for modulo operations where prime number is smaller than it is used in RSA, ECC etc. in 2021 [25]. The key advantage of Plantard arithmetic is that it can save one multiplication operation from the modular multiplication of a constant. This can lead to significant performance improvements in cryptographic algorithms where modular multiplication is a common operation [26].

Algorithm 4 Plantard Modular Reduction Algorithm

Require: $P, \Phi = \frac{1+\sqrt{5}}{2}, P < \frac{2^n}{\Phi}, R = P^{-1} \bmod 2^{2n}, A = X \cdot Y \cdot R \bmod 2^{2n}$

Ensure: C with $0 \leq C < P, C = X \cdot Y(-2^{-2n}) \bmod P$

- 1: $C \leftarrow (\lfloor \frac{A}{2^n} \rfloor) + 1$
 - 2: $C \leftarrow \lfloor \frac{CP}{2^n} \rfloor$
 - 3: **if** $C = P$ **then**
 - 4: $C \leftarrow 0$
 - 5: **end if**
-

In the Plantard modular reduction algorithm, the algorithm needs the input integer A , modulus P , and inverse modulus $P^{-1} \bmod 2^{2n}$. Plantard modular reduction algorithm that as every algorithm has its advantages for specific primes, most of the algorithms study for bigger primes ($n \gg 64$). With this method, as it is seen in Algorithm 4 is able to compute modulo operation with smaller primes with one multiply-add operation. Here one little thing is that n is not equal bit size of prime number but it is $P < \frac{2^n}{\Phi}$ shown in Algorithm 4.

3. HARDWARE IMPLEMENTATION OF MODULAR ARITHMETIC ALGORITHMS

Modular arithmetic involves operations performed within a specific modulus or modulus space. The fundamental operations in modular arithmetic are:

1. **Modular Addition:** Adding two numbers and taking the remainder when divided by a modulus.

$$(a + b) \mod m$$

2. **Modular Subtraction:** Subtracting one number from another and taking the remainder when divided by a modulus.

$$(a - b) \mod m$$

3. **Modular Multiplication:** Multiplying two numbers and then taking the remainder when divided by a modulus.

$$(a \times b) \mod m$$

In our research, our goal is to expedite the NTT operation, requiring certain mathematical operations. Specifically, modular subtraction and addition, which are relatively straightforward compared to modular multiplication, and modular addition and modular subtraction can be efficiently executed using comparators. So we compare modular multiplication and modular reduction methods.

3.1 Barret Modular Reduction Algorithm

Barrett modular reduction algorithm shown in Algorithm 2 is implemented on an FPGA and block diagram of implementation can be seen in Figure 3.1. This algorithm is particularly useful in cryptographic applications, where the modulus is often a fixed value as it is in PQC. By replacing the slow division operation with

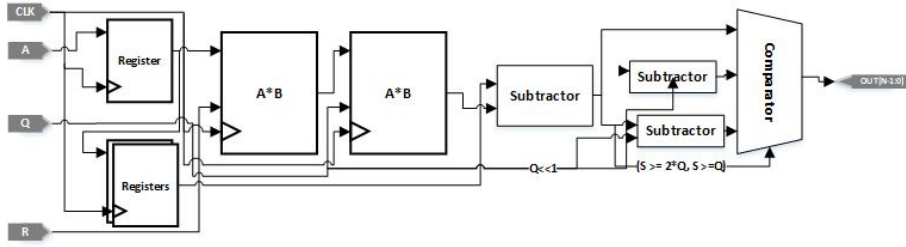


Figure 3.1 : Barrett Modular Reduction Block Diagram

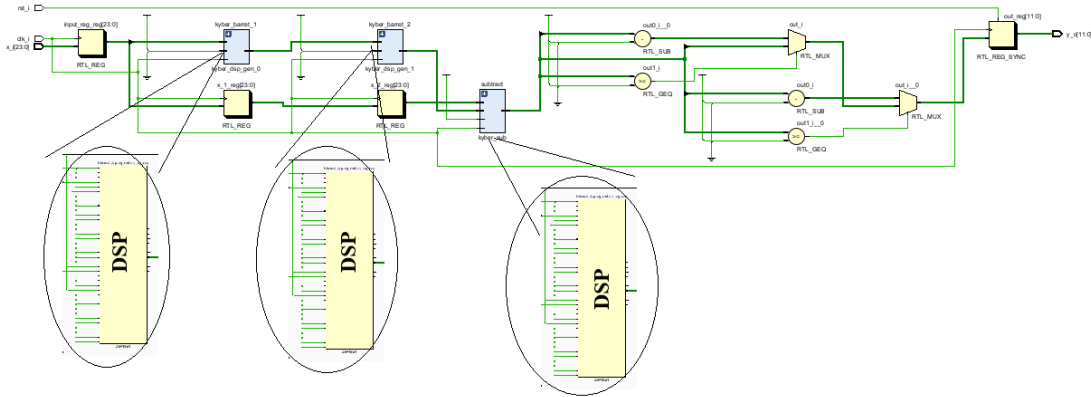


Figure 3.2 : RTL Schematic Diagram of the Design for Barrett Modular Reduction Algorithm for Kyber

faster multiplications and shifts, Barrett reduction can significantly speed up modular reductions.

In this diagram both 3329, 8380417 and 2146959361 used as a prime number used by CRYSTALS-Kyber, CRYSTALS-Dilithium and Homomorphic Encryption cases respectively.

Resulting RTL that generated on Xilinx Vivado, shows the same design at the top level, but as bit sizes increased, for example 12 bits in CRYSTALS-Kyber, multiplication operations cannot be done with one DSP since DSP uses 18×25 bit sizes for A and B. For example as for CRYSTALS-Dilithium it uses 2 DSPs per one multiplication since 24×24 is needed.

3.2 K^2 red Modular Reduction

As seen in Figure 3.3 and in Algorithm 3, multiplication operations in the designed scheme and algorithm uses for multiplication operation with smaller sizes and

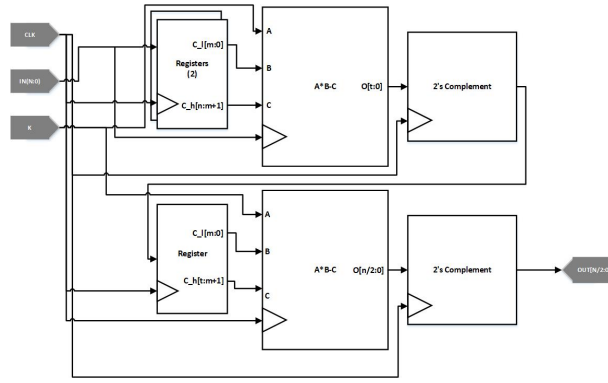


Figure 3.3 : K^2 red Modular Reduction Block Diagram

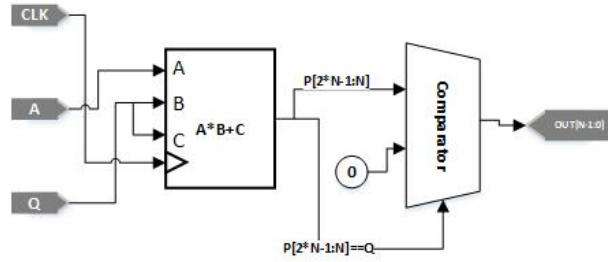


Figure 3.4 : Plantard Modular Reduction Block Diagram

multiplications in this algorithm can be optimized use cases of PQC since primes are constant. However, it cannot be generalized or optimized for non-constant primes.

3.3 Plantard Modular Reduction

As seen Figure 3.4 and also Algorithm 4, Plantard modular reduction algorithm simply uses one adder, one multiplier and one comparator with 0. As in the Algorithm 4, as we have selected primes that used in the cryptography methods, we simply need to choose “ n ” to be slightly larger than bit sizes selected primes. Therefore, for CRYSTALS-Kyber PQC algorithm we can select the “ n ” to be minimum 13 bits, for Dilithium PQC minimum 24 bits and as for the Homomorphic encryption as primes, P can be selected as $0 < P < 2^n$ if our selected prime is smaller than 2^{n-1} , our implementation can select 32 bits as “ n ”.

I have designed a hardware module for the implementation of the block diagram shown in Fig. 3.4 and implemented on and FPGA using Xilinx MultAdd IP [27] and this IP selects the optimized configuration of DSPs created inside the IP.

Implementation result of Plantard modular reduction, on selected FPGA can be seen in Table 5.1 and 5.2. As seen in Figure 3.5, elaborated RTL is the same with the block

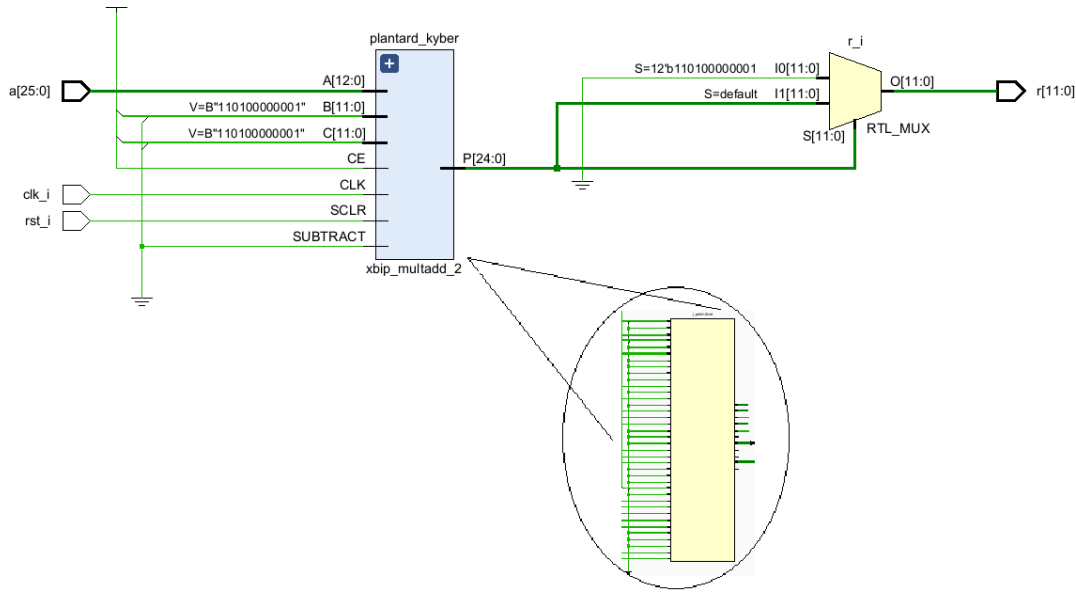


Figure 3.5 : Plantard Modular Reduction CRYSTALS-Kyber RTL Diagram

diagram shown in Figure 3.4 but since CRYSTALS-Kyber, CRYSTALS-Dilithium and Homomorphic cryptography need different sizes of primes, there is a difference of numbers of DSPs that is used for $A * B + C$ operation in plantard modular reduction algorithm. Number of DSPs that is used when implementing $A * B + C$ operation can be seen from Table 5.1.

3.4 Plantard Modular Multiplication

This method includes multiplication before modulo operation therefore operation includes precomputing twiddle factors in NTT. Algorithm 4 turns into below algorithm when multiplication is added.

Also comparisons of k^2 red and Plantard modular multiplication can also be seen in Figures 3.6 and 3.7. As our purpose in this study is integration these modular reduction or multiplication algorithms in RISC-V core and as for smaller operating frequency Plantard modular reduction has more efficiency than k^2 red modular multiplication, this study preferred Plantard modular reduction algorithm to be integrated in RISC-V core.

Algorithm 5 Plantard Modular Multiplication Algorithm

Require: $P, \Phi = \frac{1+\sqrt{5}}{2}, P < \frac{2^n}{\Phi}, R = P^{-1} \bmod 2^{2n}$

Ensure: C with $0 \leq C < P, C = X \cdot Y(-2^{-2n}) \bmod P$

- 1: $A \leftarrow X \times Y \bmod P$
 - 2: $C \leftarrow (\lfloor \frac{A}{2^n} \rfloor) + 1$
 - 3: $C \leftarrow \lfloor \frac{C \cdot P}{2^n} \rfloor$
 - 4: **if** $C = P$ **then**
 - 5: $C \leftarrow 0$
 - 6: **end if**
-

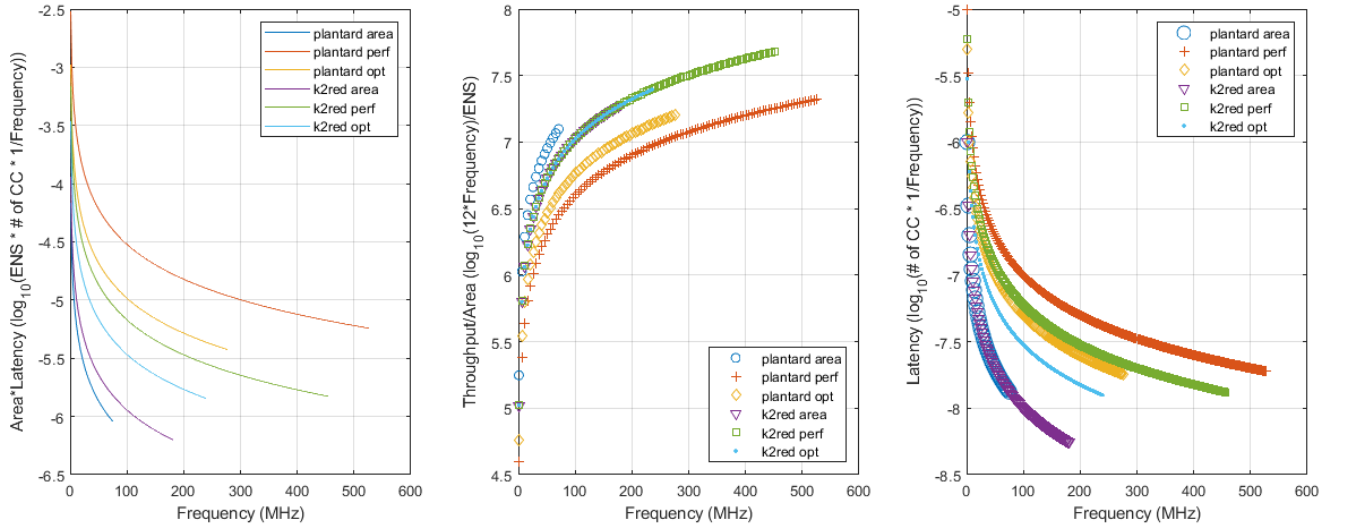


Figure 3.6 : Comparisons of K²RED and Plantard modular multiplication for CRYSTALS-Kyber [4]

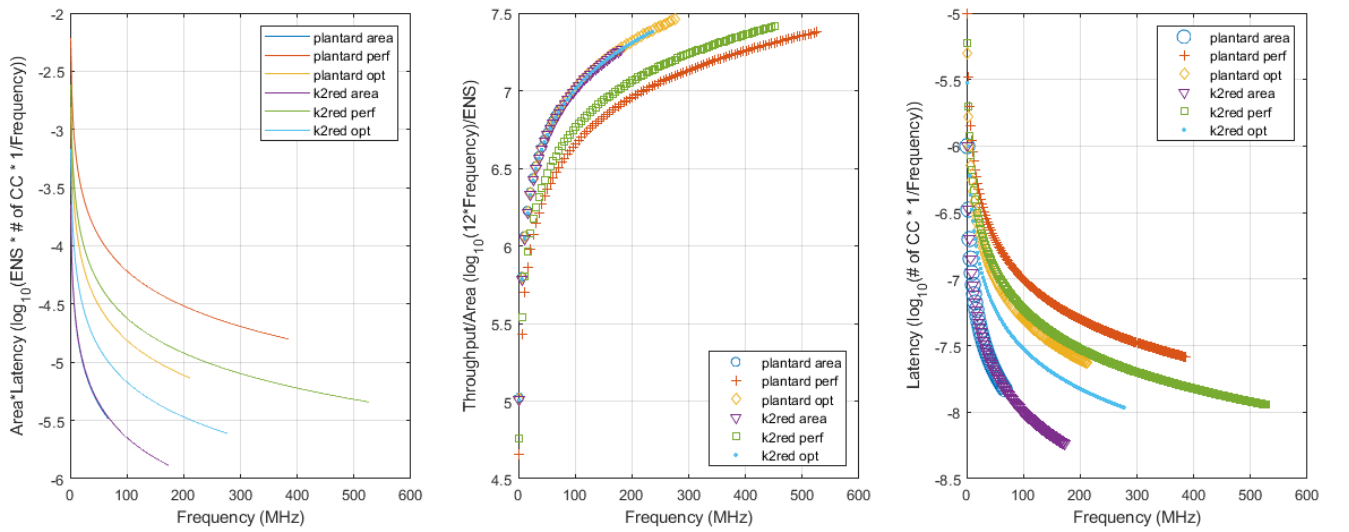


Figure 3.7 : Comparisons of K²RED and Plantard modular multiplication for CRYSTALS-Dilithium [4]

4. RISC-V

RISC-V [28] is an open-source hardware instruction set architecture (ISA) [29] based on Reduced Instruction Set Computer (RISC) principles. Unlike proprietary ISAs, RISC-V permits anyone to create, manufacture, and sell RISC-V chips and software under a permissive license, fostering innovation and reducing costs.

Initiated as a student project at UC Berkeley in 2005, RISC-V was conceived with the goal of streamlining processor design. In contrast to the complex and monolithic processors of that era, RISC-V introduced a streamlined and modular methodology, reminiscent of constructing custom CPUs using Lego-like building blocks. This adaptability and openness sparked interest, drawing industry titans such as Google, Samsung, and SiFive, thereby nurturing a dynamic and thriving ecosystem.

RISC-V's modular design allows for a wide range of applications, from tiny embedded systems to large-scale data centers. Its simplicity and efficiency make it ideal for modern computing needs. The ISA includes only the instructions that are frequently used in general-purpose computing, reducing the complexity of the processor design.

RISC-V International, the global non-profit home of the open standard RISC-V ISA, related specifications, and stakeholder community, contributes and collaborates to define RISC-V open specifications. As of June 2019, version 2.2 of the user-space ISA and version 1.11 of the privileged ISA are frozen, permitting software and hardware development to proceed.

The open nature of RISC-V encourages a collaborative approach to development, leading to rapid advancements and improvements. It also allows for customization, enabling designers to add their own instructions to meet specific needs. This is particularly beneficial in emerging fields like AI and machine learning, where specialized hardware can significantly improve performance.

RISC-V's impact extends beyond technology. By democratizing access to custom silicon, RISC-V could lead to a new era of innovation and competition in the semiconductor industry. As such, RISC-V is not just an ISA, but a catalyst for change in computing.

4.1 RISC-V ISA

The RISC-V ISA [29] is modular, meaning it has a small set of simple base instructions, with optional extensions for more complex operations. This modularity allows for a high degree of customization, making RISC-V suitable for a wide range of applications, from tiny embedded systems to large-scale data centers.

Base ISA	Instructions	Description
RV32I	47	32-bit address space and integer instructions
RV32E	47	Subset of RV32I, restricted to 16 registers
RV64I	59	64-bit address space and integer instructions, along with several 32-bit integer instructions
RV128I	71	128-bit address space and integer instructions, along with several 64- and 32-bit instructions
Extension	Instructions	Description
M	8	Integer multiply and divide
A	11	Atomic memory operations, load-reserve/store conditional
F	26	Single-precision (32 bit) floating point
D	26	Double-precision (64 bit) floating point; requires F extension
Q	26	Quad-precision (128 bit) floating point; requires F and D extensions
C	46	Compressed integer instructions; reduces size to 16 bits

Figure 4.1 : RISC-V Instruction Sets and Extension [5].

The RISC-V architecture is designed with flexibility in mind, offering 32 general-purpose registers (GPRs), 32 floating-point registers (FPRs), and 32 privileged control registers (PCRs). The width of these registers can vary, depending on their specific purpose and the system's requirements. This adaptability allows for a wide range of applications, from embedded systems to high-performance computing.

The base integer (I) and machine-level privileged Instruction Set Architectures (ISAs) encompass all the necessary functionalities of a rudimentary, general-purpose Central Processing Unit (CPU). Developers have the liberty to enhance this fundamental

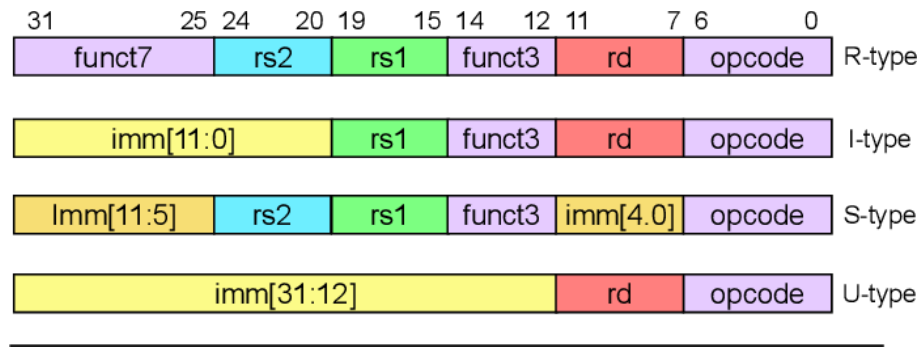


Figure 4.2 : RISC-V Base Instruction Formats [5].

capability by incorporating extensions to the ISA. While custom extensions are feasible, the RISC-V Foundation’s technical task groups manage standard extensions. These standard extensions are deemed to have widespread appeal to the design community and their instructions do not interfere with other standard extensions. Consequently, developers can incorporate any of the standard extensions they need in their design without worrying about conflicts in instruction coding. The standard extensions include:

- M — Instructions for multiplying and dividing values held in two integer registers (frozen)
- A — Instructions for atomically reading/modifying/writing memory to support synchronization (frozen)
- F, D, and Q — Instructions for single- (F), double- (D), and quad-precision (Q) floating-point computations compliant with the IEEE 754-2008 arithmetic standard. Each precision’s extension depends on the lower-precision extension being present (frozen).
- G — An implementation that includes the base integer specification (I) along with the M, A, F, and D standard extensions is so popular that the Foundation has defined the collection as G and has set the G configuration as the standard target of compiler toolchains under development (frozen).
- V — Instructions to add vector instructions to the floating-point extensions (draft)
- L — Instructions for decimal floating-point calculations (reserved)

- B — Instructions for bit-level manipulation (reserved)
- N — Instructions that handle user-level interrupts (draft)
- P — An extension to support packed single-instruction, multiple-data instructions (draft)
- T — Instructions to support transactional memory operations (reserved)
- J — An extension to support use of dynamically translated languages (reserved)
- C — Support for compressed instruction execution. The base integer (I) specification calls for instruction words to be 32 bits long and aligned on 32-bit boundaries in memory. Implementing the C standard extension provides 16-bit encodings of common operations and allows CPU designs to work with alignments on freely mixed 32- and 16-bit boundaries, resulting in a 25% to 30% reduction in code size. It can be implemented in any of the base integer bit widths and with any of the other standard extensions (frozen).
- S — The privileged ISA’s supervisor-level extension (draft)

Furthermore, the RISC-V Foundation’s committee has established design specifications that empower developers to create extensions to the instruction sets. This extensibility is one of the key strengths of RISC-V, enabling it to adapt to emerging needs and technologies. For instance, developers can add custom instructions to accelerate specific workloads, or create new extensions to support innovative hardware features [29].

This open and flexible approach has contributed to the growing popularity of RISC-V in both academia and industry, as it allows for a high degree of customization while maintaining compatibility with existing software ecosystems. As such, RISC-V is not just an ISA, but a framework for innovation in processor design.

4.2 RISC-V GNU Toolchain Collection

The RISC-V GNU toolchain, a critical component in the RISC-V ecosystem. The toolchain has evolved significantly over the years, thanks to the contributions from the expanding RISC-V community. It supports two build modes: a generic ELF/Newlib toolchain and a more sophisticated Linux-ELF/glibc toolchain. The development and refinement of the toolchain have been instrumental in facilitating the adoption and proliferation of RISC-V solutions in the market. Today, the RISC-V GNU toolchain stands as a testament to the power of open-source collaboration in advancing computing technology [30].

4.2.1 Building the RISC-V GCC Toolchain

To build the RISC-V GCC Toolchain on an Ubuntu computer, start by cloning the RISC-V GNU Toolchain repository. The tools and versions used are listed in Table 4.1, and Ubuntu 22.04.3 LTS was selected as the operating system [31]. Vivado 2022.2 is used for FPGA development [32].

Table 4.1 : Used Environment Settings

Ubuntu	22.04.3 LTS
CMake [33]	v3.22.1
GCC	v11.4.0
Python	v3.10.12
Vivado	v2022.2

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

After cloning the repository, install the required packages:

```
$ sudo apt-get install autoconf automake autotools-dev curl
python3 python3-pip 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
```

Navigate to the cloned repository:

```
$ cd riscv-gnu-toolchain
```

Configure the compiler, specifying the architecture and installation folder:

```
$ ./configure --prefix=/opt/riscv --with-arch=rv32imc_zicsr  
$ make
```

Add the installed compiler path to Ubuntu environment variables by adding the following line to **.bashrc**:

```
export PATH="/opt/riscv/bin/:$PATH"
```

After installation, a simple "Hello, World!" application can be compiled and executed:

```
$ riscv32-unknown-elf-gcc hello.c -o hello.o
```

4.3 RISC-V Core Implementation

Upon evaluating various RISC-V cores, we've chosen to proceed with Ibex due to its comprehensible and modifiable source code. Ibex supports the RV32IMC instruction set, which encompasses the 32-bit base integer set (I), the standard multiplication extension (M), and the compressed instructions extension (C). However, this project does not utilize compressed instructions. By default, Ibex lacks a bus interface with peripherals, offering only connections to data memory and instruction memory. To facilitate the addition of peripherals and enable FPGA testing, i am using and updating a version of Ibex equipped with a wishbone bus, known as `ibex_wb` [34]. This version includes a sample implementation for the Xilinx Arty A7-100: Artix-7 FPGA [35] development board, which serves as our foundation for this project.

4.3.1 Microcontroller Structure based Ibex RISC-V in Core

A microcontroller, sometimes referred to as an MCU (Microcontroller Unit), is a small computer on a single integrated circuit. It is designed to control specific tasks within electronic systems. A microcontroller contains one or more CPUs (processor cores) along with memory and programmable input/output peripherals.

Microcontrollers are used in automatically controlled products and devices, such as automobile engine control systems, implantable medical devices, remote controls,

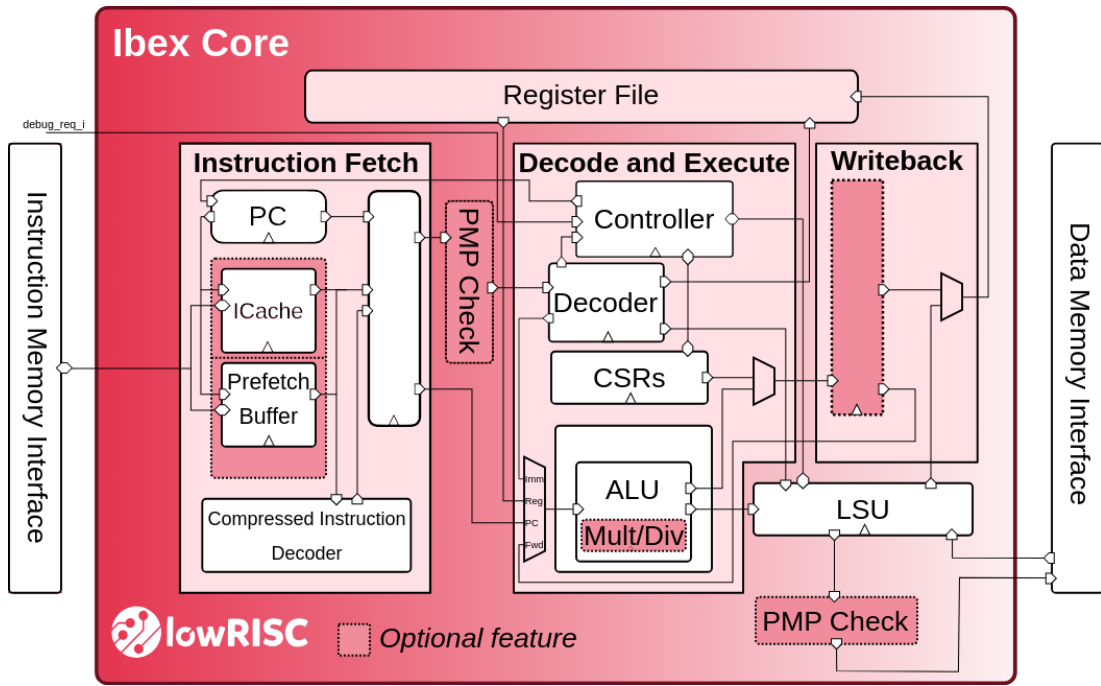


Figure 4.3 : Ibex RISC-V Core [6]

office machines, appliances, power tools, toys, and other embedded systems. By reducing the size and cost compared to a design that uses a separate microprocessor, memory, and input/output devices, microcontrollers make it economical to digitally control even more devices and processes.

The Ibex core, along with the UART, Timer, and GPIO peripherals, can be synthesized onto the Artix-7 FPGA. The resulting microcontroller can then be programmed using RISC-V compatible software tools, enabling the development of efficient, compact, and powerful embedded systems.

In conclusion, the combination of the Ibex RISC-V core with UART, Timer, and GPIO peripherals on an Artix-7 FPGA forms a powerful and flexible microcontroller system. This system can be tailored to a wide range of applications, making it a versatile solution for embedded system design.

4.4 Customization of RISC-V Core and GCC

Different implementations of Plantard modular multiplication and modular reduction are explored, with a specific focus on customization for CRYSTALS-Kyber. The design of Plantard modular reduction for CRYSTALS-Kyber is depicted in Figure ??.

4.4.1 RISC-V GNU Toolchain Modification

Adding a custom instruction to the RISC-V GNU toolchain involves several steps. First, you need to define the opcode for your custom instruction. This is typically done in the `riscv/opcodes.h` file. The opcode is a unique identifier that the processor uses to recognize your instruction.

Next, you need to modify the toolchain to recognize your new instruction. This involves editing the `riscv-gnu-toolchain/riscv-binutils/opcodes/riscv-opc.c` file to include your new instruction. This file contains the definitions of all the instructions that the toolchain recognizes.

After you've made these changes, you need to recompile the toolchain. This process builds a new version of the toolchain that includes your custom instruction. You can then use this modified toolchain to compile programs that use your new instruction. Modification that are made in the source codes of toolchain can be seen in Figures 4.4, 4.5 and 4.6.

```

/* custom instruction extension */
/* single cycle */
{"pltrd_sc", 0, INSN_CLASS_I, "d,s,t",
 → MATCH_PLANTARD_SC, MASK_PLANTARD_SC, match_opcode, 0 },
/* multi cycle */
{"pltrd_mc", 0, INSN_CLASS_ZMMUL, "d,s,t",
 → MATCH_PLANTARD_MC, MASK_PLANTARD_MC, match_opcode, 0 },

```

Figure 4.4 : RISC-V GNU Toolchain "riscv-opc.c" Modification for Custom Instruction

```

/* Instruction Set Extension */
DECLARE_INSN(pltrd_sc, MATCH_PLANTARD_SC, MASK_PLANTARD_SC)
DECLARE_INSN(pltrd_mc, MATCH_PLANTARD_MC, MASK_PLANTARD_MC)
/*-----*/

```

Figure 4.5 : Declaration of Custom Instruction in the RISC-V GNU Toolchain "riscv-opc.h"

```
/* Instruction extensions. */
#define MATCH_PLANTARD_SC 0x70000033
#define MASK_PLANTARD_SC  0xfe00707f
#define MATCH_PLANTARD_MC 0x40000033
#define MASK_PLANTARD_MC  0xfe00707f
```

Figure 4.6 : RISC-V GNU Toolchain "riscv-opc.h" Modification for Custom Instruction

4.4.2 Adding New Instruction to RISC-V Core

Adding a new instruction is discussed in this subsection, and it is beneficial for several reasons as pointed out below.

- **Performance Optimization:** Custom instructions can be designed to perform specific tasks more efficiently than a sequence of standard instructions, potentially leading to significant performance improvements.
- **Hardware Acceleration:** If a particular operation is a performance bottleneck and is executed frequently, implementing it as a custom instruction can offload the computation to hardware, speeding up the overall execution.
- **Support for Domain-Specific Applications:** Custom instructions can be added to support specific applications or domains. For example, cryptographic operations, signal processing tasks, or machine learning algorithms can benefit from custom instructions.
- **Educational Purposes:** Adding custom instructions can be a valuable learning experience, providing insights into processor design, instruction set architectures, and hardware-software trade-offs.

However, it's important to note that adding custom instructions increases the complexity of the processor design and can lead to compatibility issues with existing software tools and libraries. Therefore, the decision to add custom instructions should be made carefully, considering the trade-offs involved.

New custom instructions for Plantard modular reductions have been added to both the 'I' instruction set and the 'M' instruction set. The single-cycle operation is included in

the 'I' instruction set, while the multi-cycle operation is added to the 'M' instruction set. The latter is particularly beneficial as it is closely associated with the 'REM' instruction, covering multiplication operations.

4.4.2.1 Single Cycle Instruction Addition

In order to add a single cycle instruction, as RISC-V core does not need to stall I can add new RTL to ALU. In Figure 4.7 added custom block to implement single cycle modular reduction can be seen.

```

module plntrd_sc (
    input    logic [31:0] a_i,    // a
    input    logic [31:0] b_i,    // q

    output   logic [31:0] c_o
);

    logic [47:0] mult;
    logic [15:0] r1;
    logic [15:0] r2;
    logic [31:0] r3;
    logic [15:0] r4;

    mod_mult_add modulo (
        .A(a_i[31:16]),           // input wire [31 : 0] A
        .B(b_i[15:0]),           // input wire [31 : 0] B
        .C(b_i[15:0]),           // input wire [31 : 0] C
        .SUBTRACT(),             // input wire SUBTRACT
        .P(r3),                  // output wire [63 : 0] P
        .PCOUT()                 // output wire [47 : 0] PCOUT
    );

    always_comb begin : modulo_multiplication
        r4 = (r3[31:16] == b_i) ? 16'd0 : r3[31:16];
    end

    assign c_o = {16'd0, r4};

endmodule : plntrd_sc

```

Figure 4.7 : Single Cycle Plantard Modular Reduction Added to RISC-V ALU Block

To run single cycle plantard modular reduction operation inline assembly method is used in C language program.

```

asm("plntrd_sc %[result1], %[value1], %[value2]\n\t" : [result1]
    ↪ "=r" (result1) : [value1] "r" (a*b_) , [value2] "r"
    ↪ (KYBER_Q));

```

Here, this command is equivalent of "% (mod)" operation in C language. As it is a trivial operation we define in programs, in this study using Plantard Modular Reduction, complexity of modulo operation is reduced. Modulo operation in programs calls "REM" instruction as defined in RISC-V ISA Specifications [29], but it is calculated as remainder of division. That also, slows down the cryptographic elements. This study suggest using Plantard Modular Reduction speeds-up the modulo operation that is uses multiplication and addition operations instead of "REM" instruction which uses division operation. In Figure 4.9 speed up result of new add custom instruction can be seen. Also in Figure 4.9 used method to benchmark modular multiplication can be seen.

```
# of Errors : 0
Average Cycle count for 1000 modular multiplication :
SW Modulo operation = 73
plntrd_sc added operation = 37
```

Figure 4.8 : Average cycles with and without "plntrd_sc" modular reduction instruction

4.4.2.2 Multi Cycle Instruction Addition

To add multi cycle instruction, I need to dive deep in to the RISC-V architecture. In order to add a multi cycle operation there has to be a stall operation which informs the architecture about if it is ready to used or is it busy with another instruction. Therefore, in Ibex RISC-V core, modifications are done in "id_stage" and "ex_block". In Figure 4.10, a glimpse of modification can be seen but there are other modifications that are too much to add.

Source codes of multi cycle plantard modular reduction added to multiplier/divider block in RISC-V can be seen in Figure 4.9.

Here, if Algorithm 4 is inspected, we can conclude that even-though 'C' is needed in Line 1, which is 'n' word size $n \times 2n$, we need only n sized $[2n - 1 : n]$ bits of result. In the Figure 4.11, result when custom multicyle operation is used can be seen.

4.4.2.3 Software Application of Custom Instructions

In order to compare reduction of time, I have written a Software with using "%" operator and inline assembly "plntrd_sc" and "plntrd_mc" instructions. Code written

```

for(int i=0; i<MM_count; i++){
    a = rand() % prime;
    b = rand() % prime;
    b_ = b*correction % prime;
    b_ = b_*inv_prime;

    usleep(2);
    pcount_reset();
    mtime_start();
    asm("pltrd_sc %[result1], %[value1], %[value2]\n\t"
        ↪ : [result1] "=r" (result1) : [value1] "r"
        ↪ (a*b_) , [value2] "r" (KYBER_Q));
    mtime_stop();
    uart_printf(&uart0, "Operation Time (Cycle) for
        ↪ Custom plntrd_sc instruction = %d\r\n",
        ↪ mtime_get64());
    time1 += mtime_get64();
    pcount_reset();
    mtime_start();
    result2 = a*b % KYBER_Q;
    mtime_stop();
    uart_printf(&uart0, "Operation Time (Cycle) for SW
        ↪ modulo multiplier = %d\r\n", mtime_get64());
    time2 += mtime_get64();
    if (result1 != result2)
    {
        uart_printf(&uart0, "Operation a x b \ \% p =
            ↪ %d x %d \% %d = %d\r\n", plantard(%d, %u,
            ↪ %d) = %d\r\n", a, b, KYBER_Q, result2,
            ↪ a, b_, KYBER_Q, result1);
        errors++;
    }
}

```

Figure 4.9 : Algorithm to benchmark modular multiplication

```

// custom added
modulo_op: begin
    // plntrd_mc operation
    if (~ex_valid_i) begin
        id_fsm_d      = MULTI_CYCLE;
        rf_we_raw     = 1'b0;
        stall_modulo  = 1'b1;
    end
end

```

Figure 4.10 : Ibex RISC-V Core Modification in ID stage

```

# of Errors : 0
Average Cycle count for 1000 modular multiplication :
SW Modulo operation = 291
plntrd_mc added operation = 146
-----

```

Figure 4.11 : Software comparison of modular multiplication with using "plantard_mc" instruction

for comparison can be accessed in Appedice A. Also, as Plantard modular reduction uses n bits of $3n$ bits multiplication result following types are added to access MSB and LSB of multiplication results.

```
typedef union a_32{
    uint16_t a16[2];
    uint32_t a32;
} r32;

typedef union a_64{
    uint16_t a32[2];
    uint32_t a64;
} r64;
```

Figure 4.12 : Accessing MSBs of Result $n \times 2n \bmod 2^{2n}$ multiplication

5. RESULTS

In this chapter, we will showcase the outcome of the Plantard modular reduction implemented in HDL and RISC-V core. Additionally, we will also display the results of the Plantard modular multiplication that was designed using purely HDL codes. Implementations of Barret modular reduction and Plantard modular reduction are straightforward implementation of defined algorithms in Chapter 2.

5.1 Barrett Modular Reduction

Different Barrett modular reduction modules are compared in Table 5.1. When comparing these implementations of Barrett modular reduction with Plantard modular reduction, Plantard modular reduction exhibits better utilization. Additionally, when comparing the frequency of these implementations, Plantard modular reduction performs better than Barrett modular reduction.

5.2 Plantard Modular Reduction

Table 5.1 : Implementation Utilization Results of the Modular Reduction Algorithms

Reduction	Algorithm	# of LUTs	# of FFs	# of DSPs
Barrett	CRYSTALS-Kyber	36	84	3
Barrett	CRYSTALS-Dilithium	63	178	5
Barrett	31-bit Primes	240	493	8
Plantard	CRYSTALS-Kyber	6	12	1
Plantard	CRYSTALS-Dilithium	31	0	2
Plantard	31-bit Primes	41	31	4

5.3 Plantard Modular Multiplication Area-Performance Benchmarks

This study provides designers to selects which module they need to use for example if their design clock is close to 500 MHz and supports pipelines they can select the third one from the table but this design is for prime of CRYSTAL-Kyber which means results

Table 5.2 : Implementation Performance Results of the Modular Reduction Algorithm Designs

Reduction	Algorithm	T(ns)	Pipeline	E(nJ)	Throughput(GB/s)
Barrett	CRYSTALS-Kyber	4.18	4	0.535	0.358
Barrett	CRYSTALS-Dilithium	5.78	5	0.973	0.497
Barrett	31-bit Primes	8.76	9	0.96	0.456
Plantard	CRYSTALS-Kyber	1.818	3	0.234	0.82
Plantard	CRYSTALS-Dilithium	1.826	4	0.287	1.57
Plantard	31-bit Primes	1.818	6	0.338	2.20

are generated using 12-bit prime of CRYSTALS-Kyber. We can deduct from Table 5.1 and Table 5.2 that you should not select any design for CRYSTALS-Dilithium as prime size gets bigger than the CRYSTAL-Kyber.

Table 5.3 : Utilization and Performance of Plantard modular multiplication algorithm for CRYSTALS-Kyber and CRYSTALS-Dilithium PQC Schemes

PQC Algorithm	LUT	FF	DSP	CPD (ns)	Throughput(Mbps)	Pipeline
CRYSTALS-Kyber	270	12	0	13.39	895.56	1
CRYSTALS-Kyber	8	43	3	1.89	6319.44	10
CRYSTALS-Kyber	29	28	2	3.6	3333.24	5
CRYSTALS-Dilithium	876	23	0	15.1	1523.29	1
CRYSTALS-Dilithium	32	96	5	2.59	8846.26	12
CRYSTALS-Dilithium	881	671	0	4.69	4893.71	7

* Critical Path Delay (CPD)

5.4 Plantard Modular Reduction in Ibex RISC-V core

In this section, added custom instruction of "pltrd_sc" reduces time as seen in Figure 4.8 and also if we want to compare code size as seen in Figures 5.1, 5.2 both have 7 instructions to calculate modular multiplication with prime smaller than 16-bit size. As it is 7 instructions, we can conclude that in terms of code size both have equal code sizes.

Clock cycle difference of this two implementation comes from "REM" instruction which is remainder of division and division completes in 1 to 37 clock cycles in Ibex RISC-V core [6].

Also, utilization on Nexys 4 DDR FPGA shown in Figure 5.4-a.

```

asm("pltrd_sc %[result1], %[value1], %[value2] : [result1]
    => "=r" (result1) : [value1] "r" (a*b_) , [value2] "r" (KYBER_Q));
10224:          fd245703          lhu      a4,-46(s0)
10228:          fc042783          lw      a5,-64(s0)
1022c:          02f707b3          mul      a5,a4,a5
10230:          00001737          lui      a4,0x1
10234:          d0170713          add      a4,a4,-767 #
    => d01 <__DYNAMIC+0xd01>
10238:          70e787b3          pltrd_sc a5,a5,a4
1023c:          fcf42623          sw      a5,-52(s0)

```

Figure 5.1 : Modular multiplication with "pltrd_sc added"

```

result2 = a*b % KYBER_Q;
1028c:          fd245703          lhu      a4,-46(s0)
10290:          fd045783          lhu      a5,-48(s0)
10294:          02f70733          mul      a4,a4,a5
10298:          000017b7          lui      a5,0x1
1029c:          d0178793          add      a5,a5,-767 #
    => d01 <__DYNAMIC+0xd01>
102a0:          02f767b3          rem      a5,a4,a5
102a4:          fcf42423          sw      a5,-56(s0)

```

Figure 5.2 : Modular multiplication with software modulo

Here, this result shows that Utilization reports on Xilinx Vivado. At the and "pltrd_sc" module uses 9 Slices when whole core utilizes 796 Slices. That overall means adding such a small module made an increase in performance almost 50% as operation time for the modular multiplication operation.

Reference NTT function with custom instruction can be seen in Appendix B.

```

'r' polynomial has been randomly selected...
-----
KAT will be generated by using Montgomery Reduction
-----
Cycle count for NTT module with software call = 232913
Cycle count for NTT module with plantard modular multiplication = 208724
-----
# of errors found in NTT(r) = 0
-----

```

Figure 5.3 : NTT performance comparison with "pltrd_sc" instruction and SW modulo

Moving on 31-bit prime values, multi cycle plantard modular reduction is implemented as seen in Appendix A. This module is integrated to RISC-V Ex Block and as it is multicyle instruction processor stall signals are modified.

From the Table 5.4, performance of the equivalent instructions can be seen.

Table 5.4 : Performance Analysis of RISC-V Core with Combinations of Designed Instruction Extensions

Reduction	Prime Size	Execution Time (Cycles)
Software	16-bit	1-37*
Software	32-bit	1-37*
plantard_sc	16-bit	4
plantard_mc	32-bit	7

* Here, as REM instruction is defined as remainder of division in Ibex RISC-v Core possible division execution time is selected.

Both of this methods needs to modify ID-Stage of RISC-V processors. Timing results of new added "pltrd_mc" instruction can be seen in Figure 4.11.

(a)

Ibex RISC-V Core without any custom instruction

Name	Slice LUTs (63400)	Slice Registers (126800)	F7 Muxes (31700)	F8 Muxes (15850)	Slice (15850)	LUT as Logic (63400)	DSPs (240)
u_ibex_core (ibex_core)	2415	888	4	0	737	2415	1
wb_stage_i (ibex_wb_stage)	32	0	0	0	22	32	0
load_store_unit_i (ibex_load_store_)	132	66	0	0	63	132	0
if_stage_i (ibex_if_stage)	400	278	0	0	189	400	0
id_stage_i (ibex_id_stage)	337	81	0	0	190	337	0
ex_block_i (ibex_ex_block)	779	75	1	0	242	779	1
gen_multdiv_fast.multdiv_i (ibex_)	455	75	0	0	163	455	1
alu_i (ibex_alu)	272	0	1	0	119	272	0
cs_registers_i (ibex_cs_registers)	729	388	3	0	252	729	0
gen_regfile_ff.register_file_i (ibex_regis	607	992	256	0	483	607	0

(b)

Ibex RISC-V Core with Single Cycle Plantard modular reduction instruction

Name	Slice LUTs (63400)	Slice Registers (126800)	F7 Muxes (31700)	F8 Muxes (15850)	Slice (15850)	LUT as Logic (63400)	DSPs (240)
u_ibex_core (ibex_core)	2470	888	20	0	772	2470	2
wb_stage_i (ibex_wb_stage)	32	0	0	0	23	32	0
load_store_unit_i (ibex_load_store_)	133	66	0	0	65	133	0
if_stage_i (ibex_if_stage)	401	278	0	0	196	401	0
id_stage_i (ibex_id_stage)	348	81	0	0	180	348	0
ex_block_i (ibex_ex_block)	824	75	17	0	279	824	2
gen_multdiv_fast.multdiv_i (ibex_)	458	75	0	0	175	458	1
alu_i (ibex_alu)	311	0	17	0	120	311	1
plntrd_s (plntrd_sc)	20	0	0	0	12	20	1
cs_registers_i (ibex_cs_registers)	727	388	3	0	290	727	0
gen_regfile_ff.register_file_i (ibex_regis	607	992	256	0	628	607	0












Figure 5.4 : Comparisons of RISC-V core areas with combinations of different Plantard modular reduction algorithms

5.4.1 Critical Path Analysis

Critical Path Analysis (CPA) is crucial when integrating custom instructions into our processor's core. It ensures the critical path, the longest sequence of dependent tasks, remains unaffected. While we anticipate the new instructions to function properly, it's equally important they don't impede the processor's speed. Therefore, CPA is instrumental in maintaining the processor's performance while enhancing its capabilities with custom instructions. In essence, CPA allows us to expand the processor's functionality without compromising its performance.

(c)

Ibex RISC-V Core with Multi Cycle Plantard modular reduction instruction

Name	Slice LUTs (63400)	Slice Registers (126800)	F7 Muxes (31700)	F8 Muxes (15850)	Slice (15850)	LUT as Logic (63400)	DSPs (240)
▼  u_ibex_core (ibex_core)	2687	955	36	0	849	2687	2
 wb_stage_i (ibex_wb_stage)	32	0	0	0	25	32	0
 load_store_unit_i (ibex_load_store_)	131	66	0	0	67	131	0
>  if_stage_i (ibex_if_stage)	400	278	0	0	193	400	0
>  id_stage_i (ibex_id_stage)	352	81	0	0	163	352	0
▼  ex_block_i (ibex_ex_block)	1039	142	33	0	329	1039	2
 plntrd_multicycle (plntrd_mc)	246	67	32	0	86	246	1
 gen_multdiv_fast.multdiv_i (ibex_)	483	75	0	0	182	483	1
 alu_i (ibex_alu)	272	0	1	0	105	272	0
>  cs_registers_i (ibex_cs_registers)	728	388	3	0	271	728	0
 gen_regfile_ff.register_file_i (ibex_regis	607	992	256	0	467	607	0

(d)

Ibex RISC-V Core with Both Single-Multi Cycle Plantard modular reduction instruction













Name	Slice LUTs (63400)	Slice Registers (126800)	F7 Muxes (31700)	F8 Muxes (15850)	Slice (15850)	LUT as Logic (63400)	DSPs (240)
▼  u_ibex_core (ibex_core)	2751	955	52	0	870	2751	3
 wb_stage_i (ibex_wb_stage)	32	0	0	0	30	32	0
 load_store_unit_i (ibex_load_store_)	133	66	0	0	74	133	0
>  if_stage_i (ibex_if_stage)	401	278	0	0	214	401	0
>  id_stage_i (ibex_id_stage)	366	81	0	0	198	366	0
▼  ex_block_i (ibex_ex_block)	1088	142	49	0	378	1088	3
 plntrd_multicycle (plntrd_mc)	246	67	32	0	92	246	1
 gen_multdiv_fast.multdiv_i (ibex_)	468	75	0	0	169	468	1
▼  alu_i (ibex_alu)	313	0	17	0	122	313	1
>  plntrd_s (plntrd_sc)	19	0	0	0	10	19	1
>  cs_registers_i (ibex_cs_registers)	727	388	3	0	297	727	0
 gen_regfile_ff.register_file_i (ibex_regis	607	992	256	0	659	607	0

Figure 5.4 : Comparisons of RISC-V core areas with combinations of different Plantard modular reduction algorithms (cont.)

Also, if we analyze the critical path before adding custom instructions and after adding custom instructions, we can see that added custom instructions does not effects critical path more than 10%, therefore it is highly connected with FPGA architecture and optimization of utilized area on device.

As seen in Figure 5.5, we can observe critical path in Ibex RISC-V core.

(a)

Critical Path of Ibex RISC-V Core without any custom instruction

Name	Slack	Levels	High Fanout	From	To	Total Delay	Logic Delay	Net Delay	Requirement	Source Clock	Destination Clock
Path 1	1.301	12	84	wb_spram/spr...8/CLKARDCLK	wb_ibex_core/in...id_o_reg[20]D	18.458	4.781	13.677	20.0	clk_50_unbuf	clk_50_unbuf
Path 2	1.321	21	268	wb_ibex_core/in...id_o_reg[21]C	wb_spram/spra...am_126/WEA[2]	17.954	4.096	13.858	20.0	clk_50_unbuf	clk_50_unbuf
Path 3	1.373	21	268	wb_ibex_core/in...id_o_reg[21]C	wb_spram/spra...am_74/WEA[0]	17.937	4.096	13.841	20.0	clk_50_unbuf	clk_50_unbuf
Path 4	1.377	24	268	wb_ibex_core/in...id_o_reg[21]C	wb_ibex_core/in...q_reg[1]44CE	18.017	4.614	13.403	20.0	clk_50_unbuf	clk_50_unbuf
Path 5	1.377	24	268	wb_ibex_core/in...id_o_reg[21]C	wb_ibex_core/in...q_reg[1]5CE	18.017	4.614	13.403	20.0	clk_50_unbuf	clk_50_unbuf
Path 6	1.414	21	268	wb_ibex_core/in...id_o_reg[21]C	wb_spram/spra...am_94/WEA[2]	17.855	4.096	13.759	20.0	clk_50_unbuf	clk_50_unbuf
Path 7	1.473	21	268	wb_ibex_core/in...id_o_reg[21]C	wb_spram/spra...am_120/WEA[2]	17.797	4.096	13.701	20.0	clk_50_unbuf	clk_50_unbuf
Path 8	1.494	20	268	wb_ibex_core/in...id_o_reg[21]C	wb_spram/spra..._93/DIAD[30]	18.062	3.972	14.090	20.0	clk_50_unbuf	clk_50_unbuf
Path 9	1.510	24	268	wb_ibex_core/in...id_o_reg[21]C	wb_ibex_core/in...q_reg[0]5CE	17.889	4.614	13.275	20.0	clk_50_unbuf	clk_50_unbuf

(b)

Ibex RISC-V Core with Single Cycle Plantard modular reduction instruction

Name	Slack	Levels	High Fanout	From	To	Total Delay	Logic Delay	Net Delay	Requirement	Source Clock	Destination Clock
Path 1	0.263	18	179	wb_ibex_core/in...id_o_reg[1]C	wb_spram/spra...am_103/WEA[2]	18.972	3.843	15.129	20.0	clk_50_unbuf	clk_50_unbuf
Path 2	0.287	22	199	wb_ibex_core/in...id_o_reg[1]C	wb_spram/spra...am_36/WEA[2]	18.949	4.576	14.373	20.0	clk_50_unbuf	clk_50_unbuf
Path 3	0.386	22	204	wb_ibex_core/in...id_o_reg[1]C	wb_spram/spra...am_98/WEA[3]	18.900	4.416	14.484	20.0	clk_50_unbuf	clk_50_unbuf
Path 4	0.399	18	179	wb_ibex_core/in...id_o_reg[1]C	wb_spram/spra...am_103/WEA[0]	18.836	3.843	14.993	20.0	clk_50_unbuf	clk_50_unbuf
Path 5	0.406	22	199	wb_ibex_core/in...id_o_reg[1]C	wb_spram/spra...am_107/WEA[2]	18.835	4.576	14.259	20.0	clk_50_unbuf	clk_50_unbuf
Path 6	0.424	22	199	wb_ibex_core/in...id_o_reg[1]C	wb_spram/spra...am_98/WEA[2]	18.862	4.576	14.286	20.0	clk_50_unbuf	clk_50_unbuf
Path 7	0.455	22	199	wb_ibex_core/in...id_o_reg[1]C	wb_spram/spra...am_97/WEA[2]	18.777	4.576	14.201	20.0	clk_50_unbuf	clk_50_unbuf
Path 8	0.471	18	179	wb_ibex_core/in...id_o_reg[1]C	wb_spram/spra...am_34/WEA[2]	18.758	3.843	14.915	20.0	clk_50_unbuf	clk_50_unbuf
Path 9	0.484	22	199	wb_ibex_core/in...id_o_reg[1]C	wb_spram/spra...am_99/WEA[2]	18.790	4.576	14.214	20.0	clk_50_unbuf	clk_50_unbuf
Path 10	0.492	22	204	wb_ibex_core/in...id_o_reg[1]C	wb_spram/spr...m_98/ENARDEN	18.883	4.416	14.467	20.0	clk_50_unbuf	clk_50_unbuf

(c)

Critical Path of Ibex RISC-V Core with Multi Cycle Plantard modular reduction instruction

Name	Slack	Levels	High Fanout	From	To	Total Delay	Logic Delay	Net Delay	Requirement	Source Clock	Destination Clock
Path 1	0.195	24	43	wb_ibex_core/in...id_o_reg[2]C	wb_ibex_core/in...r_q_reg[56]CE	19.203	4.576	14.627	20.0	clk_50_unbuf	clk_50_unbuf
Path 2	0.195	24	43	wb_ibex_core/in...id_o_reg[2]C	wb_ibex_core/in...r_q_reg[57]CE	19.203	4.576	14.627	20.0	clk_50_unbuf	clk_50_unbuf
Path 3	0.195	24	43	wb_ibex_core/in...id_o_reg[2]C	wb_ibex_core/in...r_q_reg[58]CE	19.203	4.576	14.627	20.0	clk_50_unbuf	clk_50_unbuf
Path 4	0.195	24	43	wb_ibex_core/in...id_o_reg[2]C	wb_ibex_core/in...r_q_reg[59]CE	19.203	4.576	14.627	20.0	clk_50_unbuf	clk_50_unbuf
Path 5	0.313	24	43	wb_ibex_core/in...id_o_reg[2]C	wb_ibex_core/in...r_q_reg[55]CE	19.085	4.576	14.509	20.0	clk_50_unbuf	clk_50_unbuf
Path 6	0.313	24	43	wb_ibex_core/in...id_o_reg[2]C	wb_ibex_core/in...r_q_reg[62]CE	19.085	4.576	14.509	20.0	clk_50_unbuf	clk_50_unbuf
Path 7	0.313	24	43	wb_ibex_core/in...id_o_reg[2]C	wb_ibex_core/in...r_q_reg[63]CE	19.085	4.576	14.509	20.0	clk_50_unbuf	clk_50_unbuf
Path 8	0.347	24	43	wb_ibex_core/in...id_o_reg[2]C	wb_ibex_core/in...r_q_reg[48]CE	19.051	4.576	14.475	20.0	clk_50_unbuf	clk_50_unbuf
Path 9	0.347	24	43	wb_ibex_core/in...id_o_reg[2]C	wb_ibex_core/in...r_q_reg[54]CE	19.051	4.576	14.475	20.0	clk_50_unbuf	clk_50_unbuf
Path 10	0.390	24	43	wb_ibex_core/in...id_o_reg[2]C	wb_ibex_core/in...r_q_reg[49]CE	19.008	4.576	14.432	20.0	clk_50_unbuf	clk_50_unbuf

(d)

Critical Path of Ibex RISC-V Core with Both Single-Multi Cycle Plantard modular reduction instruction

Name	Slack	Levels	High Fanout	From	To	Total Delay	Logic Delay	Net Delay	Requirement	Source Clock	Destination Clock
Path 1	0.183	20	199	wb_ibex_core/in...id_o_reg[2]C	wb_spram/spra...am_82/WEA[1]	18.924	3.730	15.194	20.0	clk_50_unbuf	clk_50_unbuf
Path 2	0.254	21	199	wb_ibex_core/in...id_o_reg[2]C	wb_spram/spra...am_94/WEA[0]	18.821	3.854	14.967	20.0	clk_50_unbuf	clk_50_unbuf
Path 3	0.298	21	199	wb_ibex_core/in...id_o_reg[2]C	wb_spram/spra...am_94/WEA[1]	18.777	3.854	14.923	20.0	clk_50_unbuf	clk_50_unbuf
Path 4	0.340	20	199	wb_ibex_core/in...id_o_reg[2]C	wb_spram/spra...am_82/WEA[2]	18.767	3.730	15.037	20.0	clk_50_unbuf	clk_50_unbuf
Path 5	0.353	20	199	wb_ibex_core/in...id_o_reg[2]C	wb_spram/spr...m_95/ENARDEN	18.839	3.730	15.109	20.0	clk_50_unbuf	clk_50_unbuf
Path 6	0.363	20	199	wb_ibex_core/in...id_o_reg[2]C	wb_spram/spra...am_95/WEA[1]	18.740	3.730	15.010	20.0	clk_50_unbuf	clk_50_unbuf
Path 7	0.399	15	270	wb_ibex_core/in...id_o_reg[21]C	wb_ibex_core/in...reg_q_reg[7]D	19.422	7.397	12.025	20.0	clk_50_unbuf	clk_50_unbuf
Path 8	0.404	15	270	wb_ibex_core/in...id_o_reg[21]C	wb_ibex_core/in...reg_q_reg[7]D	19.416	7.397	12.019	20.0	clk_50_unbuf	clk_50_unbuf
Path 9	0.412	20	199	wb_ibex_core/in...id_o_reg[2]C	wb_spram/spra...am_82/WEA[3]	18.696	3.730	14.966	20.0	clk_50_unbuf	clk_50_unbuf
Path 10	0.416	15	270	wb_ibex_core/in...id_o_reg[21]C	wb_ibex_core/in...reg_q_reg[7]D	19.404	7.397	12.007	20.0	clk_50_unbuf	clk_50_unbuf

Figure 5.5 : Comparisons of Critical Path in RISC-V core areas with combinations of different Plantard modular reduction algorithms (cont.)

5.4.2 Power and Energy Analysis

Vivado is a comprehensive software suite from Xilinx that is used for the synthesis and analysis of HDL designs, superseding Xilinx ISE with additional features for system on a chip development and high-level synthesis. Vivado also includes a feature for power analysis, allowing engineers to understand the power consumption of their designs. This is crucial in the modern world where energy efficiency is a key product requirement. The power analysis feature in Vivado allows for both static and dynamic power consumption analysis. Static power analysis is the power consumed by the device when it is not performing any operations, while dynamic power analysis is the power consumed during the operation of the device. The power analysis tool in Vivado provides a detailed report of power consumption, which includes the core, I/O, clock network, signal, and leakage power.

Energy is the capacity to do work, and it's calculated as the product of power and time. Power is the rate at which work is done or energy is transferred, usually measured in watts (W). Time is typically measured in seconds (s). Therefore, the energy (E) can be calculated using the formula:

$$E = P \times t \quad (5.1)$$

where:

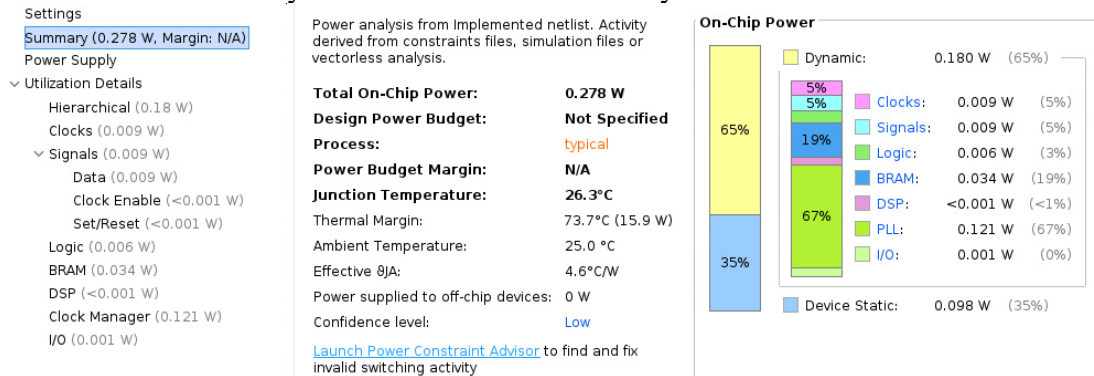
- E is the energy, measured in joules (J)
- P is the power, measured in watts (W)
- t is the time, measured in seconds (s)

For example, if a device uses 1W of power continuously for 10 seconds, the energy used would be 10J. This calculation is fundamental in many areas of engineering, including electrical and computer engineering.

The Switching Activity Interchange Format (SAIF) file is crucial for accurate power analysis. It contains annotations of how often nets in the design switch¹. When you

(a)

Power Analysis of RISC-V Ibex Demo System without SAIF File



(b)

Power Analysis of RISC-V Ibex Demo System with SAIF File

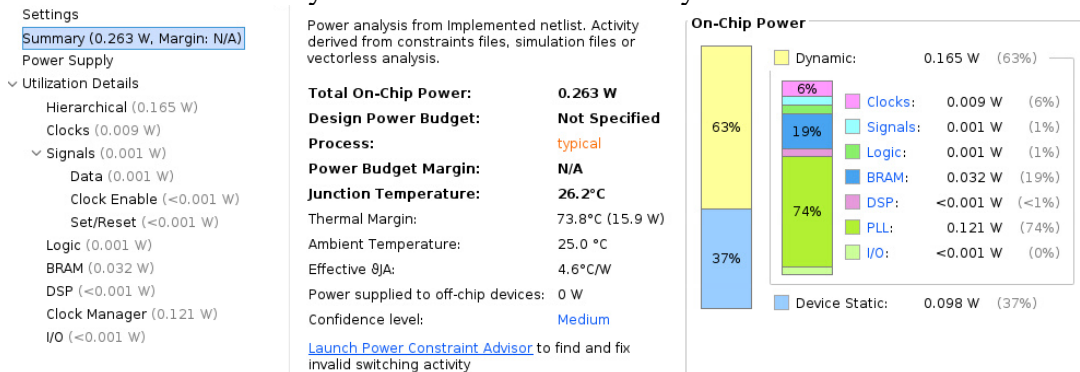


Figure 5.6 : Differences of Power Analysis with and without SAIF File

analyze power in tools like Xilinx's XPA, the tool uses a VCD/SAIF file to include the actual signal transitions for power calculations. This file helps the tool match the number of nets in the VCD file to the total number of design nets, which shows up in the power report. Therefore, a SAIF file provides detailed switching activity information, enabling more accurate power estimation and optimization.

Here, a SAIF file can be generated using post-implementation functional simulation, providing an accurate estimation of the average power of the implemented design. Once we have the average power of the implemented design, we can simply multiply the power by the period to compare different coding styles. These coding styles include software and custom hardware implementations of the modulo operation and the NTT. In the Figure 5.6, effect of SAIF File in the process of Power Analysis can be seen.

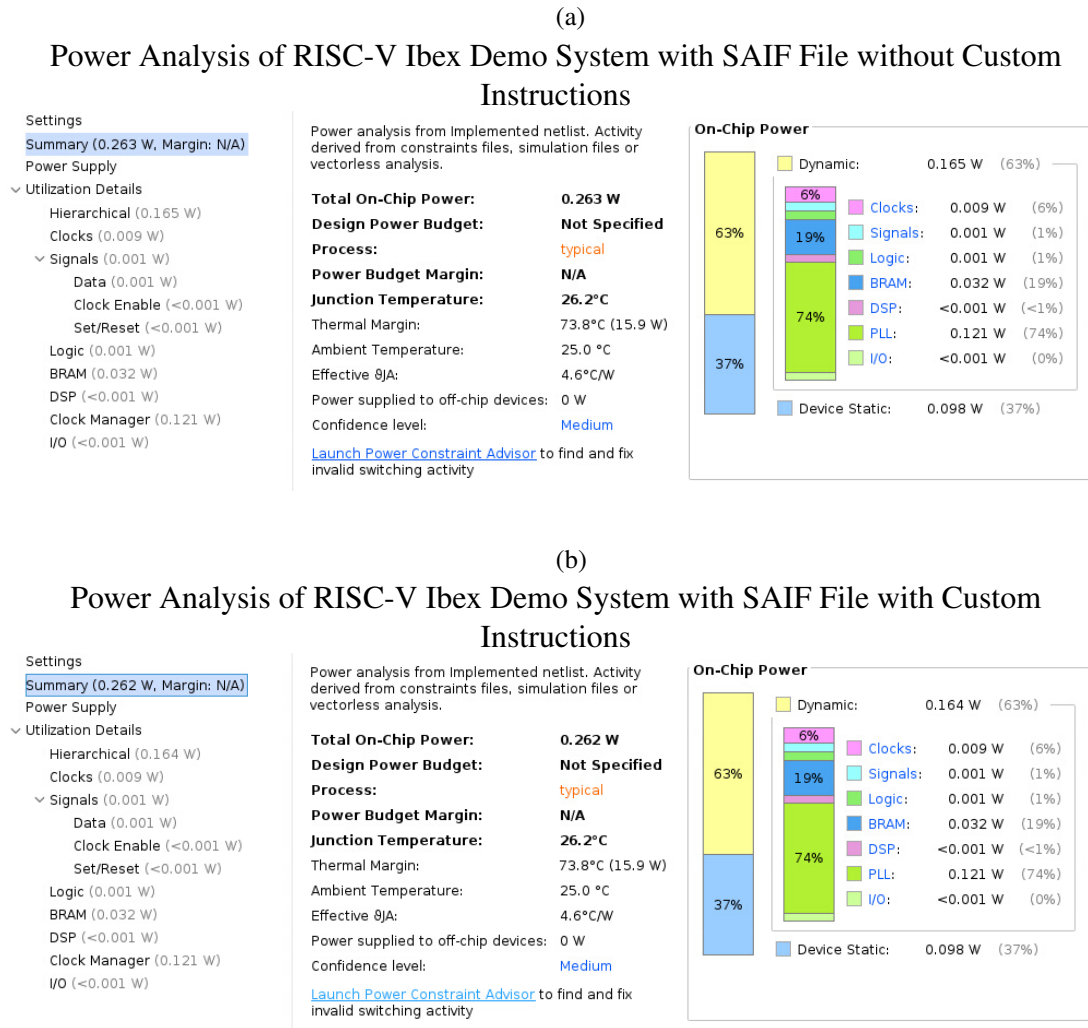


Figure 5.7 : Differences of Power Analysis with and without SAIF File

Right after as we've seen the effect of SAIF Simulation, we can compare results of power analysis, with and without custom instruction addition to RISC-V core.

Here from the Figure 5.7, it can be deduced that there is no significant power change in the core as I have added custom instruction modules "plntrd_sc" and "plntrd_mc" to the RISC-V core. Here, first expectation was to increase in power since area of the implement design is increased but we should not forget added modules to design has enable signal which is not clock enable but it is similar to clock enable as if the module is not enabled, gates and flip-flops in the module is in their resting state that means these are not switching until they are enabled.

So, from the Figure 5.7, we can simply assume that Average power is $\approx 0.263Watts$. As the power is almost equal, only variable that can affect the Energy calculation is processing time with and without new defined custom instructions.

Here we can simply create Table 5.5 as follows:

Table 5.5 : Performance Analysis of RISC-V Core with Combinations of Designed Instruction Extensions

NTT Implementation	Execution Time (Cycles)	Average Power(W)	Energy(nJ)
Software	232913	0.263	1225122
Custom Instructions	208924	0.262	1094761

* Here, one Cycle Period is taken as 20ns as both design's operating frequency is 50MHz.

6. CONCLUSIONS

In this study, RISC-V I and M set instructions are extended with customized instructions of Plantard modular reduction namely "pltrd_sc" and "pltrd_mc". Also this custom instructions proposed to be used in Cryptographic applications where modular reduction is needed and NTT is used.

The addition of Plantard modular reduction significantly increased the performance of modular multiplication, reducing the execution time by approximately 50%. Importantly, this enhancement was achieved without causing any increase in code size.

As prime sizes gets bigger complexity of operations increases and critical path begins to increase therefore aside from single cycle instruction, a multi cycle instruction is also proposed. As designed single cycle Plantard modular reduction supports 15-bits primes, designed multi cycle Plantard modular reduction supports 31-bit primes completes operation in 5-cycle which is also shared in Appendix.

Using multi cycle plantard reduction method as multiplication input values are 32-bits long, and result is 64-bits long it cannot be done with only "DIV" instruction, but using custom "pltrd_mc" instruction which takes 32-bits long inputs, requires only one instruction therefore aside from the increase in performance, decrease in code size is observed.

After adding these instruction in RTL RISC-V design, The RISC-V GNU Toolchain has been enhanced to convert high-level programs into assembly code by introducing new instructions. The opcodes for mask and match values associated with these instructions are integrated into GCC Binutils. Inline assembly is utilized within the source code to generate these new instructions, and the GCC Binutils assembler subsequently assembles them based on the assembly block present in the source code.

The impact of the newly introduced instructions on program performance is analyzed on a specifically designed microcontroller structure featuring the Ibex RISC-V

core. This involves expanding the decoder and incorporating custom modules into the RISC-V core. The behaviors of the new instructions are outlined in the Register-Transfer Level (RTL) designs of the added modules. The C code is employed to access Control and Status Registers (CSRs) of the RISC-V Core, enabling the determination of the number of clock cycles. The number of clock cycles, signifying the output, is subsequently monitored using a UART module integrated as a peripheral to the RISC-V core.

The experimental findings indicate that augmenting RISC-V with tailored custom instructions significantly impacts both the total clock cycles and the number of program instructions. For the modular multiplication operation with Single Cycle Plantard modular reduction, the execution time is decreased by approximately 50%, and there is no change in the code size. Similarly, employing Multi-Cycle Plantard modular reduction results in a roughly 50% operation time of software reduction, and the code size is decreased compared to using the software modulo operator with Plantard modular reduction.

Throughout this study, the adapted RISC-V core and NTT software were deployed on FPGA hardware, simulating the behavior of the incorporated custom instructions. In addition to monitoring the outputs of these custom instructions, the primary focus of this study is on Plantard modular reduction. It introduces a novel hardware implementation for Plantard modular reduction, addressing the absence of any existing hardware implementations for this method.

REFERENCES

- [1] **Can, F. and Örs Yalçın, S.B.** (2024). *Hardware Design of k^2 red Modular Multiplication Algorithm*.
- [2] **Jati, A., Gupta, N., Chattopadhyay, A. and Sanadhya, S.K.** (2023). A Configurable CRYSTALS-Kyber Hardware Implementation with Side-Channel Protection, *ACM Transactions in Embedded Computing Systems*.
- [3] **Niasar, M.B., Azarderakhsh, R. and Kermani, M.M.** (2021). *High-Speed NTT-based Polynomial Multiplication Accelerator for CRYSTALS-Kyber Post-Quantum Cryptography*, https://www.researchgate.net/publication/351345144_High-Speed_NTT-based_Polynomial_Multiplication_Accelerator_for_CRYSTALS-Kyber_Post-Quantum_Cryptography.
- [4] **Can, F., Üstün, A., Örs Yalçın, S.B., Alaybeylioğlu, E. and Savaş, E.** (2024). Hardware Implementation of K^2 RED and Plantard Modular Multiplication Algorithms in Post-Quantum Cryptography, *PAnhellenic Conference on Electronics and Telecommunications (PACET)*.
- [5] **Kanter, D.** (2016). *RISC-V OFFERS SIMPLE, MODULAR ISA New CPU Instruction Set Is Open and Extensible*, <https://riscv.org/wp-content/uploads/2016/04/RISC-V-Offers-Simple-Modular-ISA.pdf>.
- [6] **LowRisc** (2017). https://ibex-core.readthedocs.io/en/latest/03_reference/pipeline_details.html.
- [7] **Shor, P.W.** (1999). Polynomial-Time Algorithms for Prime Factorization and Discrete Logarithms on a Quantum Computer, *SIAM Review*, 41(2), 303–332, <https://doi.org/10.1137/S0036144598347011>, <https://doi.org/10.1137/S0036144598347011>.
- [8] **Grover, L.K.** (1996). A fast quantum mechanical algorithm for database search, *Proceedings of the twenty-eighth annual ACM symposium on Theory of computing - STOC '96*, <http://dx.doi.org/10.1145/237814.237866>.
- [9] **National Institute of Standards and Technology.** *NIST*, <https://www.nist.gov/>.

- [10] (2024). *view of improved plantard arithmetic for lattice-based*, <https://tches.iacr.org/index.php/TCHES/article/view/9833/9337>.
- [11] **Wu, Y. and Yue, Q.** (2021). Further factorization of $x^n - 1$ over a finite field (II), *Discrete mathematics, algorithms, and applications*, 13(06), <https://eprint.iacr.org/2023/1962.pdf>.
- [12] **Huang, J., Zhao, H., Zhang, J., Dai, W., Zhou, L., Cheung, R.C.C., Koç, and Chen, D.** (2023). Yet another Improvement of Plantard Arithmetic for Faster Kyber on Low-end 32-bit IoT Devices, *ArXiv*, *abs/2309.00440*, <http://arxiv.org/abs/2309.00440>.
- [13] **Bos, J., Ducas, L., Kiltz, E., Lepoint, T., Lyubashevsky, V., Schanck, J.M., Schwabe, P., Seiler, G. and Stehlé, D.** (2017). *CRYSTALS – Kyber: a CCA-secure module-lattice-based KEM*, Cryptology ePrint Archive, Paper 2017/634, <https://eprint.iacr.org/2017/634>, <https://eprint.iacr.org/2017/634>.
- [14] **Lyubashevsky, V., Peikert, C. and Regev, O.** (2010). On Ideal Lattices and Learning with Errors over Rings, *H. Gilbert, editor, Advances in Cryptology – EUROCRYPT 2010*, Springer Berlin Heidelberg, Berlin, Heidelberg, pp.1–23.
- [15] **Chen, C., Danba, O., Hostein, J., Hülsing, A., Rijneveld, J., Schanck, J., Schwabe, P., Whyte, W. and Zhang, Z.** (2019). *NTRU Algorithm Specifications And Supporting Documentation*, <https://ntru.org/f/ntru-20190330.pdf>.
- [16] **Bai, S., Ducas, L., Kiltz, E., Lepoint, T., Lyubashevsky, V., Schwabe, P., Seiler, G. and Stehlé, D.** (2021). *CRYSTALS-Dilithium Algorithm Specifications and Supporting Documentation (Version 3.1)*, <https://pq-crystals.org/dilithium/data/dilithium-specification-round3-20210208.pdf>.
- [17] **Lyubashevsky, V.** (2009). Fiat-Shamir with Aborts: Applications to Lattice and Factoring-Based Signatures, *M. Matsui, editor, Advances in Cryptology – ASIACRYPT 2009*, Springer Berlin Heidelberg, Berlin, Heidelberg, pp.598–616.
- [18] **Soni, D., Basu, K., Nabeel, M., Aaraj, N., Manzano, M. and Karri, R.** (2021). *CRYSTALS-Dilithium*, Springer International Publishing, Cham, pp.13–30, https://doi.org/10.1007/978-3-030-57682-0_2.
- [19] **Kundi, D.e.S., Zhang, Y., Wang, C., Khalid, A., O'Neill, M. and Liu, W.** (2022). Ultra High-Speed Polynomial Multiplications for Lattice-Based Cryptography on FPGAs, *IEEE Transactions on Emerging Topics in Computing*, 10(4), 1993–2005.

- [20] **Du, C. and Bai, G.** (2016). Towards efficient polynomial multiplication for lattice-based cryptography, *2016 IEEE International Symposium on Circuits and Systems (ISCAS)*, pp.1178–1181.
- [21] **Satriawan, A., Syafalni, I., Mareta, R., Anshori, I., Shalannanda, W. and Barra, A.** (2023). Conceptual Review on Number Theoretic Transform and Comprehensive Review on Its Implementations, *IEEE Access*, *11*, 70288–70316.
- [22] **Liang, Z. and Zhao, Y.** (2022). *Number Theoretic Transform and Its Applications in Lattice-based Cryptosystems: A Survey*, 2211.13546.
- [23] **Barrett, P.M.** (2007). Implementing the Rivest Shamir and Adleman Public Key Encryption Algorithm on a Standard Digital Signal Processor, 311–323, https://link.springer.com/chapter/10.1007/3-540-47721-7_24#citeas.
- [24] **Bisheh Niasar, M., Azarderakhsh, R. and Mozaffari Kermani, M.** (2021). High-Speed NTT-based Polynomial Multiplication Accelerator for Post-Quantum Cryptography, pp.94–101.
- [25] **Plantard, T.** (2021). Efficient word size modular arithmetic, *2021 IEEE 28th Symposium on Computer Arithmetic (ARITH)*, IEEE.
- [26] **Huang, J., Zhao, H., Zhang, J., Dai, W., Zhou, L., Cheung, R.C.C., Koc, C.K. and Chen, D.** (2023). *Yet another Improvement of Plantard Arithmetic for Faster Kyber on Low-end 32-bit IoT Devices*, 2309.00440.
- [27] **Xilinx** (2023). https://www.xilinx.com/products/intellectual-property/multiply_adder.html.
- [28] **Patterson, D.A.** (1985). Reduced instruction set computers, *Communications of the ACM*, *28*(1), 8–21.
- [29] **RISC-V Foundation**, (2019). The RISC-V Instruction Set Manual, 20190608-Priv-MSU-Ratified.
- [30] **(GCC), G.C.C.** (2024). <https://gcc.gnu.org/>.
- [31] **Canonical** (2023). *Ubuntu 22.04.3 LTS (Jammy Jellyfish)*, <https://releases.ubuntu.com/jammy/>.
- [32] **Xilinx** (2023). https://www.xilinx.com/products/design-tools/vivado/vivado-whats-new.html#2022_2.
- [33] **CMAKE** (2023). <https://cmake.org/cmake/help/latest/>.
- [34] **pbing. ibex_wb**, https://github.com/pbing/ibex_wb.
- [35] **Digilent. Arty A7**, <https://reference.digilentinc.com/reference/programmable-logic/arty-a7/reference-manual>.

APPENDIX A : Multi-Cycle Instruction Addition to RISC-V Core EX block

```
// Plantard Modular Reduction
// define a stage register
logic [63:0] mod_res_d;
logic [63:0] stage_results;
logic [31:0] mod_res;

logic [15:0] mod_op_a;
logic [15:0] mod_op_b;

typedef enum logic [2:0] {
    ALBL, ALBH, AHBL, AHBH, COMP
} mod_fsm_e;
mod_fsm_e mod_state_q, mod_state_d;

assign mod_res = (mod_op_a * mod_op_b);

always_ff @(posedge clk_i or negedge rst_ni) begin : mod_mult_add
    if(~rst_ni) begin
        stage_results <= 0;
    end else begin
        stage_results <= mod_res_d;
    end
end

always_comb begin
    mod_op_a    = op_a_i[`OP_L];
    mod_op_b    = op_b_i[`OP_L];
    mod_res_d   = mod_res;
    mod_state_d = mod_state_q;

    mod_valid   = 1'b0;
    mod_hold    = 1'b0;

    unique case (mod_state_q)

        ALBL: begin
            // al*bl
            mod_op_a = op_a_i[`OP_L];
            mod_op_b = op_b_i[`OP_L];

            mod_res_d = mod_res;
            mod_state_d = ALBH;
        end

        ALBH: begin
            // al*bh<<16
            mod_op_a = op_a_i[`OP_L];
            mod_op_b = op_b_i[`OP_H];
```

```

    // result of AL*BL (in imd_val_q_i[0]) always unsigned with
    ↪ no carry
    mod_res_d = stage_results + {mod_res, 16'd0} ;
    mod_state_d = AHBL;
end

AHBL: begin
    // ah*bl<<16
    mod_op_a = op_a_i[`OP_H];
    mod_op_b = op_b_i[`OP_L];
    mod_res_d = stage_results + {mod_res, 16'd0}
    mod_state_d = AHBH;
end

AHBH: begin
    // ah*bh
    mod_op_a = op_a_i[`OP_H];
    mod_op_b = op_b_i[`OP_H];
    // result of AH*BL is not signed only if signed_mode_i ==
    ↪ 2'b00
    mod_res_d      = stage_results + {mod_res, 32'd0} + op_b_i;

    // Note no state transition will occur if mod_hold is set
    mod_state_d = COMP;
    mod_hold     = ~multdiv_ready_id_i;
end

COMP: begin
    mod_res_d      = (stage_results[63:32] == op_b_i) ? 64'd0 :
    ↪ {32'd0, stage_results[63:32]};
    // custom plntrd
    mod_valid       = 1'b1;
    // end custom
    // Note no state transition will occur if mult_hold is
    ↪ set
    mod_state_d = ALBL;
    mod_hold     = ~multdiv_ready_id_i;
end

default: begin
    mod_state_d = ALBL;
end

endcase // mod_state_q
end

always_ff @(posedge clk_i or negedge rst_ni) begin
    if (!rst_ni) begin
        mod_state_q <= ALBL;
    end else begin
        if (mod_en_internal) begin
            mod_state_q <= mod_state_d;
        end
    end
end
end

```

APPENDIX B : NTT functions that are used for comparison

```
// software
const uint16_t zetas[128] = {1, 1729, 2580, 3289, 2642, 630, 1897,
↪ 848, 1062, 1919, 193, 797, 2786, 3260, 569, 1746, 296, 2447,
↪ 1339, 1476, 3046, 56, 2240, 1333, 1426, 2094, 535, 2882, 2393,
↪ 2879, 1974, 821, 289, 331, 3253, 1756, 1197, 2304, 2277, 2055,
↪ 650, 1977, 2513, 632, 2865, 33, 1320, 1915, 2319, 1435, 807,
↪ 452, 1438, 2868, 1534, 2402, 2647, 2617, 1481, 648, 2474, 3110,
↪ 1227, 910, 17, 2761, 583, 2649, 1637, 723, 2288, 1100, 1409,
↪ 2662, 3281, 233, 756, 2156, 3015, 3050, 1703, 1651, 2789, 1789,
↪ 1847, 952, 1461, 2687, 939, 2308, 2437, 2388, 733, 2337, 268,
↪ 641, 1584, 2298, 2037, 3220, 375, 2549, 2090, 1645, 1063, 319,
↪ 2773, 757, 2099, 561, 2466, 2594, 2804, 1092, 403, 1026, 1143,
↪ 2150, 2775, 886, 1722, 1212, 1874, 1029, 2110, 2935, 885,
↪ 2154};

// plantard
const uint32_t zetas_p[128] = {1290168, 2230699446, 3328631909,
↪ 4243360600, 3408622288, 812805467, 2447447570, 1094061961,
↪ 1370157786, 2475831253, 249002310, 1028263423, 3594406395,
↪ 4205945745, 734105255, 2252632292, 381889553, 3157039644,
↪ 1727534158, 1904287092, 3929849920, 72249375, 2889974991,
↪ 1719793153, 1839778722, 2701610550, 690239563, 3718262466,
↪ 3087370604, 3714391964, 2546790461, 1059227441, 372858381,
↪ 427045412, 4196914574, 2265533966, 1544330386, 2972545705,
↪ 2937711185, 2651294021, 838608815, 2550660963, 3242190693,
↪ 815385801, 3696329620, 42575525, 1703020977, 2470670584,
↪ 2991898216, 1851390229, 1041165097, 583155668, 1855260731,
↪ 3700200122, 1979116802, 3098982111, 3415073125, 3376368103,
↪ 1910737929, 836028480, 3191874164, 4012420634, 1583035408,
↪ 1174052340, 21932846, 3562152210, 752167598, 3417653460,
↪ 2112004045, 932791035, 2951903026, 1419184148, 1817845876,
↪ 3434425636, 4233039261, 300609006, 975366560, 2781600929,
↪ 3889854731, 3935010590, 2197155094, 2130066389, 3598276897,
↪ 2308109491, 2382939200, 1228239371, 1884934581, 3466679822,
↪ 1211467195, 2977706375, 3144137970, 3080919767, 945692709,
↪ 3015121229, 345764865, 826997308, 2043625172, 2964804700,
↪ 2628071007, 4154339049, 483812778, 3288636719, 2696449880,
↪ 2122325384, 1371447954, 411563403, 3577634219, 976656727,
↪ 2708061387, 723783916, 3181552825, 3346694253, 3617629408,
↪ 1408862808, 519937465, 1323711759, 1474661346, 2773859924,
↪ 3580214553, 1143088323, 2221668274, 1563682897, 2417773720,
↪ 1327582262, 2722253228, 3786641338, 1141798155, 2779020594};

void ntt(uint16_t r[256]) {
    unsigned int len, start, j, k;
    uint16_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 = zeta * r[j + len] % KYBER_Q;
            r[j + len] = (r[j] - t) % KYBER_Q;
            r[j] = (r[j] + t) % KYBER_Q;
        }
    }
}

void ntt_p(uint16_t r[256]) {
    unsigned int len, start, j, k;
    uint16_t t;
    uint32_t zeta;
    uint32_t mult;

    k = 1;
    for(len = 128; len >= 2; len >>= 1) {
        for(start = 0; start < 256; start = j + len) {
            zeta = zetas_p[k++];
            for(j = start; j < start + len; j++) {
                mult = r[j + len] * zeta;
                asm("pltrd_sc %[result1], %[value1], %[value2]\n\t" :
                    ↪ [result1] "=r" (t) : [value1] "r" (mult) , [value2] "r"
                    ↪ (KYBER_Q));
                r[j + len] = (r[j] - t) % KYBER_Q;
                r[j] = (r[j] + t) % KYBER_Q;
            }
        }
    }
}

```

APPENDIX C : Multi cycle Plantard Modular multiplication assembly example

```

// pltrd_mc
    cx = (uint64_t) a*b_;

102a4:    f9c42783          lw      a5,-100(s0)
102a8:    00078a13          mv      s4,a5
102ac:    00000a93          li      s5,0
102b0:    f8442783          lw      a5,-124(s0)
102b4:    03478733          mul     a4,a5,s4
102b8:    f8042783          lw      a5,-128(s0)
102bc:    035787b3          mul     a5,a5,s5
102c0:    00f70733          add     a4,a4,a5
102c4:    f8042783          lw      a5,-128(s0)
102c8:    034786b3          mul     a3,a5,s4
102cc:    0347b7b3          mulhu   a5,a5,s4
102d0:    f4f42e23          sw      a5,-164(s0)
102d4:    f4d42c23          sw      a3,-168(s0)
102d8:    f5c42783          lw      a5,-164(s0)
102dc:    00f707b3          add     a5,a4,a5
102e0:    f4f42e23          sw      a5,-164(s0)
102e4:    f5842783          lw      a5,-168(s0)
102e8:    f5c42803          lw      a6,-164(s0)
102ec:    f6f42c23          sw      a5,-136(s0)
102f0:    f7042e23          sw      a6,-132(s0)
102f4:    f6f42c23          sw      a5,-136(s0)
102f8:    f7042e23          sw      a6,-132(s0)

    cx = cx >> 32;

102fc:    f7c42783          lw      a5,-132(s0)
10300:    0007d793          srl     a5,a5,0x0
10304:    f4f42823          sw      a5,-176(s0)
10308:    f4042a23          sw      zero,-172(s0)
1030c:    f5042783          lw      a5,-176(s0)
10310:    f5442803          lw      a6,-172(s0)
10314:    f6f42c23          sw      a5,-136(s0)
10318:    f7042e23          sw      a6,-132(s0)

    c = cx;

1031c:    f7842783          lw      a5,-136(s0)
10320:    f6f42a23          sw      a5,-140(s0)

asm("pltrd_mc %[result1], %[value1], %[value2]\n\t"
    : [result1] "=r" (result1) : [value1] "r" (c) ,
    : [value2] "r" (prime));

10324:    f7442783          lw      a5,-140(s0)
10328:    fb842703          lw      a4,-72(s0)
1032c:    04e787b3          pltrd_mc a5,a5,a4
10330:    f8f42a23          sw      a5,-108(s0)

//////////////////////////////////// software

    result2 = (uint64_t) a*b % KYBER_Q;

10358:    f9c42783          lw      a5,-100(s0)
1035c:    00078b13          mv      s6,a5
10360:    00000b93          li      s7,0
10364:    f9842783          lw      a5,-104(s0)

```

10368:	00078c13	mv	s8,a5
1036c:	00000c93	li	s9,0
10370:	038b8733	mul	a4,s7,s8
10374:	036c87b3	mul	a5,s9,s6
10378:	00f707b3	add	a5,a4,a5
1037c:	038b0733	mul	a4,s6,s8
10380:	038b36b3	mulhu	a3,s6,s8
10384:	f6d42223	sw	a3,-156(s0)
10388:	f6e42023	sw	a4,-160(s0)
1038c:	f6442703	lw	a4,-156(s0)
10390:	00e787b3	add	a5,a5,a4
10394:	f6f42223	sw	a5,-156(s0)
10398:	007fe637	lui	a2,0x7fe
1039c:	00160613	add	a2,a2,1 #
↪ 7fe001 <_stack_start+0x77e001>			
103a0:	00000693	li	a3,0
103a4:	f6042503	lw	a0,-160(s0)
103a8:	f6442583	lw	a1,-156(s0)
103ac:	6da000ef	jal	10a86
↪ <__umoddi3> // This jal instruction goes to a function			
103b0:	00050713	mv	a4,a0
103b4:	00058793	mv	a5,a1
103b8:	f8e42823	sw	a4,-112(s0)

APPENDIX D : Makefile example

```
1 # Makefile to generate baremetal app
2
3 PROGRAM ?= newhope
4 PROGRAM_CFLAGS = -Wall -g -O1 -DSIMPLE_RNG -DBAM_CUST0 -DBAM_CUST1 -DBAM_CUST2
5 #ARCH = rv32imc
6 ARCH = rv32im
7 SRCS = $(PROGRAM).c $(wildcard ../../libs/soc/*.c) $(wildcard ../../libs/newhope512cca/*.c)
8 INCS = -I../../libs/soc -I../../libs/newhope512cca
9
10 PREFIX ?= riscv32-unknown-elf
11 CC := $(PREFIX)-gcc
12 OBJCOPY := $(PREFIX)-objcopy
13 OBJDUMP := $(PREFIX)-objdump
14
15 LINKER_SCRIPT ?= link.ld
16 CRT ?= crt0.S
17 CFLAGS ?= -march=$(ARCH) -mabi=ilp32 -static -mcmodel=medany \
18 | -fvisibility=hidden -ffunction-sections -fdata-sections \
19 | -nostdlib -nostartfiles $(PROGRAM_CFLAGS)
20 LDFLAGS ?= $(CFLAGS) -T $(LINKER_SCRIPT) -Wl,--gc-sections -Wl,--Map,$(PROGRAM).map
21
22 OBJS := $(SRCS:.c=.o) ${CRT:.S=.o}
23 DEPS = $(OBJS:%.o=%.d)
24
25 OUTFILES = $(PROGRAM).elf $(PROGRAM).mem $(PROGRAM).bin $(PROGRAM).dis
26
27 HEX2VMEM = ../../scripts/hex2vmem.pl
28 SENDAPP = ../../scripts/sendapp.sh
29
30 all: $(OUTFILES)
31
32 $(PROGRAM).elf: $(OBJS) $(LINKER_SCRIPT)
33 | $(CC) $(LDFLAGS) $(OBJS) -o $@ $(LIBS)
34
35 %.dis: %.elf
36 | $(OBJDUMP) -SD $^ > $@
37
38 %.mem: %.hex
39 | $(HEX2VMEM) $< > $@
40
41 %.hex: %.elf
42 | $(OBJCOPY) -O verilog --interleave-width=4 --interleave=4 --byte=0 $< $@
43
44 %.bin: %.elf
45 | $(OBJCOPY) -O binary $< $@
46
47 %.o: %.c
48 | $(CC) $(CFLAGS) -c $(INCS) -o $@ $<
49
50 %.o: %.S
51 | $(CC) $(CFLAGS) -c $(INCS) -o $@ $<
52
53 clean:
54 | $(RM) -f $(OBJS) $(DEPS) *.hex
55
56 distclean: clean
57 | $(RM) -f $(OUTFILES) *.map
58
59 run:
60 | $(SENDAPP) $(PROGRAM).bin
```

Figure D.1 : NTT software makefile example.

CURRICULUM VITAE



Name Surname: Ali Üstün

Place and Date of Birth: İstanbul, 28.08.1998

E-Mail: ustuna16@itu.edu.tr

EDUCATION:

- **B.Sc.:** Senior student, Istanbul Technical University, Electrical-Electronics Faculty, Department of Electronics and Communication Engineering
- **M.Sc.:** Electronics Engineering, Istanbul Technical University, Electrical-Electronics Faculty, Department of Electronics and Communication Engineering

PROFESSIONAL EXPERIENCE AND REWARDS:

- 2023 TUBITAK-BILGEM Expert Researcher
- 2020 TUBITAK-BILGEM Full-time Researcher
- 2020 TUBITAK-BILGEM Part-time Researcher
- 2020 Embedded Systems Design Laboratory Internship
- 2019 TUBITAK-BILGEM Internship
- 2017 Istanbul Technical University Solar Car Team - Embedded Systems
- 2016 Turkish National Physics Olympics